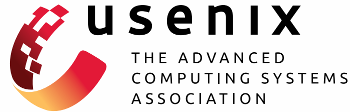


Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing

Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, and Jingren Zhou, *Microsoft;*
Zhengping Qian, Ming Wu, and Lidong Zhou, *Microsoft Research*

https://www.usenix.org/conference/osdi14/technical-sessions/presentation/boutin

This paper is included in the Proceedings of the
11th USENIX Symposium on
Operating Systems Design and Implementation.
October 6–8, 2014 • Broomfield, CO

978-1-931971-16-4

Open access to the Proceedings of the
11th USENIX Symposium on Operating Systems
Design and Implementation
is sponsored by USENIX.

Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing

Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou
Microsoft

Zhengping Qian, Ming Wu, Lidong Zhou
Microsoft Research

Abstract

Efficiently scheduling data-parallel computation jobs over cloud-scale computing clusters is critical for job performance, system throughput, and resource utilization. It is becoming even more challenging with growing cluster sizes and more complex workloads with diverse characteristics. This paper presents Apollo, a highly scalable and coordinated scheduling framework, which has been deployed on production clusters at Microsoft to schedule thousands of computations with millions of tasks efficiently and effectively on tens of thousands of machines daily. The framework performs scheduling decisions in a distributed manner, utilizing global cluster information via a loosely coordinated mechanism. Each scheduling decision considers future resource availability and optimizes various performance and system factors together in a single unified model. Apollo is robust, with means to cope with unexpected system dynamics, and can take advantage of idle system resources gracefully while supplying guaranteed resources when needed.

1 Introduction

MapReduce-like systems [7, 15] make data-parallel computations easy to program and allow running jobs that process terabytes of data on large clusters of commodity hardware. Each data-processing job consists of a number of tasks with inter-task dependencies that describe execution order. A task is a basic unit of computation that is scheduled to execute on a server.

Efficient scheduling, which tracks task dependencies and assigns tasks to servers for execution when ready, is critical to the overall system performance and service quality. The growing popularity and diversity of data-parallel computation makes scheduling increasingly challenging. For example, the production clusters that we use for data-parallel computations are growing in size, each with over 20,000 servers. A growing community of thousands of users from many different organizations submit jobs to the clusters every day, resulting in a peak rate of tens of thousands of scheduling requests per second. The submitted jobs are diverse in nature, with a variety of characteristics in terms of data volume to process, complexity of computation logic, degree of parallelism, and resource requirements. A scheduler must (i) scale to make tens of thousands of scheduling decisions per second on a cluster with tens of thousands of servers; (ii) maintain fair sharing of resources among different users and groups; and (iii) make high-quality scheduling decisions that take into account factors such as data locality, job characteristics, and server load, to minimize job latencies while utilizing the resources in a cluster fully.

This paper presents the Apollo scheduling framework, which has been fully deployed to schedule jobs in cloud-scale production clusters at Microsoft, serving a variety of on-line services. Scheduling billions of tasks daily efficiently and effectively, Apollo addresses the scheduling challenges in large-scale clusters with the following technical contributions.

- To balance scalability and scheduling quality, Apollo adopts a *distributed* and (loosely) *coordinated* scheduling framework, in which independent scheduling decisions are made in an optimistic and coordinated manner by incorporating synchronized cluster utilization information. Such a design strikes the right balance: it avoids the suboptimal (and often conflicting) decisions by independent schedulers of a completely decentralized architecture, while removing the scalability bottleneck and single point of failure of a centralized design.

- To achieve high-quality scheduling decisions, Apollo schedules each task on a server that minimizes the task completion time. The estimation model incorporates a variety of factors and allows a scheduler to perform a weighted decision, rather than solely considering data locality or server load. The data parallel nature of computation al-

lows Apollo to refine the estimates of task execution time continuously based on observed runtime statistics from similar tasks during job execution.

- To supply individual schedulers with cluster information, Apollo introduces a lightweight hardware-independent mechanism to advertise load on servers. When combined with a local task queue on each server, the mechanism provides a near-future view of resource availability on all the servers, which is used by the schedulers in decision making.

- To cope with unexpected cluster dynamics, suboptimal estimations, and other abnormal runtime behaviors, which are facts of life in large-scale clusters, Apollo is made robust through a series of *correction mechanisms* that dynamically adjust and rectify suboptimal decisions at runtime. We present a unique *deferred correction mechanism* that allows resolving conflicts between independent schedulers only if they have a significant impact, and show that such an approach works well in practice.

- To drive high cluster utilization while maintaining low job latencies, Apollo introduces *opportunistic scheduling*, which effectively creates two classes of tasks: *regular tasks* and *opportunistic tasks*. Apollo ensures low latency for regular tasks, while using the opportunistic tasks for high utilization to fill in the slack left by regular tasks. Apollo further uses a *token* based mechanism to manage capacity and to avoid overloading the system by limiting the total number of regular tasks.

- To ensure no service disruption or performance regression when we roll out Apollo to replace a previous scheduler deployed in production, we designed Apollo to support *staged rollout* to production clusters and *validation at scale*. Those constraints have received little attention in research, but are nevertheless crucial in practice and we share our experiences in achieving those demanding goals.

We observe that Apollo schedules over 20,000 tasks per second in a production cluster with over 20,000 machines. It also delivers high scheduling quality, with 95% of regular tasks experiencing a queuing delay of under 1 second, while achieving consistently high (over 80%) and balanced CPU utilization across the cluster.

The rest of the paper is organized as follows. Section 2 presents a high-level overview of our distributed computing infrastructure and the query workload that Apollo supports. Section 3 presents an architectural overview, explains the coordinated scheduling in detail, and describes the correction mechanisms. We describe our engineering experiences in developing and deploying Apollo to our cloud infrastructure in Section 4. A thorough evaluation is presented in Section 5. We review related work in Section 6 and conclude in Section 7.

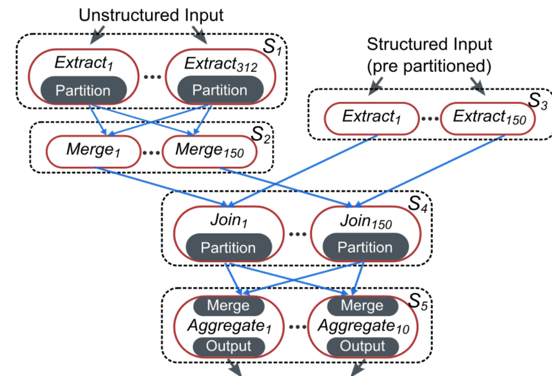


Figure 1: A sample SCOPE execution graph.

