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. Efficiently scheduling data processing jobs on distributed compute clusters requires complex algorithms. Current systems, however, use simple generalized heuristics and ignore workload structure, since developing and tuning a bespoke heuristic for each workload is infeasible. In this paper, we show that modern machine learning techniques can generate highly-efficient policies automatically.

Decima uses reinforcement learning (RL) and neural networks to learn workload-specific scheduling algorithms without any human instruction beyond specifying a high-level objective such as minimizing average job completion time. 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 on a 25-node cluster shows that Decima outperforms several heuristics, including hand-tuned ones, by at least 21%. Further experiments with an industrial production workload trace demonstrate that Decima delivers up to a 17% reduction in average job completion time and scales to large clusters.

1 Introduction

Efficient utilization of their expensive compute clusters matters for enterprises: even small improvements in utilization can save millions of dollars at scale [6, §1.2]. Cluster schedulers are key to realizing these savings. A good scheduling policy packs work tightly to reduce fragmentation [18; 19; 46], prioritizes jobs according to high-level metrics such as user-perceived latency [47], and avoids inefficiencies due to incorrect job configurations [14].

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. By using general heuristics like fair scheduling [3; 15], shortest-job-first, and simple packing strategies [18], current systems forego potential performance optimizations. For example, widely-used schedulers do not use readily available information about job structure (i.e., the internal dependency graph) to make better decisions. Unfortunately, the workload-specific scheduling policies that could use this information require expert knowledge and take significant effort to devise, implement, and validate. For many organizations, these skills are either unavailable or uneconomic because the labor cost exceeds potential savings.

In this paper, we show that modern machine-learning techniques can help side-step this trade-off by *automatically learning* highly efficient, workload-specific scheduling policies. We present Decima[1], 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) [5; 9; 23; 50]; and (*ii*) scheduling DAGs is a hard algorithmic problem whose optimal solutions are intractable and difficult to capture in good heuristics [19].

Given only a high-level goal (e.g., minimal average job completion time), Decima uses existing cluster monitoring information and past workload logs to automatically learn sophisticated scheduling policies. For example, instead of a rigid fair sharing policy, Decima learns to give jobs shares of resources that optimize overall performance; it learns to use jobs' dependency structure to plan ahead and avoid waiting at "choke points"; and it learns job-specific parallelism levels that avoid wasting resources on diminishing returns for jobs with little inherent parallelism. The right algorithms and thresholds for these policies are all workload-dependent, and achieving them with a current scheduler requires painstaking manual customization.

Decima learns scheduling policies *entirely from experience* using modern deep reinforcement learning (RL) techniques. Decima uses a neural network to encode its scheduling policy; it trains this neural network through a large number of simulated experiments, where it schedules a workload, observes the outcome, and gradually improves its policy. We built Decima using neural networks and RL because these techniques have achieved remarkable recent success on challenging decision-making tasks, such as learning Go and Chess purely through self-play [41]. To apply these techniques to complex cluster scheduling problems, however, we had to solve several key problems.

First, 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 (§5.1). Our approach builds upon recent work on learning graph embeddings [10; 11; 26], but is tailored to the scheduling domain.

[1] Decima makes decisions and spins threads of life in Roman mythology.

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 (each with dozens of stages) and thousands of machines, and must decide among potentially hundreds of configurations per job (e.g., different levels of parallelism). 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*).[2] We therefore had to design a scalable RL formulation, and our models include several innovations (§5.2). For example, our neural network architecture processes any number of jobs with the same underlying parameters. We also 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. These scalability innovations affect both the models' ability to learn successfully and the performance of training and decision making.

Third, continuous, streaming job arrivals introduce undesirable variance that the conventional RL training approaches cannot tolerate [42, §3.7]. This variance exists because conventional RL training algorithms cannot tell whether two reward values 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 [30]. 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 (§5.3).

We integrated Decima with Spark and evaluated it on both an experimental testbed and an industrial workload trace. Our evaluation (§7) shows that (*i*) Decima outperforms existing heuristics on a 25-node Spark cluster, reducing average job completion time of TPC-H query mixes by 21% or more; (*ii*) Decima can offer efficiency gains of 5–17% on a production workload from a large (anonymized) company; and (*iii*) using the same trace, Decima's core learning framework can easily adapt to packing resources in multiple dimensions (CPU and RAM), achieving a 37% improvement over meticulously designed heuristics such as Graphene's [19].

In summary, we make the following key contributions:

1. A novel scalable graph processing technique that converts job DAGs of arbitrary shape and size into vectors suitable for neural network and end-to-end RL (§5.1–§5.2).
2. A set of variance reduction techniques to make RL training feasible for unbounded job arrival sequences (§5.3).
3. Decima, the first generalizable, RL-based scheduler that schedules complex data processing jobs and learns high-quality scheduling policies without human-encoded input, and a prototype implementation of it (§6).
4. An evaluation of Decima both in simulation and in a real Spark cluster, and a comparison with state-of-the-art scheduling heuristics (§7).

[2] For example, the state of the game of Go can be represented by $19 \times 19 = 361$ numbers, which also bound the number of legal moves per turn.

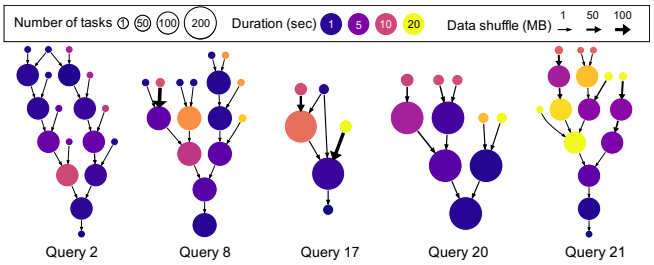


Figure 1: Data-parallel jobs have complex data-flow graphs like the ones shown (TPC-H queries in Spark), with each node having a distinct number of tasks, task durations, and input/output sizes.

2 Motivation

Data processing systems and query compilers such as Hive, Pig, SparkSQL, and DryadLINQ create *DAG-structured* jobs, which consist of processing stages connected by input/output dependencies (Figure 1). For recurring jobs, which are common in batch-processing clusters [2], it may also have reasonable estimates of runtimes and intermediate data sizes. Most cluster schedulers, however, ignore this job structure in their decisions and use e.g., coarse-grained fair sharing [3; 7; 15; 16], rigid priority levels [47], and manual encoding of each stage's parallelism in the job specification [39, §5]. Existing schedulers choose to largely ignore this rich, easily-available job structure information because designing scheduling algorithms that make use of it is a complex task. We illustrate the challenges of using job-specific information in scheduling decisions with two concrete examples: (*i*) dependency-aware scheduling, and (*ii*) automatically choosing the right number of parallel tasks.

2.1 Dependency-aware task scheduling

Many job DAGs in practice have tens or hundreds of stages with varying durations and numbers of parallel tasks in a complex dependency structure. An ideal schedule ensures that independent stages run in parallel as much as possible, and that no stage ever blocks on a dependency if there are available resources. Ensuring this requires the scheduler to understand the dependency structure and plan ahead. Writing a heuristic to this end is feasible, albeit non-trivial, even for a single DAG. However, a cluster runs multiple DAGs in parallel; an ideal scheduler must hence figure out a *combined* schedule for all of them. Such a schedule shifts resources between DAGs and their parallel stages as necessary to always keep them busy, while completing stages at a rate that avoids "choke points". This "DAG packing problem" is algorithmically hard: see, e.g., the illustrative example by Grandl et al. [19, §2.2] and the one we describe in detail in Appendix A. Hence, writing a heuristic to gen-

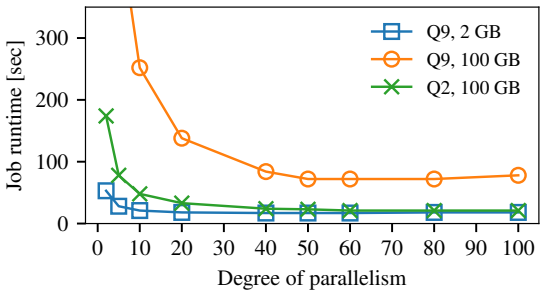


Figure 2: TPC-H queries scale differently with parallelism: Q9 on a 100 GB input sees speedups up to 40 parallel tasks, while Q2 stops gaining at 20 tasks; Q9 on a 2 GB input needs only 5 tasks. Picking "sweet spots" on these curves for a mixed workload is non-trivial.

erate optimal schedules for all possible DAG combinations is intractable [19; 31]. Existing schedulers therefore ignore this challenge: they enqueue tasks from a stage as soon as it becomes available, or run stages in an arbitrary topological order.

2.2 Setting the right level of parallelism

In addition to understanding dependencies, an ideal scheduler must also understand how to best split limited resources between jobs. Jobs vary in the amount of data that they process, and in the amount of parallel work available when different stages run. A job with large input or large intermediate data can efficiently harness additional parallelism; by contrast, a job running on small input data, or one with less efficiently parallelizable operations, sees diminishing returns beyond modest parallelism.

Figure 2 illustrates this with the job runtime of two TPC-H [43] queries running on Spark as they are given additional resources to run more parallel tasks. Even when both process 100 GB of input, Q2 and Q9 exhibit widely different scalability: Q9 sees significant speedup up to 40 parallel tasks, while Q2 only obtains marginal returns beyond 20 tasks. When Q9 runs on a smaller input of 2 GB, however, it needs no more than ten parallel tasks. For all jobs, assigning additional parallel tasks beyond a "sweet spot" in the curve adds only diminishing gains. Hence, the scheduler should reason about which job will see the largest marginal gain from extra resources and accordingly pick the sweet spot for each job. In principle, this is a straightforward application of Amdahl's Law [21], but given a mixed workload, the specific curves and sweet spots are difficult to predict.

Existing schedulers largely side-step this problem. Most burden the user with the choice of how many parallel tasks to use [39, §5], or rely on a separate "auto-scaling" component based on coarse heuristics [4; 14]. Indeed, many fair scheduling policies [15; 24] divide resources without paying attention to their decisions' efficiency: sometimes, an "unfair" schedule results in a more efficient overall execution.

2.3 An illustrative example on Spark

The aspects described are just two examples of how schedulers can exploit knowledge of the workload. To achieve the best performance, schedulers must also respect other considerations, such as the execution order (e.g., favoring short jobs) and avoiding resource fragmentation [18; 47]. Considering all these dimensions together — as Decima does — makes a substantial difference. We illustrate this by running a mix of ten randomly chosen TPC-H [43] queries with input sizes drawn from a long-tailed distribution on a Spark cluster with 50 parallel task slots.[3] Figure 3 visualizes the schedules imposed by *(a)* Spark's naïve FIFO scheduling; *(b)* a shortest-job-first (SJF) policy that strictly prioritizes short jobs; *(c)* a more realistic, fair scheduler that dynamically divides task slots between jobs; and *(d)* a scheduling policy learned by Decima. We measure average job completion time (JCT) over the ten jobs. Having access to the graph structure helps Decima improve average JCT by 45% over the naïve FIFO scheduler, and by 19% over the fair scheduler. It achieves this speedup (*i*) by completing short jobs quickly, as five jobs finish in the first 40 seconds; and (*ii*) by maximizing parallel-processing efficiency. SJF naïvely dedicates all task slots to the next-smallest job in order to finish it early (but inefficiently); by contrast, Decima runs jobs near their parallelism sweet spot. By controlling parallelism, Decima reduces the total time to complete all jobs by 30% compared to SJF. Further, unlike fair scheduling, Decima partitions task slots non-uniformly across jobs, improving average JCT by 19%.

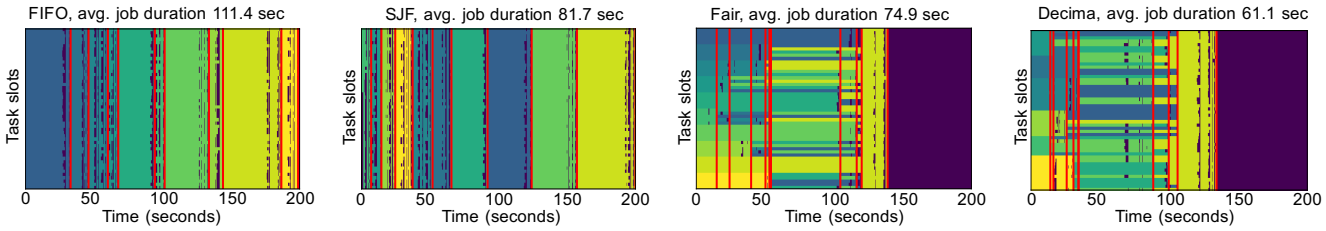
Designing general-purpose heuristics to achieve these benefits is difficult, as each additional dimension (DAG structure, parallelism, job sizes, etc.) increases complexity and introduces new edge cases. It may be feasible to tune a heuristic for a specific workload, but rarely happens in practice, since devising, implementing, and testing a scheduling policy requires expert knowledge and significant effort. Decima opens up a new option: using data-driven techniques, it *automatically* learns workload-specific policies that can reap these gains. Decima does so without requiring human guidance beyond a high-level goal (e.g., minimal average JCT), and without explicitly modeling the system or the workload.

3 The DAG scheduling problem in Spark

Decima is a general framework for learning scheduling algorithms for DAG-structured jobs. For concreteness, we describe its design in the context of the Spark system.

A Spark job consists of a DAG whose nodes are dependent *stages*. Each stage represents an operation that the system runs in parallel over many shards of the stage's inputs. The inputs are the outputs of one or more parent stages, and each shard is processed by a single *task*. A stage's tasks become runnable as soon as all parent stages have completed.

[3] See §7 for details of the workload and our cluster setup.



(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.

How many tasks can run in parallel depends on the number of *executors* (i.e., parallel task slots) that the job holds. Usually, a stage has more tasks than there are executors, and the tasks therefore run in several "waves". Executors are typically assigned by the Spark master based on user-requests (and subject to available resources in the cluster). However, Spark's scheduling design also supports "dynamic resource allocation", which dynamically allocates executors based on pending tasks' wait time [4].

Spark must therefore handle three kinds of scheduling decisions: (*i*) deciding how many executors to give to each job; (*ii*) deciding which stages' tasks to run next, and (*iii*) deciding which task to run next when an executor becomes idle. When a stage completes, its job's high-level *DAG scheduler* handles the activation of dependent child stages and enqueues their tasks with a lower-level *task scheduler*. The task scheduler maintains task queues from which it assigns a task every time an executor becomes idle.

We allow the scheduler to move executors between job DAGs as it sees fit (dynamic allocation). Decima thus focuses on DAG scheduling (i.e., which stage to run next) and executor allocation (i.e., each job's degree of parallelism). Since tasks in a stage run identical code and request identical resources, we use Spark's existing task-level scheduling.

4 Overview and Design Challenges

Decima represents the scheduler as an agent that uses a neural network to make decisions. 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*. At a high level, the state captures the status of the DAGs in the scheduler's queue and the executors, while the actions assign executors to work on different DAG stages across time.

Decima trains its neural network through a large number of offline (simulated) experiments, where it attempts to schedule a workload, observes the outcome, and uses a RL algorithm to gradually improve the scheduling policy. To guide the RL algorithm, Decima gives the agent a *reward* after each action based on a high-level scheduling objective. The goal of the RL algorithm is to maximize the total sum of rewards. For example, if the objective is to minimize average JCT, Decima penalizes the agent $-\tau \times J$ 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 [28]. For a set of jobs, these penalties add up to the sum of the job completion times; hence the agent learns to minimize average JCT.

Decima's RL framework (Figure 4) is general and it can be applied to a variety of systems and objectives. For concreteness, we describe the design for scheduling DAGs on a set of executors to minimize average JCT. Our results in §7 will show how to apply the same design to schedule multiple resources (e.g., CPU and memory), optimize for other objectives like makespan [37], and learn qualitatively different polices depending on the underlying system artifacts (e.g., different costs for moving jobs between machines).

Challenges. Decima's design tackles three key challenges:

1. **Scalable information processing.** The scheduler must consider a large amount of information to make scheduling decisions: hundreds of job DAGs, each with dozens of stages, and executors with different job locality. Processing all of this information via neural networks is challenging, particularly because neural networks usually require fixed-sized numerical vectors as input.
2. **Efficient encoding of scheduling decisions as actions.** The scheduler effectively maps runnable stages to available executors. The exponentially large space of possible mappings poses a challenge for RL algorithms, which must "explore" the action space to learn a good policy.
3. **Handling continuous stochastic job arrivals**. Jobs continuously arrive to the scheduler, and randomness in the job arrival pattern can lead to large performance variations that make RL training difficult. For example, a period of high load is likely to increase JCT, reducing the agent's rewards compared to a low-load period. But such variations in reward have little to do with scheduling decisions, and therefore create noise for RL algorithms that seek to identify good actions based on the rewards.

5 Design

This section describes Decima's design, structured according to how it address the three aforementioned challenges: scalable processing of job DAGs (§5.1), encoding scheduling decisions as actions (§5.2), and RL training with continuous

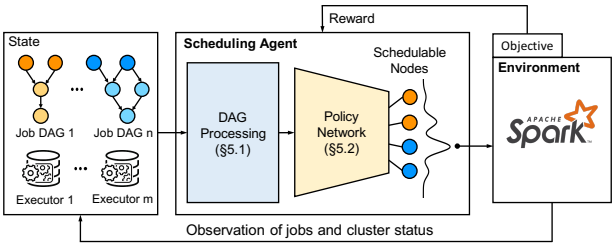


Figure 4: In Decima's RL framework, a *scheduling agent* observes the *cluster state* to decide on a scheduling *action* to invoke on the *environment* (the cluster), and receives a *reward* based on a high-level objective. The agent uses a *graph embedding* to turn job DAGs into vectors for *policy neural networks*, which output actions.

entity	symbol	entity	symbol
job	i	per-node embedding	$\mathbf{e}_v^i$
stage (DAG node)	v	per-job embedding	$\mathbf{y}^i$
job i's DAG	G_i	global embedding	$\mathbf{z}$
per-node feature vector	$\mathbf{x}_v^i$	non-linear functions	f, g, q

Table 1: Notation used throughout §5.

stochastic job arrivals (§5.3).

5.1 Scalable DAG processing

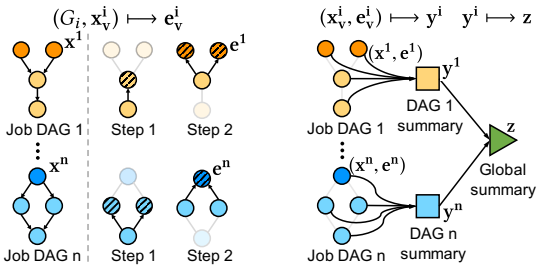
On each state observation, Decima converts the DAGs (of arbitrary shapes and sizes) to vectors using a novel *graph embedding* technique. The goal of the graph embedding is to encode or "embed" information about the DAG (e.g., stage attributes, parent-child relationships, etc.) in these vectors. Our method is based on graph convolutional neural networks [26], but it is customized for scheduling. Table 1 defines the notation we use.

The graph embedding takes as input the job DAGs whose nodes carry *features* in the form of stage attributes (e.g., the number of remaining tasks, expected task duration, etc.), and outputs three different types of embeddings:

1. per-node embeddings, which capture the graph structure by embedding information about the node and its children (containing, e.g., aggregated work along critical path);
2. per-job embeddings, which aggregate information across an entire job DAG (containing, e.g., the total work in the job); and
3. a global embedding, which combines information from all job-level summaries into a cluster-level summary (containing, e.g., the current cluster load).

Importantly, what information to store in these embeddings is not hard-coded—Decima automatically learns what is statistically important and how to compute it from the input DAGs through end-to-end training. In other words, the embeddings can be thought of as feature vectors that the graph embedding learns without manual feature engineering.

Per-node vectors. 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, starting from the leaves of the DAG (Figure 5a). In each message passing step, a node v whose children have aggregated messages from all of their children (shaded nodes in Figure 5a's examples) computes its own embedding as:



(a) Per-node embedding. **(b)** Summarizations.