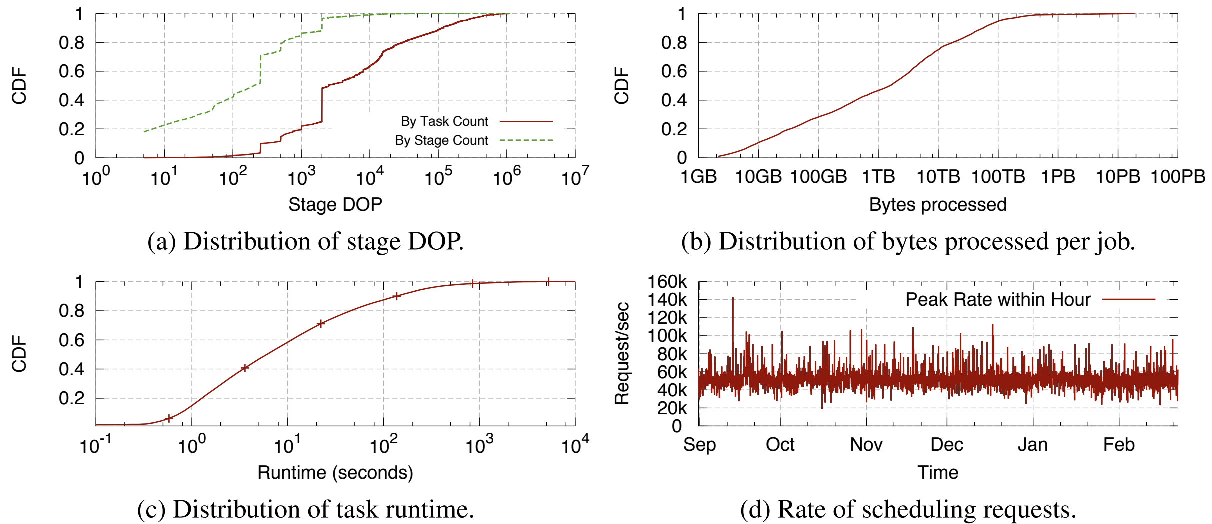
2 Scheduling at Production Scale

Apollo serves as the underlying scheduling framework for Microsoft's distributed computation platform, which supports large-scale data analysis for a variety of business needs. A typical cluster contains tens of thousands of commodity servers, interconnected by an oversubscribed network. A distributed file system stores data in *partitions* that are distributed and replicated, similar to GFS [12] and HDFS [3]. All computation jobs are written using SCOPE [32], a SQL-like high-level scripting language, augmented with user-defined processing logic. The optimizer transforms a job into a physical execution plan represented as a directed acyclic graph (DAG), with tasks, each representing a basic computation unit, as vertices and the data flows between tasks as edges. Tasks that perform the same computation on different partitions of the same inputs are logically grouped together in *stages*. The number of tasks per stage indicates the degree of parallelism (DOP).

Figure 1 shows a sample execution graph in SCOPE, greatly simplified from an important production job that collects user click information and derives insights for advertisement effectiveness. Conceptually, the job performs a join between an unstructured user log and a structured input that is pre-partitioned by the join key. The plan first partitions the unstructured input using the partitioning scheme from the other input: stages S_1 and S_2 respectively partition the data and aggregate each partition. A partitioned join is then performed in stage S_4. The DOP is set to 312 for S_1 based on the input data volume, set to 10 for S_5, and set to 150 for S_2, S_3, and S_4.

2.1 Capacity Management and Tokens

In order to ensure fairness and predictability of performance, the system uses a *token-based* mechanism to allocate capacity to jobs. Each token is defined as the right



(a) Distribution of stage DOP.

(b) Distribution of bytes processed per job.

(c) Distribution of task runtime.

(d) Rate of scheduling requests.

Figure 2: Heterogeneous workload.

to execute a regular task, consuming up to a predefined amount of CPU and memory, on a machine in the cluster. For example, if a job has an allocation of 100 tokens, this means it can run 100 tasks, each of which consumes up to a predefined maximum amount of CPU and memory.

A *virtual cluster* is created for each user group for security and resource sharing reasons. Each virtual cluster is assigned a certain amount of capacity in terms of number of tokens, and maintains a queue of all submitted jobs. A job submission contains the target virtual cluster, the necessary credentials, and the required number of tokens for execution. Virtual cluster management utilizes various admission control policies and decides how and when to assign its allocated tokens to submitted jobs. Jobs that do not get their required tokens will be queued in the virtual cluster. The system also supports a wide range of capabilities, such as job priorities, suspension, upgrades, and cancellations.

Once a job starts to execute with required tokens, it is a scheduler's responsibility to execute its optimized execution plan by assigning tasks to servers while respecting token allocation, enforcing task dependencies, and providing fault tolerance.

2.2 The Essence of Job Scheduling

Scheduling a job involves the following responsibilities: (i) *ready list*: maintain a list of tasks that are ready to be scheduled: initially, the list includes those leaf tasks that operate on the original inputs (e.g., tasks in stages S_1 and S_3 in Figure 1); (ii) *task priority:* sort the ready list appropriately; (iii) *capacity management:* manage the capacity assigned to the job and decide when to schedule a task based on the capacity management policy; (iv) *task scheduling:* decide where to schedule a task and dispatch it to the selected server; (v) *failure recovery:* monitor scheduled tasks, initiate recovery actions when tasks fail, and mark the job failed if recovery is not possible; (vi) *task completion:* when a task completes, check its dependent tasks in the execution graph and move them to the *ready list* if all the tasks that they depend on have completed; (vii) *job completion:* repeat the whole process until all tasks in the job are completed.

2.3 Production Workload Characteristics

The characteristics of our target production workloads greatly influence the Apollo design. Our computation clusters run more than 100,000 jobs on a daily basis. At any point in time, there are hundreds of jobs running concurrently. Those jobs vary drastically in almost every dimension, to meet a wide range of business scenarios and requirements. For example, large jobs process terabytes to petabytes of data, contain sophisticated business logic with a few dozen complex joins, aggregations, and user-defined functions, have hundreds of stages, contain over a million tasks in the execution plan, and may take hours to finish. On the other hand, small jobs process gigabytes of data and can finish in seconds. In SCOPE, different jobs are also assigned with different amounts of resources. The workload evolves constantly as the supporting business changes over time. This workload diversity poses tremendous challenges for the underlying scheduling framework to deal with efficiently and effectively. We describe several job characteristics in our production environment to illustrate the diverse and dynamic nature of the computation workload.

In SCOPE, the DOP for a stage in a job is chosen based on the amount of data to process and the complexity of each computation. Even within a single job, the DOP changes for different stages as the data volume changes over the job's lifetime. Figure 2(a) shows the distribution

of stage DOP in our production environment. It varies from a single digit to a few tens of thousands. Almost 40% of stages have a DOP of less than 100, accounting for less than 2% of the total workload. More than 98% of tasks are part of stages with DOP of more than 100. These large stage sizes allow a scheduler to draw statistics from some tasks to infer behavior of other tasks in the same stage, which Apollo leverages to make informed and better decisions. Job sizes vary widely from a single vertex to millions of vertices per job graph. As illustrated in Figure 2(b), the amount of data processed per job ranges from gigabytes to tens of petabytes. Task execution times range from less than 100ms to a few hours, as shown in Figure 2(c). 50% of tasks run for less than 10 seconds and are sensitive to scheduling latency. Some tasks require external files such as executables, configurations, and lookup tables for their execution, thus incurring initialization costs. In some cases, such external files required for execution are bigger than the actual input to be processed, which means locality should be based on where those files are cached rather than input location. Collectively, such a large number of jobs create a high scheduling-request rate, with peaks above 100,000 requests per second, as shown in Figure 2(d).

The very dynamic and diverse characteristics of our computing workloads and cluster environments impose several challenges for the scheduling framework, including scalability, efficiency, robustness, and resource usage balance. The Apollo scheduling framework has been designed and shown to address these challenges over large production clusters at Microsoft.

3 The Apollo Framework

To support the scale and scheduling rate required for the production workload, Apollo adopts a *distributed* and *coordinated* architecture, where the scheduling of each job is performed independently and incorporates aggregated global cluster load information.