Figure 5: *Graph embedding* transforms raw information each node of job DAGs into a vector representation. This example shows two steps of local message passing and two levels of summarizations.

$$\mathbf{e}_v = g\left[\sum_{w \in \xi(v)} f(\mathbf{e}_w)\right] + \mathbf{x}_v, \tag{1}$$

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 first term is a general, non-linear aggregation operation that summarizes the embeddings of v's children; adding this summary term to v's feature vector ($\mathbf{x}_v$) yields the embedding for v.

The key to a *scalable* embedding, i.e., one that works for DAGs of any size and shape, is to use the same non-linear transformations $f(\cdot)$ and $g(\cdot)$ at all nodes, and in all message passing steps. Equation (1) meets this requirement by using a sum to aggregate information across the children nodes; hence it can be applied to any number of children in $\xi(v)$.

Taking a sum across neighbors is a common technique in graph embeddings [10; 11; 26]. A typical approach is to compute the embedding of a node using an operation of the form $\mathbf{e_v} = \sum_{w \in \xi(v)} f(\mathbf{e_w})$. However, we found that adding the second non-linear transformation $g(\cdot)$ in Eq. (1) is critical for learning strong scheduling policies. The reason is that without $g(\cdot)$, the aggregation operation cannot express some useful computations for scheduling. For example, it cannot compute the critical path [25] of a DAG, which requires computing the maximum of values across the children of each node. 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.

Per-job and global vectors. 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, $\{\mathbf{y^1}, \mathbf{y}^2, \ldots\} \longmapsto \mathbf{z}$. To compute these summaries, Decima adds DAG-level summary nodes (squares in Figure 5b) that have all the nodes in their DAG as children, and which are in turn children of a single global summary node (the triangle in Figure 5b). These summarizations also use generic non-linear transformations similar to local message passing in Equation (1) to compute their embeddings. Each level of summarization has its own non-linear transformations (and thus, neural networks) f and g; in other words, the graph embedding uses six neural networks in total, two for each level of summarization.

5.2 Encoding scheduling decisions as actions

The key challenge of encoding scheduling decisions lies in the learning and computational complexities of dealing with large action spaces. As a naive approach, consider an "executor-centric" solution, which invokes the scheduling agent to pick a stage every time an executor becomes available. This provides exact control, but it requires long sequences of actions to schedule a given set of jobs. On the other extreme, a solution that partitions available executors between waiting jobs and returns the joint assignment in one shot has to choose from an exponentially large set of combinations. Large action spaces or long action sequences both require significantly more exploration during training and can make RL algorithms prohibitively slow [42].

Decima balances the size of the action space and the length of action sequences by decomposing 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.

Scheduling events. Decima invokes the scheduling agent when the set of runnable stages — i.e., stages whose parents are completed and which have at least one waiting task — in any job DAG changes. Such scheduling events happen when (*i*) a stage runs out of tasks (i.e., needs no more executors), (*ii*) a stage completes, unlocking the tasks of one or more of its children, or (*iii*) a new job arrives in the system.

At each scheduling event, the agent schedules a set of free executors in one or more actions. Specifically, it passes the processed vectors from §5.1 as input to a *policy neural network*, 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. If job i currently has fewer than l_i executors, Decima assigns executors to v up to the limit. If free executors remain, Decima invokes the agent again to select another stage and a parallelism limit. This process repeats until all the executors have been assigned, or there are no more runnable stages.

Stage selection. Figure 6 visualizes the stage selection process that occurs on a scheduling event. For a scheduling event at time step t, during which the system is in state s_t, the selection computes — using the graph embedding vectors $\mathbf{e}_v^j$, $\mathbf{y}^j$, and $\mathbf{z}$ — a score $q(\cdot)$ that encodes the priority of each node (i.e., stage) in a job DAG. Decima uses a standard "softmax" operation [8] to compute the probability of selecting node v in job $j(v)$ from the priority scores:

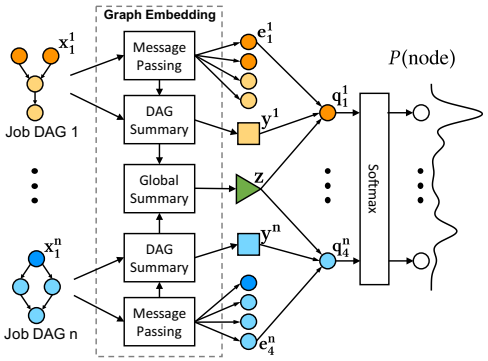


Figure 6: For each node v in job i, the *node selection network* uses the message passing summary $\mathbf{e}_v^i$, DAG summary $\mathbf{y}^i$ and global summary $\mathbf{z}$ to compute a priority score $\mathbf{q}_v^i$ used to sample a node.

$$P(a_t = v) = \frac{\exp\left\{q(\mathbf{e}_v^{j(v)}, \mathbf{y}^{j(v)}, \mathbf{z})\right\}}{\sum_{u \in \mathcal{A}_t} \exp\left\{q(\mathbf{e}_u^{j(u)}, \mathbf{y}^{j(u)}, \mathbf{z})\right\}}, \quad (2)$$

where $q : \{(\mathbf{e}_u^{j(u)}, \mathbf{y}^{j(u)}, \mathbf{z})\} \to \mathbb{R}$ denotes a non-linear mapping of the embedding vectors to a scalar value, implemented via a neural network that learns the priority function, and $\mathcal{A}_t$ is the set of nodes that can be scheduled at time t.

Parallelism limits on jobs. The stage selection model can decide the order in which stages are scheduled, but it cannot control the level of parallelism for a job. Thus, we augment Decima's 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. For maximum control, we could assign parallelism limits for each stage. But this 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. Therefore, the probability of choosing a node and a limit at state s_t can be simplified as:

$$\begin{aligned} P(\text{node}, \text{limit}|s_t) &= P(\text{node}|s_t) \times P(\text{limit}|\text{node}, s_t) \\ &= P(\text{node}|s_t) \times P(\text{limit}|\text{job}, s_t), \end{aligned} \quad (3)$$

where Equation (3) follows from the observation that the limit value is a property of the job and not the node. The action priorities can now be described by the pair of simple probability functions — $P(\text{node}|s_t)$ and $P(\text{limit}|\text{job}, s_t)$ — instead of one complex joint distribution function. Consequently Decima outputs actions via two samplings: first a

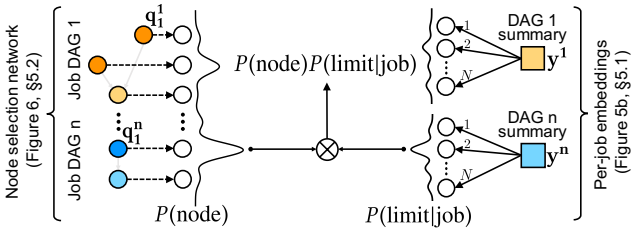


Figure 7: Decima's policy for jointly sampling a node and parallelism limit is implemented as the product of a node distribution, computed from graph embeddings (§5.1), and a limit distribution, computed from the DAG summaries.

node, and then a limit for the node's job (Figure 7). Concretely, node selection occurs using the graph embedding vectors as before (Figure 6). For limit selection, we use a similar model to the node selection (a non-linear score $q(\cdot)$ and then softmax) that acts based on the per-job embeddings of the respective jobs. In a later experiment, we demonstrate that this formulation significantly reduces Decima's inference time, i.e., scheduling decision latency (§7.5).

5.3 Training

All of the above operations, from graph embedding (§5.1) to selecting stages and parallelism limits (§5.2), are differentiable. Moreover, they are all part of the same (large) neural network, which we can train end-to-end using RL. The learning process consists of simulating multiple runs of scheduling experiments (§6.2). Each run (or "episode") begins with a sequence of jobs, and ends with completion of all jobs. We use standard *policy gradient* methods that have recently been applied to multiple domains [29; 36; 40] for our learning algorithm. The main idea in policy gradient is to learn by performing gradient descent on the policy parameters using empirically observed rewards.

Consider a sequence of interactions with the scheduling environment of length T, where the agent collects (*state, action, reward*) experiences, i.e., (s_t, a_t, r_t), at each step t. It then updates the parameter θ of its policy $\pi_\theta(s_t, a_t)$ (defined as the probability of taking action a_t in state s_t) using the well-known REINFORCE algorithm [49]:

$$\theta \leftarrow \theta + \alpha \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(s_t, a_t) \left(\sum_{t'=t}^{T} r_{t'} - b_t \right), \qquad (4)$$

where α is the step size and b_t is a *baseline* used to reduce the variance of the estimated gradient [48]. An example of a baseline is a "time-based" baseline [20; 28], which sets b_t to the cumulative reward from time t onwards, averaged over all training episodes. Intuitively, $(\sum_{t'} r_{t'} - b_t)$ estimates how much better (or worse) the total reward is in a particular episode compared to the average case; and $\nabla_\theta \log \pi_\theta(s_t, a_t)$ provides a direction to increase the probability of choosing action a_t at state s_t. As a result, the net effect of this equation is to reinforce the probability of choosing an action that leads to a large reward. We implement Decima's training framework using TensorFlow [1] and we defer the implementation details and hyperparameter settings in Appendix B.

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 $J_1, J_2, \ldots, J_N$ sampled from a training set, 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 $\mathbb{E}[\lim_{N\to\infty} \sum_{i=1}^{N} T(J_i)/N]$. We originally attempted to train using the standard sum-of-rewards objective. But we found that the scheduler learns to systematically defer large jobs when scheduling a finite set of jobs, as this results in the lowest sum of JCTs (highest sum of rewards). With continuous job arrivals, this scheduler starves the large jobs.

Fortunately, there is an alternative RL formulation that optimizes for the *average reward* in problems with an infinite time horizon [20; 42, §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 (as defined in §4) and $\hat{r}$ is a moving average of the rewards r_t across a large number of previous time steps (across many training iterations). Intuitively, such a reward signal 'normalizes' the sum completion times of jobs in an RL episode by the length of the episode to counter any bias arising due to varying episode lengths. The net result is an objective that closely correlates to the average number of concurrent jobs in the system. We refer the readers to Sutton and Barto [42, §10.3] for details on how this approach optimizes average reward.

For a policy to generalize well with stochastic job arrivals, the training episodes must include many different job arrival sequences.[4] This creates a second challenge: different job arrival patterns have a large impact on performance, and hence on the rewards. 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 feedback and distorts the policy gradient estimation in Equation (4).

To account for the variance caused by the arrival process, we build upon recently-proposed variance reduction techniques for "input-driven" environments [30]. 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 with a large mean (e.g., a few

[4] Training Decima in a "batch" setting without jobs arriving over time does not generalize to continuous job arrivals; see Figure 20 in Appendix H.

hundreds of job arrivals on average). We then compute the baseline at time step t, i.e., the value of b_t to use in Eq. (4), as the cumulative reward from t onwards, averaged over the rollouts of that particular job arrival sequence. This method enables us to correctly account for differences between rewards caused by different job arrival sequences, significantly improving the quality of the policy gradients (Figure 14b). The training repeats this procedure for a large number of randomly-sampled job arrival sequences.

These two techniques are crucial for Decima to achieve good performance in streaming setting. We also empirically evaluate the importance of each component in §7.5.

6 Implementation

We have implemented Decima as a pluggable scheduler service that parallel data processing platforms (e.g., Spark, Dryad, or YARN) can communicate with over an RPC interface. The Decima service consists of 1389 lines of Python, of which 647 lines comprise the TensorFlow model served. §6.1 describes our prototype integration of the Decima service with Spark.

In addition, our Python-based training infrastructure (§5) consists of 1654 lines of code and includes a faithful cluster simulator (§6.2), which we drive with TPC-H inputs and a workload trace from a large company's production clusters.

6.1 Spark integration

A Spark cluster[5] runs multiple parallel *applications*, which contain one or more "jobs" that together form a DAG of processing stages. The Spark master manages application execution and monitors the health of many *workers*, which each split their resources between multiple executors. Executors are created for, and remain associated with, a specific application, which handles its own scheduling of work to executors. Internally to an application, two levels of scheduling exist: the application's *DAG scheduler* chooses stages to work on and submits their tasks to a lower-level *task scheduler* that maps running stages' fine-grained parallel *tasks* to executors. Once an application completes, its executors terminate. Figure 8 illustrates this architecture.

To integrate Decima in Spark, we made two major changes:

1. Each application's **DAG scheduler** contacts Decima on startup and whenever a scheduling event occurs. Decima responds with the next stage to work on and the parallelism limit.
2. The Spark **master** contacts Decima when a new job arrives to determine how many executors to launch for it, and whether to preempt any existing executors.

Our changes amount to a 901-line patch to Spark v2.2.

State observations. In Decima, the feature vector (§5.1) $\mathbf{x}_v^i$ of a node v in job DAG i consists of: (*i*) the number of tasks remaining in the stage, (*ii*) the average task duration, (*iii*) the number of executors currently working on the node, (*iv*) the number of available executors, and (*v*) whether available executors are local to the job. These features are *raw*, i.e., chosen without a human design intention. However, they intuitively capture some useful information. For example, the task count and average task duration can help estimate the remaining work in a stage. To learn a good scheduling policy, Decima must learn to exploit this raw information.

[5] We discuss Spark's "standalone" mode of operation here (http://spark.apache.org/docs/latest/spark-standalone.html); YARN-based deployments can, in principle, use Decima, but require modifying both Spark and YARN.

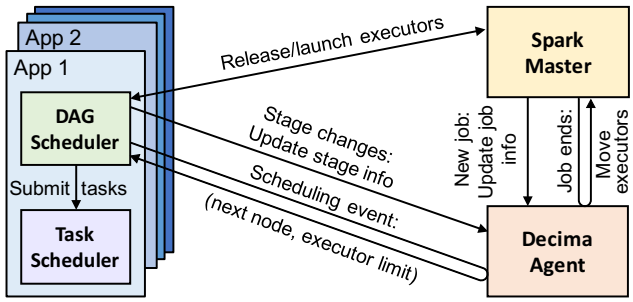


Figure 8: Spark standalone cluster architecture, with Decima additions highlighted.

6.2 Faithful simulation

Decima's training happens offline using a faithful simulator that has access to profiling information (e.g., task durations) from a real Spark cluster (§7.2) and the job run time characteristics from an industrial trace (§7.3). To faithfully represent the effect of Decima's decisions on a cluster environment, our simulator models several real-world effects:

1. The first "wave" of tasks from a particular stage often runs slower than subsequent tasks. This is due to Spark's pipelined task execution [35], JIT compilation [27] of task code, and warmup costs (e.g., making TCP connections to other executors). Decima's simulated environment thus picks the actual runtime of first-wave tasks from a different distribution than later waves.
2. Adding an executor to a Spark job involves launching a JVM process, which takes 2–3 seconds. Executors are tied to a job for isolation and because Spark assumes them to be long-lived. Decima's environment therefore imposes idle time reflecting the startup delay every time Decima moves an executor across jobs.
3. A high degree of parallelism can *slow down* individual Spark tasks, as wider shuffles require additional TCP connections and create more work when merging data from many shards. Decima's environment captures these effects by picking task durations from distributions sampled at different levels of parallelism if this data is available.

The Decima agent is initially unaware of these effects, it learns to anticipate them in its decisions durings training. In Appendix C, we validate the fidelity of our simulator by comparing it with real Spark executions (e.g., Figure 2).

7 Evaluation

We evaluated Decima on a real Spark cluster testbed and in simulations with a production workload from a large company. Our experiments address the following questions:

1. How does Decima perform compared to human-engineered heuristics in a real Spark cluster, both on batch and streaming job arrivals? We find that Decima outperforms all schemes we compare to in both settings, achieving 21% to 3.1× lower average JCTs (§7.2).
2. Does Decima offer improvements over existing heuristics on a real-world industry workload? Our simulated results suggest that Decima consistently rivals or outperforms existing schemes for this realistic workload (§7.3).
3. Can Decima's learning framework generalize to other scheduling problems — e.g., packing multi-dimensional resources such as CPU and memory? In an augmented multi-resource Spark environment, we find that Decima uses resources efficiently and outperforms state-of-the-art heuristic like those in Graphene [19] (§7.4).
4. What polices does Decima learn, and how does each of our key ideas contribute to Decima's performance? We find that Decima learns a range of qualitatively different policies for different goals, and that all key ideas matter to achieving good performance in practice (§7.5).

7.1 Existing baseline algorithms

In our evaluation, we compare Decima's performance to that of seven baseline algorithms:

1. Spark's default, naïve FIFO scheduling, which runs jobs in the same order they arrive in and grants as many executors to each job as the user requested.
2. A shortest-job-first critical-path heuristic (SJF-CP), which prioritizes jobs based on their total work (in task-seconds), and within each job runs tasks from the next stage on its critical path.
3. Simple fair scheduling, which gives each job an equal fair share of the executors and round-robins over tasks from runnable stages to drain all branches concurrently.
4. Naïve weighted fair scheduling, which assigns executors to jobs proportional to their total work.
5. A carefully-tuned weighted fair scheduling, which gives each job $T_i^\alpha / \sum_i T_i^\alpha$ of total executors, where T_i is the total work of each job i and α is a tuning factor. Notice that $\alpha = 0$ reduces to simple fair scheme and $\alpha = 1$ means naive weighted fair. We sweep through $\alpha \in \{-2, -1.9, ..., 2\}$ for the optimal factor.
6. The standard multi-resource packing algorithm from Tetris [18], which greedily schedules the stage that maximizes the dot product of the requested resource vector and the available resource vector.
7. Graphene*, an adaptation of Graphene [19] for Decima's discrete executor classes. Graphene* detects and groups "troublesome" nodes using Graphene's algorithm [19, §4.1], but keeps them unscheduled until *all* trouble-some nodes are runnable, thus achieving the essence of Graphene's planning strategy (cf. Appendix D).

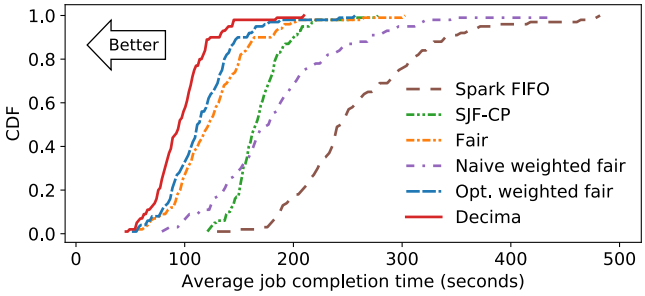


Figure 9: Decima's learned scheduling policy achieves 21%–3.1× lower average job completion time than baseline algorithms for 100 batches of 20 concurrent TPC-H jobs in a real Spark cluster.

7.2 Spark cluster

We use a cluster running Spark v2.2, modified as described in §6.1. The cluster runs on OpenStack in the Chameleon Cloud testbed[6], and consists of 25 worker VMs, each running two executors on an `m1.xlarge` instance (8 CPUs, 16 GB RAM) and a master VM on an `m1.xxxlarge` instance (16 CPUs, 32 GB RAM). We run experiments for two arrival processes: (*i*) *batched*, in which a batch of multiple jobs arrives, and (*ii*) *continuous*, in which jobs arrivals follow a stochastic inter-arrival time distribution.

Batched arrivals. We randomly sample jobs from six different input sizes (2, 5, 10, 20, 50, and 100 GB) and all 22 TPC-H [43] queries, producing a heavy-tailed distribution: 23% of the jobs contain 82% of the total work. A combination of 20 jobs (unseen in training) arrives as a batch at the start of the experiment (i.e., no further jobs arrive), and we measure their average JCT. Figure 9 shows a cumulative distribution of the average JCT achieved over 100 experiments.

There are three key observations. First, SJF-CP and fair scheduling, albeit simple, outperform the naïve FIFO policy by 1.6× and 2.5× on average. This is expected, since these heuristics prioritize shorter jobs and constraint the parallelism of each job, while FIFO grants the entire cluster.