3.1 Architectural Overview

Figure 3 provides an overview of Apollo's architecture. A *Job Manager* (JM), also called a scheduler, is assigned to manage the life cycle of each job. The global cluster load information used by each JM is provided through the cooperation of two additional entities in the Apollo framework: a *Resource Monitor* (RM) for each cluster and a *Process Node* (PN) on each server. A PN process running on each server is responsible for managing the local resources on that server and performing local scheduling, while the RM aggregates load information from PNs across the cluster continuously, providing a global view of the cluster status for each JM to make informed scheduling decisions.

While treated as a single logical entity, the RM can

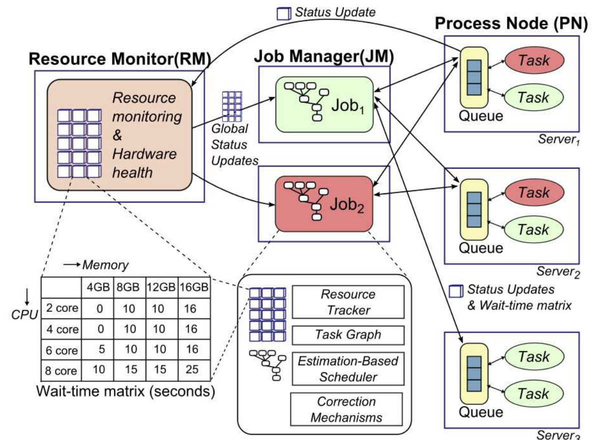


Figure 3: Apollo architectural overview.

be implemented physically in different configurations with different mechanisms, as it essentially addresses the well-studied problem of monitoring dynamically changing state of a collection of distributed resources at a large scale. For example, it can use a tree hierarchy [20] or a directory service with an eventually consistent gossip protocol [8, 26]. Apollo's architecture can accommodate any of such configurations. We implemented the RM in a master-slave configuration using Paxos [18]. The RM is never on the performance critical path: Apollo can continue to make scheduling decisions (at a degraded quality) even when the RM is temporarily unavailable, for example, during a transient master-slave switch due to a machine failure. In addition, once a task is scheduled to a PN, the JM obtains up-to-date load information directly from the PN via frequent status updates.

To better predict resource utilization in the near future and to optimize scheduling quality, each PN maintains a local queue of tasks assigned to the server and advertises its future resource availability in the form of a *wait-time matrix* inferred from the queue (Section 3.2). Apollo thereby adopts an estimation-based approach to making task scheduling decisions. Specifically, Apollo considers the wait-time matrices, aggregated by the RM, together with the individual characteristics of tasks to be scheduled, such as the location of inputs (Section 3.3). However, cluster dynamics pose many challenges in practice; for example, the wait-time matrices might be stale, estimates might be suboptimal, and the cluster environment might sometimes be unpredictable. Apollo therefore incorporates correction mechanisms for robustness and dynamically adjusts scheduling decisions at runtime (Section 3.4). Finally, there is an inherent tension between providing guaranteed resources to jobs (e.g., to ensure SLAs) and achieving high cluster utilization, because both the load on a cluster and the resource needs of a job fluctuate constantly. Apollo resolves this tension through *opportunistic scheduling*, which creates second-class tasks to use idle resources (Section 3.5).

3.2 PN Queue and Wait-Time Matrix

The PN on each server manages a *queue* of tasks assigned to the server in order to provide projections on future resource availability. When a JM schedules a task on a server, it sends a task-creation request with (i) fine grained resource requirement (CPU cores and memory), (ii) estimated runtime, and (iii) a list of files required to run the task (e.g., executables and configuration files). Once a task creation request is received, the PN copies the required files to a local directory using a peer-to-peer data transfer framework combined with a local cache. The PN monitors CPU and memory usage, considers the resource requirements of tasks in the queue, and executes them when the capacity is available. It maximizes resource utilization by executing as many tasks as possible, subject to the CPU and memory requirements of individual tasks. The PN queue is mostly FIFO, but can be reordered. For example, a later task requiring a smaller amount of resources can fill a gap without affecting the expected start time of others.

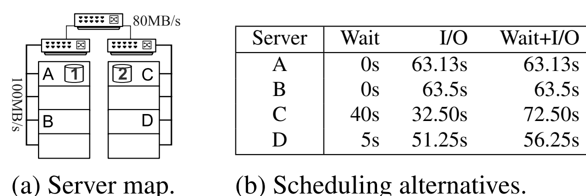
The use of task queues enables schedulers to dispatch tasks to the PNs proactively based on future resource availability, instead of based on instantaneous availability. As illustrated later in Section 3.3, Apollo considers task wait time (for sufficient resources to be available) and other task characteristics holistically to optimize task scheduling. The use of task queues also masks task initialization cost by copying the files before execution capacity is available, thereby avoiding idle gaps between tasks. Such a direct-dispatch mechanism provides the efficiency needed particularly by small tasks, for which any protocol to negotiate incurs significant overhead.

The PN also provides feedback to the JM to help improve accuracy of task runtime estimation. Initially, the JM uses conservative estimates provided by the query optimizer [32] based on the operators in a task and the amount of data to be processed. Tasks in the same stage perform the same computation over different datasets. Their runtime characteristics are similar and the statistics from the executions of the earlier tasks can help improve runtime estimates for the later ones. Once a task starts running, the PN monitors its overall resource usage and responds to the corresponding JM's status update requests with information such as memory usage, CPU time, execution time (wall clock time), and I/O throughput. The JM then uses this information along with other factors such as operator characteristics and input size to refine resource usage and predict expected runtime for tasks from the same stage.

The PN further exposes the load on the current server to be aggregated by its RM. Its representation of the load information should ideally convey a projection of the future resource availability, mask the heterogeneity of servers in our data centers (e.g., servers with 64GB of memory and 128GB of memory have different capacities), and be concise enough to allow frequent updates. Apollo's solution is a *wait-time matrix*, with each cell corresponding to the expected wait time for a task that requires a certain amount of CPU and memory. Figure 3 contains a matrix example: the value 10 in cell ⟨12 GB, 4 cores⟩ denotes that a task that needs 4 CPU cores and 12GB of memory has to wait 10 seconds in this PN before it can get its resource quota to execute. The PN maintains a matrix of expected wait times for any hypothetical future task with various resource quotas, based on the currently running and queued tasks. The algorithm simulates local task execution and evaluates how long a future task with a given CPU/memory requirement would wait on this PN to be executed. The PN updates this matrix frequently by considering the actual resource situation and the latest task runtime and resource estimates. Finally, the PN sends this matrix, along with a timestamp, to every JM that has running or queued tasks in this PN. It also sends the matrix to the RM using a heartbeat mechanism.

3.3 Estimation-Based Scheduling

A JM has to decide which server to schedule a particular task to using the wait-time matrices in the aggregated view provided by the RM and the individual characteristics of the task to be scheduled. Apollo has to consider a variety of (often conflicting) factors that affect the quality of scheduling decisions and does so in a single unified model using an estimation-based approach.



Server	Wait	I/O	Wait+I/O
A	0s	63.13s	63.13s
B	0s	63.5s	63.5s
C	40s	32.50s	72.50s
D	5s	51.25s	56.25s

(a) Server map. (b) Scheduling alternatives.

Figure 4: A task scheduling example.