Second, the fair scheduling policies outperform SJF-CP since they work on multiple jobs, while SJF-CP focuses exclusively on the shortest job. Perhaps surprisingly, unweighted fair scheduling outperforms fair scheduling weighted by job size ("naïve weighted fair"). This is because weighted fair scheduling grants small jobs *fewer* executors than their fair share, slowing them down and increasing average JCT. Our tuned weighted fair heuristic ("opt. weighted fair") counters this effect by calibrating the weights for each job *on each experiment* (as per §7.1). The best α is usually around -1, i.e., the heuristic grants executors inversely proportional to job size. This policy effectively fair-shares the cluster between small jobs in the beginning, but later works on large jobs in parallel; it outperforms naïve fair

[6] https://www.chameleoncloud.org

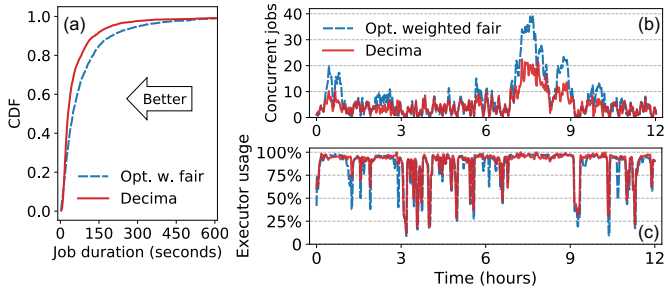
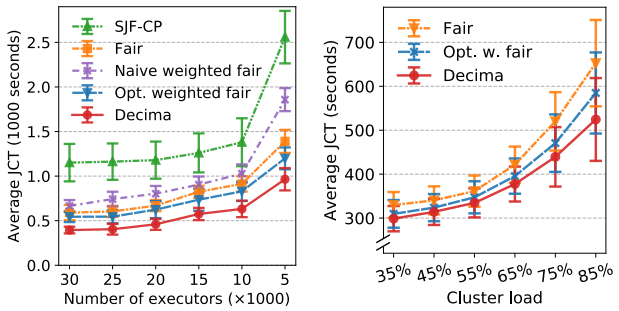


Figure 10: Streaming job arrival of 1,000 TPC-H jobs over 12 hours in a real Spark cluster. Decima achieves 29% better average JCT than the best heuristic (other heuristics cannot keep up) and has fewer active jobs at most points in time.



(a) Batch setting. **(b)** Streaming setting.

Figure 11: On a simulated industrial workload, Decima outperforms all baselines on randomly-sampled batches of 150 real-world industry jobs at different load levels (a); and replaying the industrial trace directly for different effective loads, Decima outperforms the best alternative by 5–12%; algorithms not shown are unstable with continuous arrivals (b). Error bars: ± one standard deviation.

scheduling by 11% on average. These results illustrate the difficulty of balancing the conflicting goals of prioritizing small jobs to improve average JCT, and dividing the cluster fairly while running jobs at efficient parallelism levels.

Third, 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 (§7.5 breaks down the benefit of each of these factors). Decima learned this highly efficient scheduling policy completely on its own through end-to-end RL training here, while the best-performing baseline algorithms required careful tuning.

Continuous arrivals. We randomly sample 1,000 TPC-H jobs, and model their arrival as 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 (§5.3) and evaluate with an unseen sequence of job arrivals.

A busy period 8h into the experiment causes some scheduling policies to fall behind as they cannot finish jobs fast enough. Figure 10 shows the results for Decima and the only baseline algorithm that can keep up ("opt. weighted fair"). Decima achieves a 29% better average JCT than the carefully-tuned weighted fair scheduler (Fig. 10a). Moreover, Decima uses the executors more efficiently: it has a higher executor usage (Fig. 10b) and consistently maintains a lower active job count (Fig. 10c) as it completes jobs sooner.

7.3 Industrial workload

To evaluate Decima's ability to learn scheduling policies that benefit complex, real-world workloads, we evaluate it using a workload trace from a large company. The one month-trace includes about 20,000 jobs in a batch-processing cluster. Many jobs have complex DAGs: 59% have four or more stages, and some are as large as hundreds of stages. On average, the cluster runs around 150 concurrent jobs that use about 60% of the peak resource consumption. We run the experiments using our simulator (§6.2). Since the trace contains no information about cluster capacity, we use the maximum number of concurrently active tasks (27,986) to guide our setup and simulate cluster sizes up to 30,000 executors.

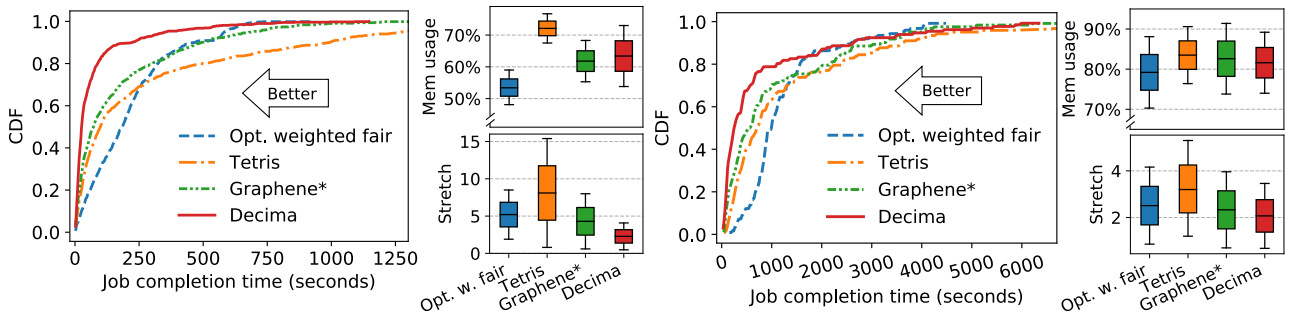
Job batches. We synthesize batches of real-world jobs to compare the relative performance of Decima and other algorithms to the TPC-H experiment in a setting that allow all algorithms to finish the workload. We sample sets of 150 jobs from the first 2/3 of the trace and train models with different numbers of executors, representing different cluster loads. Figure 11a shows the average JCT over 100 experiments, testing on the remaining (unseen) 1/3 of jobs. Decima consistently outperforms all baseline algorithms, improving by 12–17% over hand-tuned weighted fair scheduling, a result similar to the one for the TPC-H DAGs.

Trace replay. Next, we replay the continuous job arrival pattern recorded in the real-world cluster. In the actual industrial cluster, the average JCT of these jobs was 396.1 seconds. We successively shrink the cluster similar to increase the effective load. For each load, we train Decima on only the first half of the trace. Specifically, each training iteration randomly samples a starting point in the first half of the trace and simulates the next 1,000 job arrivals.

During testing, we evaluate each algorithm on the second half of the trace (unseen by Decima during training). Figure 11b shows the average JCTs they achieve. First, Decima outperforms the best baseline heuristics by 5–12% for all cluster loads we simulated. Second, at 45% cluster load Decima achieves the same average JCT (396s) as the real cluster, despite having only 20k executor slots available (vs. up to 28k parallel tasks in the real cluster). Moreover, Decima's gain over other algorithms grows as the cluster load increases and the scheduling problem becomes harder.

7.4 Multi-dimensional resource packing

Many cluster managers allow users to specify task resource requests using two or more resource dimensions (e.g., ⟨CPU,



(a) TPC-H workload.

(b) Industrial trace replay.

Figure 12: In a multi-resource scheduling environment, Decima learns a policy that outperforms Graphene* by 37% to 53% in average JCT. "Memory usage" (task memory/executor memory) indicates how much memory is wasted due to fragmentation; Decima performs similarly to Graphene*. The "stretch" of a job is the JCT normalized to its JCT on an otherwise idle cluster; Decima consistently has the lowest stretch.

memory⟩). Packing tasks efficiently in multiple resource dimensions requires complex heuristics [18; 19]. We investigate if, after modifying only the information it observes, Decima learns a good multi-dimensional resource packing policy using the same core approach as before.

Multi-resource environment. We modify our environment to provide executors of several discrete *classes* with different memory sizes. A DAG stage's tasks now require a minimum amount of CPU and memory, i.e., a task must fit into the executor that runs it. Tasks can run in executors larger than or equal to their resource request. The Decima scheduler now chooses a DAG stage to schedule, a parallelism level, and an executor class to use. The discretization to fixed executor classes is needed to maintain Decima's discrete action space; we find that it imposes only a moderate overhead over continuous resource allocations of arbitrary sizes.

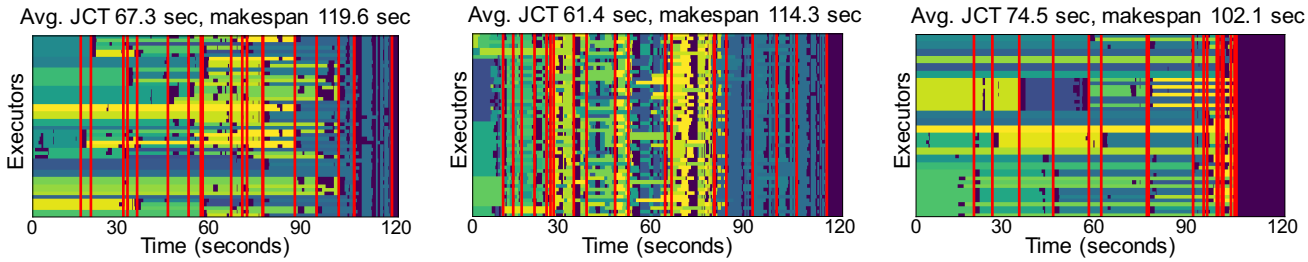
Results. We run simulated multi-resource experiments for both TPC-H and the industrial trace. The experiments use four executor types, each with 1 CPU core and $(0.25, 0.5, 0.75, 1)$ unit of normalized memory; each executor class makes up 25% of total cluster executors. The TPC-H experiments use 200 total executors, and the industrial trace experiment uses 30,000 executors. In TPC-H, we uniformly sample each DAG node's memory request from $(0, 1]$, while the industrial trace has resource requests. Figure 12 shows results for Decima and three other algorithms.

We make three key observations. First, Decima outperforms all other algorithms in both settings, achieving a 37%–53% lower average JCT. This suggests that Decima's learning techniques are sufficiently general to learn good policies in a complex, multi-dimensional resource environment. Second, since Tetris greedily packs the tasks into the best-fitting executor class, Tetris achieves the highest memory usage (i.e., lowest fragmentation) in both experiments. Decima, however, achieves significantly improved JCTs while maintaining a memory usage within 4%–13% of Tetris's. Thus, Decima learns to trade-off resource utilization for better JCTs (e.g., temporarily borrowing large executors to finish a nearly-completed job). Third, Decima achieves a significantly lower job stretch (i.e., JCT normalized to their runtime on an idle cluster) than all other algorithms, including next-best Graphene*. Graphene* tries to combine several concerns (DAG structure, priority for short jobs, etc.) via additive score functions, and does not control parallelism well, so Decima can outperform it by better fitting the workload. Decima tailors each job's parallelism and resource use to the current cluster utilization, ensuring that jobs operate at efficient parallelism levels without fragmenting resources.

7.5 Decima drill down

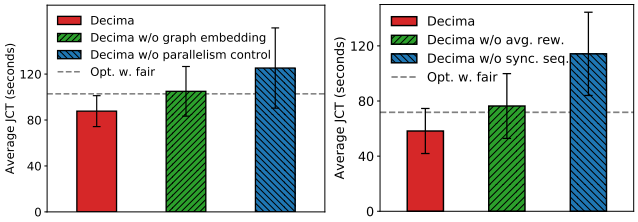
Finally, we demonstrate the wide qualitative range of scheduling policies Decima can learn, and break down the impact of our key ideas and scalable RL techniques on Decima's performance. In appendices, we further evaluate Decima's optimality via an exhaustive search of job orderings (Appendix E), its learned policies' robustness to changing environments (Appendix F), and Decima's sensitivity to incomplete information (Appendix G).

Learned policies. Decima outperforms other algorithms because it can learn different policies depending on the high-level objective, the workload, and environmental conditions. When Decima optimizes for average JCT (Figure 13a), it learns to share resources for small jobs to finish them quickly, but defers "going wide" on large jobs until the end. Decima also keeps the executors working on tasks from the same job to avoid the overhead of moving executors (§6.1). For a workload of long tasks that easily amortize this overhead, or if executors could run tasks from multiple jobs, executor motion is effectively free. In such a setting, Decima learns a policy that more eagerly moves executors among jobs (cf. the frequent color changes in Figure 13b), reducing both overall average JCT and makespan. Finally, given a different objective of minimizing the overall *makespan* for a batch of jobs, Decima learns yet another qualitatively different policy (Figure 13c). Since only the *final* job's completion



(a) Average JCT objective. **(b)** Avg. JCT, with zero-cost executor motion. **(c)** Minimal makespan objective.

Figure 13: Decima learns qualitatively different policies depending on the environment (e.g., costly (a) vs. free executor migration (b)) and the objective (e.g., average JCT (a) vs. makespan (c)). Red lines at job completions, colors indicate tasks in different jobs, dark purple is idle.



(a) Batch setting. **(b)** Streaming setting.

Figure 14: Breakdown of the contributions due to key ideas in Decima. Omitting any of these idea results in Decima underperforming the tuned weighted fair policy.

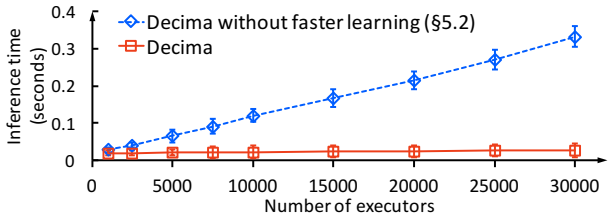


Figure 15: Without the domain-specific conditional probability insight (§5.2), Decima's inference time grows with cluster size.

time matters for a makespan objective, Decima no longer works to finish jobs early if given this objective. Instead, many jobs complete together at the end of the workload, which gives the scheduler more choices of jobs throughout the execution, increasing cluster utilization.

Contributions breakdown. To validate that Decima uses all the raw information provided in the state and benefits from all its key components, we consider variant models that each omit a piece of information.

We run an experiment with batched arrivals of 20 TPC-H jobs (as in §7.2) and using 50 executors. First, we remove the graph embedding by only supplying raw information on each node — i.e., skipping the message-passing step and using the raw feature vector on each node $\mathbf{x}$ directly (§5.1). Second, we use the model without parallelism control (§5.2). Figure 14a shows that removing either component degrades Decima's performance to be worse than the comparing heuristic. Disabling parallelism control has a larger impact (14% worse average JCT) than removing the graph embedding (8% worse), as the TPC-H DAGs are fairly simple, but choosing efficient parallelism levels is crucial.

Next, we repeat the continuous job arrival setup from §7.2 in simulation, but remove the key ideas needed for training on continuous arrivals (§5.3). Figure 14b shows the results: training with a regular reward results in a poorer learned policy that degrades performance by 32%. Further, disabling the synchronized termination of workload sequences in training increases variance, which doubles the average JCT. Hence, variance reduction is key to learning high-quality policies in long-horizon scheduling problems.

Scalability. The inference time of Decima's model is crucial to scheduling latency and must therefore scale well with cluster size. The original model in §5.2 — without our insight that $P(limit|node) = P(limit|job)$ — has an inference time of $O(\#nodes \times \#executors)$. Figure 15 shows that this grows to 400ms of scheduling delay for a cluster with 30k executors. Our domain-specific insight that the parallelism level is independent of node choice, however, reduces the inference time to a more scalable $O(\#nodes + \#executors)$.

8 Related Work

There is little prior work on applying machine learning techniques to cluster scheduling. DeepRM [28], which uses reinforcement learning to train a neural network for multi-dimensional resource packing, is closest to Decima in aims and approach. However, DeepRM's learning model only works with simple resource time simulator, and does not support real jobs involving graph structure. Mirhoseini et al.'s work on learning device placement in TensorFlow (TF) computations [33] also uses reinforcement learning. However, instead of a graph embedding, it relies on recurrent neural networks (RNNs), to scan through all nodes for RL inputs. Such model does not generalize the policy to unseen job combinations [32]. Moreover, the objective is to schedule a *single* TF job well, rather than a general job mix.

Paragon [12] and Quasar [13] use collaborative filtering to match workloads to difference machine types and avoid interference; their goal is complementary to Decima's. Tetrisched [44], like Decima, plans ahead in time, but uses a constraint solver to optimize job placement and requires

user to supply explicit constraints with their jobs. Firmament [17] also uses a constraint solver and achieves high-quality placements, but requires an administrator to configure an intricate scheduling policy. Graphene [19] use heuristics to schedule job DAGs, but cannot set appropriate parallelism levels. Some systems "auto-scale" parallelism levels to meet job deadlines [14] or opportunistically accelerate jobs using spare resources [39, §5]. As general-purpose cluster managers like Borg [47], Mesos [22], or YARN [45] support many different applications, workload-specific scheduling policies are difficult to apply at this level. However, Decima could run as a framework atop Mesos or Omega [39].

9 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 processing DAGs (e.g., query optimizers). We will open-source Decima, our models, and our experimental infrastructure.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. "TensorFlow: A System for Large-scale Machine Learning". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. Savannah, Georgia, USA, Nov. 2016, pp. 265–283.

[2] Sameer Agarwal, Srikanth Kandula, Nicolas Bruno, Ming-Chuan Wu, Ion Stoica, and Jingren Zhou. "Re-optimizing Data-parallel Computing". In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. San Jose, California, USA, 2012, pp. 281–294.

[3] Apache Hadoop. *Hadoop Fair Scheduler*. Accessed 13/03/2014. URL: http://hadoop.apache.org/common/docs/stable1/fair_scheduler.html.

[4] Apache Spark. *Spark: Dynamic Resource Allocation*. Spark v2.2.1 Documentation. URL: http://spark.apache.org/docs/2.2.1/job-scheduling.html#dynamic-resource-allocation (visited on 01/17/2018).

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

[6] Luiz André Barroso, Jimmy Clidaras, and Urs 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.

[7] Arka A. Bhattacharya, David Culler, Eric Friedman, Ali Ghodsi, Scott Shenker, and Ion Stoica. "Hierarchical Scheduling for Diverse Datacenter Workloads". In: *Proceedings of the 4th Annual Symposium on Cloud Computing (SoCC)*. Santa Clara, California, Oct. 2013, 4:1–4:15.

[8] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[9] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan 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.

[10] Hanjun Dai, Elias B. Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song. "Learning Combinatorial Optimization Algorithms over Graphs". In: *NIPS*. 2017.

[11] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. "Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering". In: *CoRR* (2016).

[12] Christina Delimitrou and Christos Kozyrakis. "Paragon: QoS-aware Scheduling for Heterogeneous Datacenters". In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '13. Houston, Texas, USA: ACM, 2013, pp. 77–88.

[13] Christina Delimitrou and Christos Kozyrakis. "Quasar: Resource-efficient and QoS-aware Cluster Management". In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '14. Salt Lake City, Utah, USA: ACM, 2014, pp. 127–144.

[14] Andrew D Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. "Jockey: guaranteed job latency in data parallel clusters". In: *Proceedings of the 7th ACM european conference on Computer Systems*. ACM. 2012.

[15] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. "Dominant Resource Fairness: Fair Allocation of Multiple Resource Types". In: NSDI'11. Boston, MA: USENIX Association, 2011, pp. 323–336.

[16] Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. "Choosy: max-min fair sharing for datacenter jobs with constraints". In: *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*. Prague, Czech Republic, Apr. 2013, pp. 365–378.

[17] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. "Firmament: fast, centralized cluster scheduling at scale". In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Savannah, Georgia, USA, Nov. 2016, pp. 99–115.

[18] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. "Multi-resource Packing for Cluster Schedulers". In: *SIGCOMM*. 2014.

[19] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. "Graphene: Packing and dependency-aware scheduling for data-parallel clusters". In: *OSDI*. USENIX Association. 2016, pp. 81–97.

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

[21] Mark D Hill and Michael R Marty. "Amdahl's law in the multicore era". In: *Computer* 41.7 (2008).

[22] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, et al. "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center". In: *NSDI*. 2011.

[23] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis 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.

[24] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. "Quincy: fair scheduling for distributed computing clusters". In: *ACM SIGOPS*. 2009.

[25] James E Kelley Jr and Morgan R Walker. "Critical-path planning and scheduling". In: *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*. ACM. 1959, pp. 160–173.