We use an example to illustrate the importance of considering various factors all together, as well as the benefit of having a local queue on each server. Figure 4(a) shows a simplified server map with two racks, each with four servers, connected via a hierarchically structured network. Assume data can be read from local disks at 160MB/s, from within the same rack at 100MB/s, and from a different rack at 80MB/s. Consider scheduling a task with two inputs (one 100MB stored on server A and the other 5GB stored on server C) whose runtime is dominated by I/O. Figure 4(b) shows the four scheduling choices, where servers A and B are immediately available, while server C has the best data locality. Yet, D is the optimal choice among those four choices. This can be recognized only when we consider data locality

and wait time together. This example also illustrates the value of local queues: without a local queue on each server, any scheduling mechanism that checks for immediate resource availability would settle on the non-optimal choice of server A or B.

Apollo therefore considers various factors holistically and performs scheduling by estimating task completion time. First, we estimate the task completion time if there is no failure, denoted by E_{succ}, using the formula

$$E_{succ} = I + W + R \tag{1}$$

I denotes the initialization time for fetching the needed files for the task, which could be 0 if those files are cached locally. The expected wait time, denoted as W, comes from a lookup in the wait-time matrix of the target server with the task resource requirement. The task runtime, denoted as R, consists of both I/O time and CPU time. The I/O time is computed as the input size divided by the expected I/O throughput. The I/O could be from local memory, disks, or network at various bandwidths. Overall, estimation of R initially incorporates information from the optimizer and is refined with runtime statistics from the tasks in the same stage.

Second, we consider the probability of task failure to calculate the final completion time estimate, denoted by C. Hardware failures, maintenance, repairs, and software deployments are inevitable in a real large-scale environment. To mitigate their impact, the RM also gathers information on upcoming and past maintenance scheduled on every server. Together, a success probability P_{succ} is derived and considered to calculate C, as shown below. A penalty constant K_{fail}, determined empirically, is used to model the cost of server failure on the completion time.

$$C = P_{succ} E_{succ} + K_{fail}(1 - P_{succ}) E_{succ} \tag{2}$$

Task Priorities. Besides completion time estimation, the task-execution order also matters for overall job latency. For example, for the job graph in Figure 1, the tasks in S_1 run for 1 minute on average, the tasks in S_2 run for an average of 2 minutes, with potential partition-skew induced stragglers running up to 10 minutes, and the tasks in S_3 run for 30 seconds on average. As a result, efficiently executing S_1 and S_2 surely appears more critical to achieve the fastest runtime. Therefore, the scheduler should prioritize resources to S_1 and S_2 before considering S_3. Within S_2, the scheduler should start the vertex with the largest input as early as possible, because it is the most likely to be on the critical path of the job.

A static task priority is annotated per stage by the optimizer through analyzing the job DAG and calculating the potential critical path of the job execution. Tasks within a stage are prioritized based on the input size. Apollo schedules tasks and allocates their resources in a

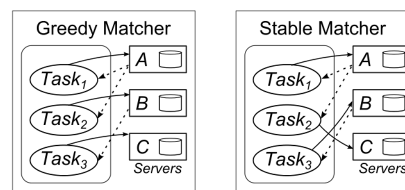


Figure 5: A matching example.

descending order of their priorities. Since a job contains a finite number of tasks, the starvation of a task with low static priority is impossible, because eventually it will be the only task left to execute, and will be executed.

Stable Matching. For efficiency, Apollo schedules tasks with similar priorities in *batches* and turns the problem of task scheduling into that of *matching* between tasks and servers. For each task, we could search all the servers in a cluster for the best match. The approach becomes prohibitively expensive on a large cluster. Instead, Apollo limits the search space for a task to a *candidate set* of servers, including (i) a set of servers on which inputs of significant sizes are located (ii) a set of servers in the same rack as those from the first group (iii) two servers randomly picked from a set of lightly-loaded servers; the list is curated in the background.

A greedy algorithm can be applied for each task *sequentially*, choosing the server with the earliest estimated completion time at each step. However, the outcome of the greedy algorithm is sensitive to the order in which tasks are matched and often leads to suboptimal decisions. Figure 5 shows an example with a batch of three tasks being scheduled. Assume both $Task_1$ and $Task_2$ read data from server A while $Task_3$ reads from server B, as shown with dotted lines. Each server has capacity to start one task. The greedy matcher first matches $Task_1$ to server A, then matches $Task_2$ to server B because $Task_1$ is already scheduled on A, and finally $Task_3$ to server C, as shown with solid lines. A better match would have assigned $Task_3$ to server B for better locality.

Apollo therefore adopts a variant of the stable matching algorithm [10] to match tasks with servers. For each task in a batch, Apollo finds the server with the earliest estimated completion time as a *proposal* for that task. A server accepts a proposal from a task if that is the only proposal assigned. A conflict arises when more than one task proposes to the same server. In this case, the server picks the task whose completion time saving is the greatest if it is assigned to the server. The tasks not picked withdraw their proposals and enter the next iteration that tries to match the remaining tasks and servers. The algorithm iterates until all tasks have been assigned, or until it reaches the maximum number of iterations. As shown in Figure 5, the stable matcher matches $Task_2$ to C and $Task_3$ to B, which effectively leverages locality and results in better job performance.

The scheduler then sorts all the matched pairs based on their *quality* to decide the dispatch order. A match is considered with a higher quality if its task has a lower server wait time. The scheduler iterates over the sorted matches and dispatches in order until it is out of the allocated capacity. If opportunistic scheduling (Section 3.5) is enabled, the scheduler continues to dispatch the tasks until the opportunistic scheduling limit.

To simplify the matching algorithm for a tradeoff between efficiency and quality, Apollo assigns only one task to each server in a single batch, because otherwise Apollo has to update the wait-time matrix for a server to take into account the newly assigned task, which increases the complexity of the algorithm. This simplification might lead to a suboptimal match for a task in a case where servers taking on a task in the same batch already remains a better choice. Apollo mitigates the effect in two ways: if the suboptimal match is of a low quality, sorting the matches by quality will cause the dispatching of this task to be postponed, and later re-evaluated. Even if the suboptimal match is dispatched, the correction mechanisms described in Section 3.4 are designed to catch this case and reschedule the task if needed.

3.4 Correction Mechanisms

In Apollo, each JM can schedule tasks independently at a high frequency, with no delay in the process. This is critical for scheduling a large number of small tasks in the workload. However, due to the distributed nature of the scheduling, several JMs might make competing decisions at the same time. In addition, the information used, such as wait-time matrices, for scheduling decisions might be stale; the task wait time and runtime might be under or overestimated. Apollo has built-in mechanisms to address those challenges and dynamically adjust scheduling decisions with new information.

Unlike previous proposals (e.g., as in Omega [23]) in which conflicts are immediately handled at the *scheduling time*, Apollo optimistically *defers* any correction until after tasks are dispatched to PN queues. This design choice is based on our observation that conflicts are not always harmful. Two tasks scheduled to the same server simultaneously by different job managers might be able to run concurrently if there are sufficient resources for both; tasks that are previously scheduled on the server might complete soon, releasing the resources early enough to make any conflict resolution unnecessary. In those cases, a deferred correction mechanism, made possible with local queues, avoids the unnecessary overhead associated with eager detection and resolution. Correction mechanisms continuously re-evaluate the scheduling decisions with up-to-date information and make appropriate adjustments whenever necessary.

Duplicate Scheduling. When a JM gets fresh information from a PN during task creation, task upgrade, or while monitoring its queued tasks, it compares the information from this PN (and the elapsed wait time so far) to the information that was used to make the scheduling decision. The scheduler re-evaluates the decision if (i) the updated expected wait time is significantly higher than the original; (ii) the expected wait time is greater than the average among the tasks in the same stage; (iii) the elapsed wait time is already greater than the average. The first condition indicates an underestimated task completion time on the server, while the second/third conditions indicate a low matching quality. Any change in the decision triggers scheduling a duplicate task to a new desired server. Duplicates are discarded when one task starts.

Randomization. Multiple JMs might schedule tasks to the same lightly loaded PN, not aware of each other, thereby leading to scheduling *conflicts*. Apollo adds a small random number to each completion time estimation. This random factor helps reduce the chances of conflicts by having different JMs choose different, almost equally desirable, servers. The number is typically proportional to the communication interval between the JM and the PN, introducing no noticeable impact on the quality of the scheduling decisions.

Confidence. The aggregated cluster information obtained from the RM contains wait-time matrices of different ages, some of which can be stale. The scheduler attributes a lower confidence to older wait-time matrices because it is likely that the wait time changed since the time the matrix was calculated. When the confidence in the wait-time matrix is low, the scheduler will produce a pessimistic estimate by looking up the wait time of a task consuming more CPU and memory.

Straggler Detection. Stragglers are tasks making progress at a slower rate than other tasks, and have a crippling impact on job performances [4]. Apollo's straggler detection mechanism monitors the rate at which data is processed and the rate at which CPU is consumed to predict the amount of time remaining for each task. Other tasks in the same stage are used as a baseline for comparison. When the time it would take to rerun a task is significantly less than the time it would take to let it complete, a duplicate copy is started. They will execute in parallel until the first one finishes, or until the duplicate copy caught up with the original task. The scheduler also monitors the rate of I/O and detects stragglers caused by slow intermediate inputs. When a task is slow because of abnormal I/O latencies, it can rerun a copy of the upstream task to provide an alternate I/O path.

3.5 Opportunistic Scheduling

Besides achieving high quality scheduling at scale, Apollo is also designed to operate efficiently and drive high cluster utilization. Cluster utilization fluctuates over

time for several reasons. First, not all users submit jobs at the same time to consume their allocated capacities fully. A typical example is that the cluster load on weekdays is always higher than on weekends. Second, jobs differ in their resource requirements. Even daily jobs with the same computation logic consume different amount of resources as their input data sizes vary. Finally, a complete job typically goes through multiple stages, with different levels of parallelism and varied resource requirements. Such load fluctuation on the system provides schedulers with an opportunity to improve job performance by increasing utilization, at the cost of predictability. How to judiciously utilize occasionally idle computation resources *without* affecting SLAs remains challenging.

We introduce *opportunistic scheduling* in Apollo to gracefully take advantage of idle resources whenever they are available. Tasks can execute either in the *regular* mode, with sufficient tokens to cover its resource consumption, or in the *opportunistic* mode, without allocated resources. Each scheduler first applies optimistic scheduling to dispatch regular tasks with its allocated tokens. If all the tokens are utilized and there are still pending tasks to be scheduled, opportunistic scheduling may be applied to dispatch opportunistic tasks. Performance degradation of regular task is prevented by running opportunistic tasks at a lower priority at each server, and any opportunistic task can be preempted or terminated if the server is under resource pressure.

One immediate challenge is to prevent one job from consuming all the idle resources unfairly. Apollo uses *randomized allocation* to achieve probabilistic resource fairness for opportunistic tasks. In addition, Apollo upgrades opportunistic tasks to regular ones when tokens become available and assigned.

Randomized Allocation Mechanism. Ideally, the opportunistic resources should be shared fairly among jobs, proportionally to jobs' token allocation. This is particularly challenging as both the overall cluster load and individual server load fluctuate over time, which makes it difficult, if not impossible, to guarantee absolute instantaneous fairness. Instead, we focus on avoiding the worst case of a few jobs consuming all the available capacity of the cluster and target average fairness.

Apollo achieves this by setting a maximum opportunistic allowance for a given job proportionally to its token allocation. For example, a job with n tokens can have up to cn opportunistic tasks dispatched for some constant c. When a PN has spare capacity and the regular queue is empty, the PN picks a *random* task to execute from the opportunistic-task queue, regardless of when it was dispatched. If the chosen task requires more resources than what is available, the randomized selection process continues until there is no more task that can execute. Compared to a FIFO queue, the algorithm has the benefit of allowing jobs that start later to get a share of the capacity quickly. If a FIFO queue were used for opportunistic tasks, it could take an arbitrary amount of time for a later task to make its way through the queue, offering unfair advantages to tasks that start earlier.

As the degree of parallelism for a job varies in its lifetime, the number of tasks that are ready to be scheduled also varies. As a result, a job may not always be able to dispatch enough opportunistic tasks to use its opportunistic allowance fully. We further enhance the system by allowing each scheduler to increase the weight of an opportunistic task during random selection, to compensate for the reduction in the number of tasks. For example, a weight of 2 means a task has twice the probability to be picked. The total weight of all opportunistic tasks issued by the job must not exceed its opportunistic allowance.

Under an ideal workload, in which tasks run for the same amount of time and consume the same amount of resources, and in a perfectly balanced cluster, this strategy averages to sharing the opportunistic resources proportionally to the job allocation. However, in reality, tasks have large variations in runtime and resource requirements. The number of tasks dispatched per jobs change constantly as tasks complete and new tasks become ready. Further, jobs may not have enough parallelism at all times to use their opportunistic allowance fully. Designing a fully *decentralized* mechanism that maintains a strong fairness guarantee in a dynamic environment remains a challenging topic for future work.

Task Upgrade. Opportunistic tasks are subject to starvation if the host server experiences resource pressure. Further, the opportunistic tasks can wait for an unbounded amount of time in the queue. In order to avoid job starvation, tasks scheduled opportunistically can be upgraded to regular tasks after being assigned a token. Because a job requires at least one token to run and there is a finite amount of tasks in a job, the scheduler is able to transition a starving opportunistic task to a regular task at one point, thus preventing job starvation.

After an opportunistic task is dispatched, the scheduler tracks the task in its ready list until it completes. When scheduling a regular task, the scheduler considers both unscheduled tasks and previously scheduled opportunistic tasks that still wait for execution. Each scheduler allocates its tokens to tasks and performs task matches in a descending order of their priorities. It is not required that an opportunistic task be upgraded on the same machine, but it might be preferable as there is no initialization time. By calculating all costs holistically, the scheduler favors upgrading opportunistic tasks on machines with fewer regular tasks, while waiting for temporarily heavily loaded machines to drain. This strategy results in a better utilization of the tokens and better load balancing.

4 Engineering Experiences

Before the development of Apollo, we already had a production system running at full scale with our previous generation DAG scheduler, also referred to as the baseline scheduler. The baseline scheduler schedules jobs without any global cluster load information. When tasks wait in a PN queue, the baseline scheduler systematically triggers duplicates to explore other idle PNs and balance the load. Locality is modeled as a per-task scheduling constraint that is relaxed over time, instead of using completion time estimation.