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

[27] Prasad A Kulkarni. "JIT compilation policy for modern machines". In: *ACM SIGPLAN Notices*. Vol. 46. 10. ACM. 2011, pp. 773–788.

[28] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. "Resource Management with Deep Reinforcement Learning". In: *Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNets)*. Atlanta, GA, 2016.

[29] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. "Neural Adaptive Video Streaming with Pensieve". In: *Proceedings of the 2017 conference on ACM SIGCOMM 2017 Conference*. ACM. 2017.

[30] Hongzi Mao, Shaileshh Bojja Venkatakrishnan, Malte Schwarzkopf, and Mohammad Alizadeh. "Variance Reduction for Reinforcement Learning in Input-Driven Environments". In: *arXiv preprint arXiv:1807.02264* (2018).

[31] Monaldo Mastrolilli and Ola Svensson. "(Acyclic) job shops are hard to approximate". In: *Foundations of Computer Science, 2008. FOCS'08. IEEE 49th Annual IEEE Symposium on*. IEEE. 2008, pp. 583–592.

[32] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V Le, and Jeff Dean. "A Hierarchical Model for Device Placement". In: (2018).

[33] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, et al. "Device Placement Optimization with Reinforcement Learning". In: *Proceedings of The 33rd International Conference on Machine Learning*. 2017.

[34] Andrew Y Ng and Michael Jordan. "PEGASUS: A policy search method for large MDPs and POMDPs". In: *Proceedings of the Sixteenth conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc. 2000, pp. 406–415.

[35] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. "Making Sense of Performance in Data Analytics Frameworks". In: *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Oakland, California, USA, May 2015, pp. 293–307.

[36] Xue Bin Peng, Glen Berseth, and Michiel van de Panne. "Terrain-Adaptive Locomotion Skills Using Deep Reinforcement Learning". In: *ACM Transactions on Graphics (Proc. SIGGRAPH 2016)* 35.4 (2016).

[37] Chandrasekharan Rajendran. "A no-wait flowshop scheduling heuristic to minimize makespan". In: *Journal of the Operational Research Society* 45.4 (1994), pp. 472–478.

[38] John Schulman, Sergey Levine, Philipp Moritz, Michael I Jordan, and Pieter Abbeel. "Trust region policy optimization". In: *CoRR, abs/1502.05477* (2015).

[39] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. “Omega: flexible, scalable schedulers for large compute clusters”. In: *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM. 2013, pp. 351–364.

[40] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, et al. “Mastering the game of Go with deep neural networks and tree search”. In: *Nature* 529 (2016), pp. 484–503.

[41] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, et al. “Mastering the game of go without human knowledge”. In: *Nature* 550.7676 (2017), p. 354.

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

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

[44] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. “TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters”. In: *Proceedings of the Eleventh European Conference on Computer Systems*. ACM. 2016, p. 35.

[45] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, et al. “Apache Hadoop YARN: Yet Another Resource Negotiator”. In: SOCC ’13. Santa Clara, California: ACM, 2013, 5:1–5:16.

[46] 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.

[47] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. “Large-scale cluster management at Google with Borg”. In: *Proceedings of the European Conference on Computer Systems (EuroSys)*. Bordeaux, France, 2015.

[48] Lex Weaver and Nigel Tao. “The optimal reward baseline for gradient-based reinforcement learning”. In: *Proceedings of the Seventeenth conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc. 2001, pp. 538–545.

[49] Ronald J Williams. “Simple statistical gradient-following algorithms for connectionist reinforcement learning”. In: *Machine learning* 8.3-4 (1992), pp. 229–256.

[50] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. “Spark: cluster computing with working sets.” In: *HotCloud* 10 (2010), pp. 10–10.

Appendices

A An illustrative example of dependency-aware job scheduling

Figure 16 shows a common example: a DAG with two branches that converge in a join stage. A simple critical path heuristic would choose to work on the right branch, which contains a larger aggregate work: 90 task-seconds vs. 10 in the left branch. Once the orange stage finishes, however, the final join stage cannot yet run, since its other parent stage (in green) is still incomplete. Completing it next, followed by the join stage — as a critical-path schedule would — results in an overall makespan of $28 + 3\epsilon$. The optimal schedule, by contrast, completes this DAG in $20 + 3\epsilon$ time, a 29% improvement. Intuitively, an ideal schedule allocates resources such that both branches reach the final join stage at the same time, and execute it without blocking.

B Implementation details of training

We represent the scheduling policy as a neural network (called policy network) which takes as input processed vectors from the graph embedding step (§5.1), and outputs a probability distribution over all possible actions (i.e., combination of stage and parallelism, §5.2). We train the policy network in an *episodic* setting. In each episode, a fixed number of jobs arrive and are scheduled based on the policy, as described in §3. The episode terminates when *all* jobs finish executing or a synchronized termination step arrives (§5.3).

To train a policy that generalizes well, we consider multiple examples of job arrival sequences during training. In each training iteration, we simulate N independent episodes for the *same* job sequence to explore the probabilistic space of possible actions using the current policy, and use the resulting rewards to improve the policy for all job sequences. Specifically, we record the state, action, and reward information for all timesteps of each episode, and use these values to compute the cumulative reward, R_t, at each timestep t of each episode. We then train the neural network using a variant of the REINFORCE algorithm described in §5.3 (more details in [28; 49, §2].

Recall that REINFORCE estimates the policy gradient using Equation (4). A drawback of vanilla REINFORCE is that the gradient estimates can have high variance. To reduce the variance, it is common to subtract a *baseline* value (b_t in equation (4)) from the returns, R_t. The baseline can be calculated in different ways. A simple approach that we adopt is to use the average of the return values, R_t, where the average is taken at the same *timestep* t across all episodes[7] with the same job sequence (a similar approach has been used in [28; 34; 38]). Algorithm 1 shows the pseudo-code for the training algorithm.

[7] Some episodes terminate earlier, thus we zero-pad them to make every episode equal-length L.

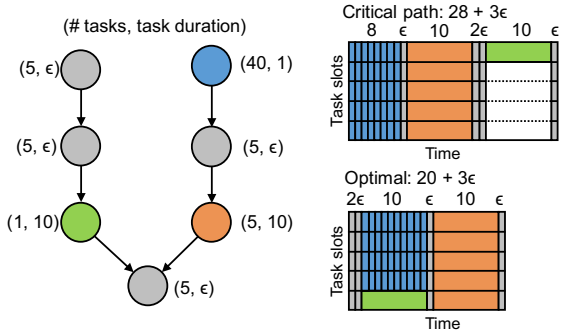
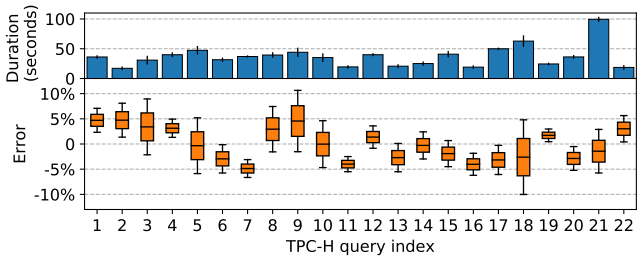


Figure 16: By planning ahead, the optimal, DAG-aware schedule avoids the "choke point" that occurs for the green node using the critical path heuristic, and improves completion time by 29%.

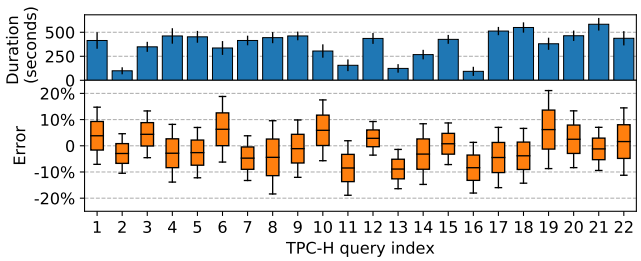
Algorithm 1 Policy gradient method for RL training.

for each iteration **do**
  $\Delta\theta \leftarrow 0$
  for each job sequence **do**
    run episode $i = 1, \ldots, N$:
    $\{s_1^i, a_1^i, r_1^i, \ldots, s_{L_i}^i, a_{L_i}^i, r_{L_i}^i\} \sim \pi_\theta$
    compute returns: $R_t^i = \sum_{s=t}^{L_i} r_s^i$
    for t = 1 to L **do**
      compute baseline: $b_t = \frac{1}{N} \sum_{i=1}^{N} R_t^i$
      for i = 1 to N **do**
        $\Delta\theta \leftarrow \Delta\theta + \alpha \nabla_\theta \log \pi_\theta(s_t^i, a_t^i)(R_t^i - b_t)$
      end for
    end for
  end for
end for

The hyperparameter setting of Decima is the following. The graph embedding transformation functions f and g in §5.1 are implemented with a two hidden-layer neural network, with 32 and 16 hidden units on each layer. The same network architectures are used across per-node, per-job and global embedding. During training, there are 16 parallel workers to compute rollouts for speedup. For streaming training (§5.3) specifically, the moving window for estimating $\hat{r}$ spans 10^5 time steps; the number of incoming jobs is capped at 500; and the episode termination probability decays linearly from 5×10^{-7} to 5×10^{-8} throughout training. We train Decima for at least 50,000 iterations for all experiments. Evaluations in §7 are all performed on unseen test job sequences (e.g., unseen TPC-H job combinations, unseen part of the company trace, etc.). Our simulator and training method (§5.3) are efficient. For example, every batch training (50 executors 20 jobs as in §7.2) iteration (including interaction in simulation, model inference and model update from all training workers) takes roughly 1.5 seconds on a machine with Intel Xeon E5-2640 CPU and Nvidia Tesla P100 GPU.



(a) Single job.



(b) Mixture of jobs.

Figure 17: Testing the fidelity of our Spark simulator. The scheduling agent is Decima. The error bar in real spark job duration (blue bars in upper figures) spans one standard deviation across 10 experiments. The span in simulation error (orange bars in lower figures) denotes 95% confidence interval. (a) Each of the 22 TPC-H jobs runs along in the cluster. The discrepancy between simulated and actual job duration is at most ±5%. (b) Running mixture of all 22 TPC-H jobs. The mean error is at most ±9%.