In our first attempt to improve the baseline scheduler, we took an approach of a centralized global scheduler, which is responsible for scheduling all jobs in a cluster. While the global scheduler theoretically oversees all activities in the cluster and could make optimal scheduling decisions, we found that it became a performance bottleneck and its scheduling quality degraded as the numbers of machines in the cluster and concurrent jobs continued to grow. In addition, as described in Section 2, our heterogeneous workload consists of jobs with diverse characteristics. Any scheduling delay could have a direct and severe impact on small tasks. Such lessons ultimately led us to choose Apollo's distributed and loosely coordinated scheduling paradigm.

During the development of Apollo, we had to ensure the availability of our production system throughout the process, from early prototyping and experimentation to evaluation and eventual full deployment. This has posed many interesting challenges in validation, migration, deployment, and operation at full production scale.

Validation at Scale. We started by evaluating Apollo in an isolated test cluster at a smaller scale. It helps us verify scheduling decisions at every step and track down performance issues quickly. However, the approach has limitations as the scale of the test cluster is rather limited and cannot emulate the dynamic cluster environment. Many interesting challenges arise as the scale and complexity of the workload grows. For example, Apollo and the baseline scheduler make different assumptions around the capacity of the machines. In a test cluster with a single job running at a time, the baseline scheduler schedules a single task to a server, resulting in much lower machine utilization compared to Apollo. However, this improvement does not translate into gain in production environments because the utilization in production clusters is already high. Therefore, it is important for us to evaluate Apollo in real production clusters, side by side with busy production workloads.

Another lesson we learned from our first failed attempt was to validate design and performance continuously, instead of delaying full validation until completion and exposing scalability and performance issues when it is too late. This is particularly important as each engineering attempt is significant and time-consuming at this large scale.

Apollo's fully decentralized design allows each scheduler to run side by side with other schedulers, or other versions of Apollo itself. Such engineering agility is critical and allows us to compare performance across different schedulers at scale in the same production environments. We sampled production jobs and reran them twice, one with the baseline scheduler and the other with the Apollo scheduler, to compare the job performance. In order to minimize other random factors, we initially ran them side by side. However, the approach resulted in artificial resource contention as both jobs read the same inputs. Instead, we chose to run the two jobs one after another. Our experiences show that the cluster load is unlikely to change drastically between the two consecutive runs. We also modified the baseline scheduler to produce accurate estimates for task runtime and resource requirements using Apollo's logic so that Apollo could perform adequately well in this mixed mode environment. This allowed us to get performance data from early exposure to large scale environment in the design and experimentation phase.

Migration at Scale. We designed Apollo to replace the previous scheduler *in place*. On the one hand, this means that protocols had to be carefully designed to be compatible; on the other hand, it also means that we had to make sure both schedulers could coexist in the same environment, each scheduling a part of the workload on the same set of machines, without creating interferences. For example, Apollo judiciously performs opportunistic scheduling with significantly less resource. In isolation, Apollo issues 98% less duplicates than the baseline scheduler, without losing performance. However, in the mixed mode where both schedulers runs, the baseline scheduler gains an unfair advantage by issuing more duplicates to get work done. We therefore tuned the system during the transition to increase the probability to start opportunistic tasks for Apollo-scheduled jobs in order to correct the bias caused by the reduction in the number of tasks scheduled.

Deployment at Scale. Without any service downtime or unavailability, we rolled out Apollo to our users in *stages* and increased user coverage over time until eventually fully deployed on all clusters. At each stage, we closely watched various system metrics, verified job performance, and studied impact on other system components before proceeding to the next stage.

One interesting observation was that users who had not yet migrated to Apollo also experienced some performance improvement during the deployment process. This was because Apollo avoided scheduling tasks to hotspots inside the system, which helped improve job performance across the cluster, including the ones sched-

uled by the baseline scheduler. When we finally enabled those jobs with Apollo, the absolute percentage of improvement was less than what we observed initially.

Operation at Scale. Even with thoughtful design and implementation, the dynamic behavior of such a large scale system continues to pose new challenges, which motivate us to refine and improve the system continuously while operating Apollo at scale. For example, Apollo leverages an opportunistic scheduling mechanism to increase system utilization by operating tasks either in normal-priority *regular* mode or in lower-priority *opportunistic* mode. Initially, this priority is enforced by local operating system but not by the underlying distributed file system. During the early validation, this did not pose a problem. However, as we deployed Apollo, both CPU and I/O utilization in the cluster increased. The overall increase in I/O pressure caused a latency impact and interfered with tasks even running in *regular* mode. This highlights the importance of interpreting task priority throughout the entire stack to maintain predictable performance at high utilization.

Another example is that we developed a server failure model by mining historical data and various factors to predict the likelihood of possible repeated server failures in the near future, based on recent observed events. As described in Section 3.3, such failure model has a direct impact on task completion time estimation and thus influences scheduling decisions. The model works great in most cases and helps Apollo avoid scheduling tasks to repeated offenders, thereby improving system reliability. However, a rare power transformer outage caused failures on a large number of servers all at once. After the power was restored, the model predicted that they were likely to fail again and prevented Apollo from using those machines. As a result, the recovery of the cluster was unnecessarily slowed down. This indicates the importance of further distinguishing failure types and dealing with them presumably in different models.

5 Evaluation

Since late 2013, Apollo has been deployed in production clusters at Microsoft, each containing over 20,000 commodity servers. To evaluate Apollo thoroughly, we use a combination of the following methods: (i) We analyze and report various system metrics on large-scale production clusters where Apollo has been deployed; further, we compare the behavior before and after enabling Apollo. (ii) We perform in-depth studies on representative production jobs to highlight our observations at per-job level. (iii) We use trace-driven simulations on certain specific points in the Apollo design to compare with alternatives. Whenever possible, we prefer reporting the production Apollo behavior, instead of using simulated results, because it is hard, if not infeasible, to model the complexity of real workload and production environment faithfully in a simulator. To our knowledge, this is the first detailed analysis of production schedulers at such a large scale with such a complex and diverse workload.

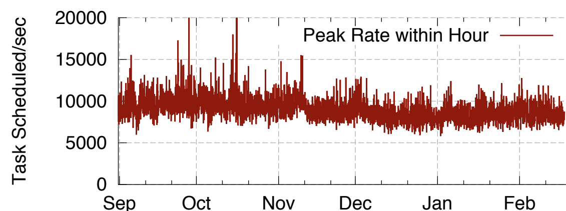


Figure 6: Scheduling rates.

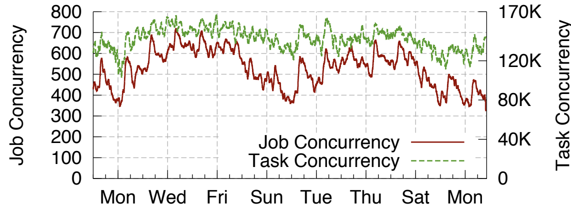
We intend to answer the following questions: (a) How well does Apollo scale and utilize resources in a large-scale cluster? (b) What is the scheduling quality with Apollo? (c) How accurate are the estimates on task execution and queuing time used in Apollo and how much do the estimates help? (d) How does Apollo cope with dynamic cluster environment? (e) What is the complexity of Apollo's core scheduling algorithm?

5.1 Apollo at Scale

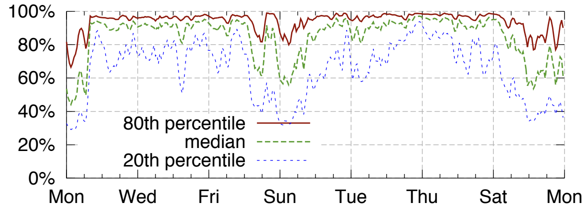
We first measured the aggregated scheduling rate in a cluster over time to understand Apollo's scalability. We define *scheduling rate* as the number of *materialized* scheduling decisions (those resulting a task execution at a PN) made by all the individual schedulers in the cluster per second. Figure 6 shows the peak *scheduling rates* for each hour over the past 6 months, highlighting that Apollo can constantly provide a scheduling rate of above 10,000, reaching up to 20,000 per second in a single cluster. This confirms the need for a distributed scheduling infrastructure, as it would be challenging for any single scheduler to make high quality decisions at this rate. It is important to note that the *scheduling rate* is also governed by the capacity of the cluster and the number of concurrent jobs. With its distributed architecture, we expect the scheduling rate to increase with the number of jobs and the size of the cluster.

We then drill down into a period of two weeks and report various aspects of Apollo, without diluting data over a long period of time. Figure 7(a) shows the number of concurrently running jobs and their tasks in the cluster while Figure 7(b) shows server CPU utilization in the same period, both sampled at every 10 seconds. Apollo is able to run 750 concurrent complex jobs (140,000 concurrent regular tasks) and achieve over 90% CPU utilization when the demand is high during the weekdays, reaching closely the capacity of the cluster.

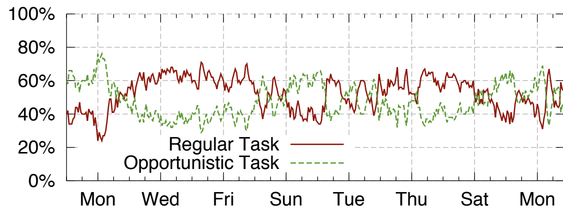
To illustrate the distribution of system utilization among all the servers in the cluster, Figure 7(b) shows the median, as well as the 20th and 80th percentiles in CPU utilization. When the demand surges, Apollo makes use of all the available resources and only leaves a 3%



(a) Concurrent jobs and tasks.



(b) CPU utilization.



(c) CPU time breakdown: regular and opportunistic task.

Figure 7: Apollo in production.

gap between the 20th and the 80th percentiles. When the demand is low, such as on Sundays, the load is less balanced and the machine utilization is mostly correlated to the popularity of the data stored on them. The figures shows a clear weekly pattern of the workloads. The task concurrency decreases during the weekends and recovers back to a large volume during the weekdays. However, the dip in system utilization is significantly less than that of the number of jobs submitted. With opportunistic scheduling, Apollo allows jobs to gracefully exploit idle system resources to achieve better job performances and continues to drive system utilization high even with fewer number of jobs. This is further validated in Figure 7(c), which shows the percentage of CPU hours attributed to regular and opportunistic tasks. During the weekdays, 70% of Apollo's workload comes from regular tasks. The balance shifts during the weekends: more opportunistic tasks get executed on the available resources when there are fewer regular tasks.

Task location	% Tasks	% I/Os
The same server that contains the input	28%	46%
Within the same rack as the input	56%	47%
Across rack	16%	7%

Table 1: Breakdown of tasks and their I/Os.

We also measured the average task queuing time for all regular tasks to verify that the queuing time remains low despite the high concurrency and system utilization. At the 95th percentile, the tasks show less than 1 second queuing time across the entire cluster. Apollo achieves this by considering data locality, wait time, and other factors holistically when distributing tasks. Table 1 categorizes tasks into three groups and reports the percentage of I/Os they account for. 72% of the tasks are dispatched to servers that require reading inputs remotely, either within or across rack, to avoid wait time. If only data locality is considered, tasks are likely to concentrate on a small group of servers that contain hot data.

Summary. Combined, those results show that Apollo is highly scalable, capable of scheduling over 20,000 requests per second, and driving high and balanced system utilization while incurring minimum syqueuing time.

5.2 Scheduling Quality

We evaluate the scheduling quality of Apollo in two ways: (i) compare with the previously implemented baseline scheduler using production workloads and (ii) study business critical production jobs and use trace-based simulations to compare the quality.

Performing a fair comparison between the baseline scheduler and Apollo in a truly production environment with real workload is challenging. Fortunately, we replaced the baseline scheduler in place with Apollo, allowing us to observe both schedulers in the same cluster with similar workloads. Further, about 40% of the production jobs in the cluster has a *recurring* pattern and such recurring jobs account for more than 75% system resource utilization [5]. We therefore choose two time frames, before and after the Apollo deployment, to compare performance and speedup of each recurring job, running Apollo and the baseline scheduler respectively. The recurring nature of the workload produced a strong correlation in CPU time between the workloads in the two time frames, as shown in Figure 8(a). Figure 8(b) shows the CDF of the speedups for all recurring jobs and it indicates that about 80% of recurring jobs receive various degrees of performance improvements (up to 3x in speedup) with Apollo. To understand the reason for the differences, we measured the average task queuing time on each server for every window of 10 minutes. Figure 8(c) shows the standard deviation of the average task queue time across servers, comparing Apollo with the baseline scheduler, which indicates clearly that Apollo achieves much more balanced task queues across servers.

For the second experiment, we present a study of one business critical production job, which runs every hour. The job consumes logs from a search and advertisement engine and analyzes user click information. Its execution graph consists of around ten thousands tasks, processing a few terabytes of data. The execution graph shown in Figure 1 is a much simplified version of this

(a) Comparable workloads. (b) Recurring job speedup. (c) Scheduling balance.

Figure 8: Comparison between Apollo and the baseline scheduler.

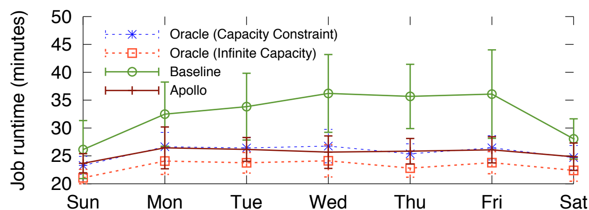


Figure 9: Job latencies with different schedulers.

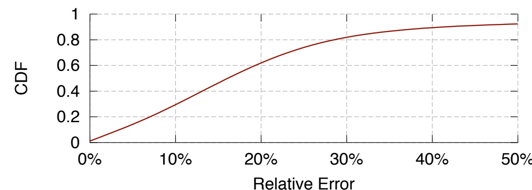


Figure 10: CPU time estimation.

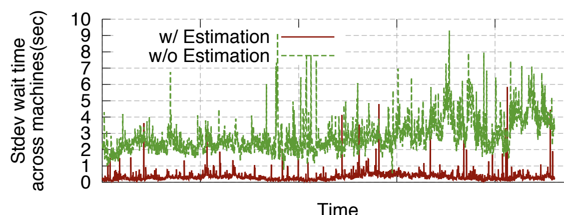


Figure 11: Scheduling balance (estimation effect).

job. The performance of the job varies by weekdays because of periodic fluctuations in the input volume of user clicks. To compare performance, we use one job per day at the same hour in a week and evaluated the scheduling performance of Apollo, baseline scheduler, and a simulated oracle scheduler, which has zero task wait time, zero scheduling latency, zero task failures, and knows exact runtime statistics about each task. Further, we use two variants of the oracle scheduler: (i) oracle with capacity constraint, which is limited to the same capacity that was allocated to the job when it ran in the production environment and (ii) oracle without capacity constraint, which has access to unlimited capacity, roughly representing the best case scenario.