C Simulator fidelity

Figure 17 shows how simulated and real Spark differs in terms of job completion time for 10 runs of TPC-H job sets (§7.2). The results show that the simulator closes matches the actual run time of each job, even we run multiple jobs together in the clusters. Importantly, capturing all first-order effect in the Spark environment is crucial to achieve such accuracy (§6.2). For example, without modeling the executor moving delay, the simulated runtime incurs a consistent negative offset. Training in such environment would therefore result in a policy that moves executor more often than intended (§7.5). Also, omitting the waves and stretches artifacts significantly increase the variance in simulated runtime, making it unfaithful to reflect the real cluster. Only when all the factors are consider, can we achieve the small discrepancy between real and simulation as shown in Figure 17.

D Competing heuristics in multi-resource scheduling environment

When evaluating Decima's performance in multi-resource scheduling environment (§7.4), we compare with several heuristics. First, we consider the optimally tuned weighted fair heuristic from §7.2. This heuristic grants each job an executor quota based on the total work in the job. Then the heuristic chooses a stage the same way as in §7.2. Among the available executor types, the heuristic exhausts the best-fitting category before choosing any others. The scheduler ensures that the aggregate allocated resources (across different executor types) do not exceed the job's quota.

Second, we compare to the standard resource-packing algorithm from Tetris [18]. To maximize resource utilization, we select the DAG node that yields the largest dot product of requested resource vector and available resource vector for each executor type. Then we greedily grant as much parallelism as the tasks need in this node.

The two heuristics lack each other's key scheduling ingredients (fairness and packing), and neither understands the DAG structure. Finally, we compare to Graphene [19], whose hybrid heuristic combines these factors, Our multi-resource scheduling environment with discrete executor classes differs from the original Graphene setting, which assumes continuous, infinitely divisible resources. We adapted the Graphene algorithm for discrete executors, but kept its essence: specifically, we estimate and group the "troublesome" nodes the same way [19, §4.1]. To ensure that troublesome nodes are scheduled at the same time, we dynamically suppress the priority on all troublesome nodes of a DAG until *all* of these nodes are available in the frontier. We also include parallelism control by sharing the executors according to the weighted partition heuristic; and we pack resources by prioritizing the executor type that best fits the resource request. In the end, we perform a grid search on all the hyperparameters (e.g., threshold for picking troublesome nodes) in this heuristic for the best scheduling performance in each of the experiments in §7.4.

E Optimality of Decima

In §7, we show Decima is able to rival or outperform existing scheduling schemes in a wide range of complex cluster environments, including a real Spark testbed, real-world cluster trace simulations and a multi-resource packing environment. However, the optimality of Decima in those environments remains unknown due to the intractability of computing exact optimal scheduling solutions [19; 31], or tight lower bounds.[8] Nevertheless to understand Decima's optimality, we test Decima in simplified settings where a brute-force search over different scheduling configurations is possible as a near-optimal baseline for comparison.

We consider the Spark executor based scheduling framework simulated in §6.2 with average JCT objective for a batch of jobs. To simplify the environment, we turn off the "wave" effect, executor startup delays and the artifact of task slowdowns at high degrees of parallelism. As a result, the duration of a stage has a strict inverse relation to the number of executors the stage runs on; the scheduler is also free to move executors across jobs without any overhead. The dom-

[8] In our scheduling settings (e.g., Spark's executor based scheduling), we find lower bounds based on total work or critical path to be too loose to unveil meaningful information of how optimal Decima stands.

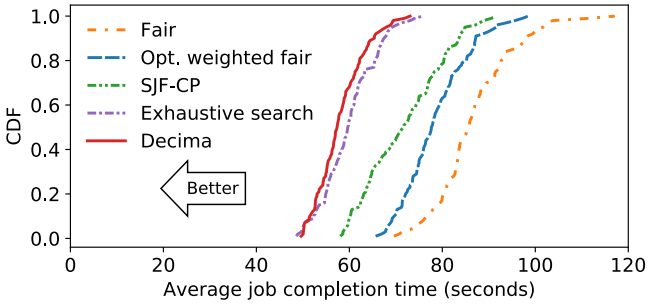


Figure 18: Comparing Decima with near optimal heuristics in a simplified scheduling environment.

inating challenge in this environment is to pack jobs tightly and favor short jobs first as much as possible.

To find a good schedule for a batch of n jobs, we consider *exhaustive search* over all $n!$ possible job orderings, and select the ordering having the lowest average JCT. To make the exhaustive search feasible, we reduce the number of batch jobs to 10. For a fixed job ordering, at each scheduling event (§5.2), we select the unfinished job appearing earliest in the order (we use critical path to choose the order of nodes within each job) to assign to available executors. By enumerating over all possible job orderings, the algorithm is guaranteed to find a schedule where jobs finish in order of their respective sizes thus resulting in small average JCT. We believe this policy to be close to optimal, as we have empirically observed job orderings to have a dominating effect on the average JCT in TPC-H workloads (§7.5).

Next, we train an *unmodified* Decima agent in this environment with the same batch setting as in §7.2. We compare the performance of Decima with our exhaustive search baseline, a shortest-job-first critical-path heuristic, and the optimally tuned weighted fair scheduler (described in §7.2). The results are shown in Figure 18.

We make three key observations. First, different from the results in real spark (Figure 9), the SJF-CP scheme outperforms the optimally tuned weighted fair scheduler. This is expected because SJF-CP strictly favors small jobs to optimize for the average JCT (any misassignment of executors off the shortest job would hurt the performance). Second, the exhaustive search heuristic performs better than SJF-CP scheme. This is due to SJF-CP not exploiting information about the DAG structure or the current cluster state—beyond just the critical path or total work—resulting in a suboptimal packing. Whereas by trying out different job orderings, the exhaustive search heuristic is able to find a schedule in which jobs are not only ordered correctly, but also pack well. Third, remarkably, Decima matches or achieves slightly better average JCT — on average, Decima reduces the JCT by 9%. We found that Decima is better at dynamically packing jobs based on their current structure at run time (e.g., how much work remains on each dependency path). It enables Decima to outperform the heuristic that strictly follows the order determined in a static exhaustive search. This experiment hallmarks Decima's ability to automatically search for a near optimal scheduling algorithm, by interacting with the cluster with only the raw observation of the job states.

Decima training scenario	average JCT (1k seconds)
Decima trained with 10 jobs	3.54 ± 0.45
Decima trained with 150 jobs	3.29 ± 0.68
Decima trained with 1k executors	0.63 ± 0.07
Decima trained with 10k executors	0.61 ± 0.09

Table 2: Decima generalizes well to deployment scenarios in which the workload or cluster differ from the training setting.

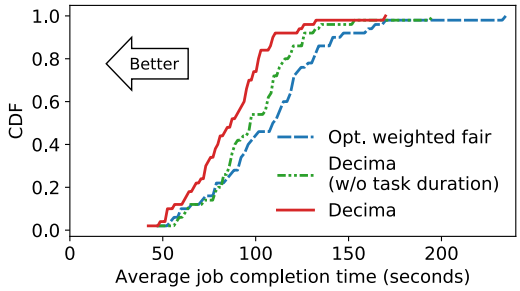


Figure 19: Decima performs worse on unseen jobs without task duration estimates, but still outperforms the best heuristic.

F Generalizing Decima to different deployment environments

Practical clusters often have verying workloads, and their available resources also change over time. Ideally, Decima would generalize from a model trained for a specific load and cluster size to similar workloads with different parameters. To test this, we train Decima's agent on a scaled-down version of the industrial workload, using with 15× fewer concurrent jobs and 10× fewer executors than in the test setting. Table 2 shows how the performance of this agent compares with that of one trained on the real workload and cluster size. Decima is robust to changing parameters: it generalizes to 15× more jobs with a 7% worse average JCT, and to a 10× larger cluster with a 3% worse average JCT. Generalization to a larger cluster is robust as the policy correctly limits jobs' parallelism even if vastly more resources are available. By contrast, generalizing to a workload with many more jobs is harder, as the smaller-scale training lacks experiences with complex job combinations.

G Decima with incomplete information

In a real cluster, Decima will occasionally encounter unseen jobs without reliable task duration estimates. Unlike heuristics that fundamentally rely on profiling information (e.g., weighted fair scheduling based on total work), Decima can still work with the remaining information and extract a reasonable scheduling policy. Running the same setting in §7.2, Figure 19 shows that training without task durations yields a policy that still outperforms the best heuristic, as Decima can still exploit the graph structure and other information such

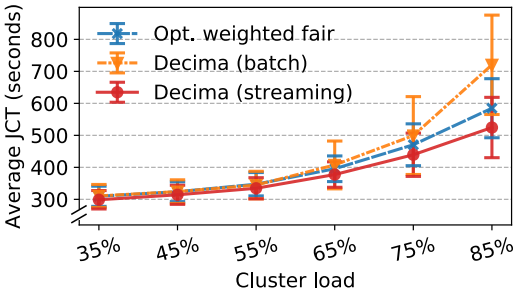


Figure 20: Decima trained with batch jobs does not trivially generalize to streaming. With streaming training techniques (§5.3) enabled, Decima is able to outperform the best comparing heuristics by at least 5–12%. However, Decima trained in batch setting performs worse than the heuristics, especially with large cluster load—e.g., 19% worse than the best heuristic in 85% cluster load.

as the correlation between number of tasks and the efficient parallelism point.

H Using Decima 's batch policy in streaming

Decima outperforms comparing heuristics in streaming settings (§7.2, §7.3, §7.4) as shown in Figures 10, 11b, 12. In §5.3 we have mentioned that new training techniques are required to achieve this—Decima's scheduling policies trained with batch job settings do not directly generalize to the streaming case. To illustrate this point, in Figure 20 we repeat the streaming experiments on the industrial workload (§7.3), and compare with a Decima policy trained on batches of jobs (with no arrival of new jobs beyond the initial batches). As shown, Decima trained on batch jobs consistently underperforms Decima trained directly over streaming scenarios; it even performs worse than the comparing heuristic. The difference is larger when the cluster load increases, which creates a harder scheduling problem. For example, batch Decima scheduling policy performs 19% worse than the best heuristic at 85% cluster load.