Figure 9 shows job performance using Apollo and the baseline scheduler, respectively, and compares them with the oracle scheduler using runtime traces. On average, the job latency improved around 22% with Apollo over the baseline scheduler, and Apollo performed within 4.5% of the oracle scheduler. On some days, Apollo is even better than the oracle scheduler with the capacity constraint because the job is able to get some extra speedup from opportunistic scheduling, allowing the job to get more capacity than the capacity constraint used by the oracle scheduler.

Summary. Apollo delivers excellent job performance compared with the baseline scheduler and its scheduling quality is close to the optimal case.

5.3 Evaluating Estimates

Estimating task completion time, as described in Section 3.3, plays an important role in Apollo's scheduling algorithm and thus job performance. Both task initialization time and I/O time can be calculated when the inputs and server locations are known at runtime.

We first measure the accuracy of the estimated task wait time, as a result of applying task resource estimation to the wait-time matrix. Over 95% tasks have a wait time estimation error of less than 1 second. We then measure the CDF of the estimation error for task CPU time, as shown in Figure 10. For 75% of tasks, the CPU time predicted when the task is scheduled is within 25% of the actual CPU consumption. Apollo continues to refine runtime estimates based on statistics from the finished tasks within the same stage at runtime. Nevertheless, a number of factors make runtime estimation challenging. A common case is for tasks with *early-out* behavior without reading all of its input. An example of such tasks may consist of a filter operator followed by a `TOP` N operator. Different tasks may consume different amount of input data before collecting N rows satisfying the filter condition, which makes inference based on past task runtime difficult. Complex user code whose resource consumption and execution time varies by input data characteristics also makes prediction difficult, if not infeasible. We evaluate how Apollo dynamically adjusts scheduling decisions at runtime in Section 5.4.

In order to evaluate the overall estimation impact, we compare the Apollo performance with and without estimation. As we rolled out Apollo to one production cluster, we went through a phase in which we used a default estimate for all tasks uniformly, before we enabled all the internal estimation mechanism. We refer the phase

as Apollo without estimation. Comparing the system behavior before and after allows us to understand the impact of estimation on scheduling decisions in a production cluster because the workloads are similar as we reported in Section 5.2. Figure 11 shows the distributions of task queuing time. With estimation enabled, Apollo achieves much more balanced scheduling across servers, which in turn leads to shorter task queuing latency.

Summary. Apollo provides good estimates on task wait time and CPU time, despite all the challenges, and estimation does help improve scheduling quality. Further improvements can be achieved by leveraging statistics of recurring jobs and better understanding task internals, which is part of our future work.

5.4 Correction Effectiveness

In case of inaccurate estimates or sudden changes in a cluster environment, Apollo applies a series of correction mechanisms to mitigate effectively. For duplicate scheduling, we call a duplicate task successful if it starts before the initial task.

Conditions (W: wait time)	Trigger rate	Success rate
New expected W significantly higher	0.12%	81.3%
Expected W greater than average	0.12%	81.3%
Elapsed W greater than average	0.17%	83.0%

Table 2: Duplicate scheduling efficiency.

Table 2 evaluates different heuristics of duplicate scheduling, described in Section 3.4, for the same two-week period and reports how frequently they are triggered and their success rate. Overall, Apollo's duplicate scheduling is efficient, with 82% success rates, and accounts for less than 0.5% of task creations. Such a low correction rate confirms the viability of optimistic scheduling and deferred corrections for this workload.

Straggler detection and mitigation is also important for job performance. Apollo is able to catch more than 70% stragglers efficiently and apply mitigation timely to expedite query execution. We omit the detailed experiments due to space constraints.

Summary. Apollo's correction mechanisms are shown effective with small overhead.

5.5 Stable Matching Efficiency

In a general case, the complexity of the stable matching algorithm, when using a red-black tree to maintain a sorted set of tasks to schedule, is $O(n^2)$ while the greedy algorithm is $O(n\log(n))$. However in our case the complexity of the stable matching algorithm is limited to $O(n\log(n))$. The algorithm usually converges in less than 3 iterations and our implementation limits the number of iterations of the matcher, which makes the worst case complexity $O(n\log(n))$, the same as the greedy algorithm. In practice, a scheduling batch contains less than 1,000 tasks and the computation overhead is negligible, with no observed performance differences.

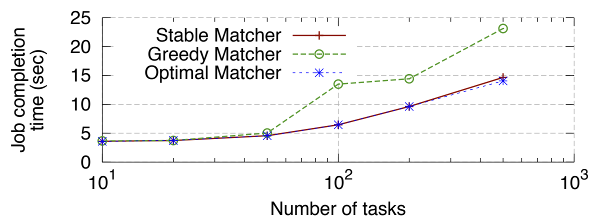


Figure 12: Matching quality.

We verify the effectiveness of the stable matching algorithm using a simulator. We measure the amount of time it takes for a workload of N tasks to complete on 100 servers, using the greedy, stable matching, and optimal matching algorithms, respectively. The optimal matching algorithm uses an exhaustive search to compute the best possible sequence of scheduling. Each server has a single execution slot and have an expected wait time that is exponentially distributed with an average of 1. The expected runtime of each tasks is exponentially distributed with an average of 1. Each task is randomly assigned a server preference and runs faster on the preferred server. Figure 12 shows that the stable matching algorithm performs within 5% of the optimal matching under the simulated conditions while the greedy approach, which schedules tasks one at a time on the server with the minimum expected completion time, was 57% slower than the optimal matching.

Summary. Apollo's matching algorithm has the same asymptotic complexity as a naive greedy algorithm with negligible overhead. It performs significantly better than the greedy algorithm and is within 5% of the optimal scheduling in our simulation.

6 Related Work

Job scheduling was extensively studied [24, 25] in high-performance computing for scheduling batch CPU-intensive jobs and has become a hot topic again with emerging data-parallel systems [7, 15, 29]. Monolithic schedulers, such as Hadoop Fair Scheduler [28] and Quincy [16], implement a scheduling policy for an entire cluster using a centralized component. This class of schedulers suffers from scalability challenges when serving large-scale clusters.

Mesos [14] and YARN [27] aim to ease the support for multiple workloads by decoupling resource management from application logic in a two-layer design. Facebook Corona [1] uses a similar design but focuses on reducing job latency and improving scalability for Hadoop by leveraging a push-based message flow and an optimistic locking pattern. Mesos, YARN, and Corona remain fundamentally centralized. Apollo in contrast makes decentralized scheduling decisions with no central scheduler, which facilitates small task scheduling.

Distributed schedulers such as Omega [23] and Sparrow [22] address the scalability challenges by not relying on a centralized entity for scheduling. Two different approaches have been explored to resolve conflicts. Omega resolves any conflict using optimistic concurrency control [17] where only one of the conflicting schedulers succeeds and the others have to roll back and retry later. Sparrow instead relies on (random) sampling and speculative scheduling to balance the load.

Similar to Omega, schedulers in Apollo makes optimistic scheduling decisions based on their views of the cluster (with the help of the RM). Unlike Omega, which detects and resolves conflicts at the scheduling time, Apollo is optimistic in conflict detection and resolution by deferring any corrections until after tasks are dispatched. This is made possible by Apollo's design of having local queues on servers. The use of local task queue and task runtime estimates provides critical insight about future resource availability and allows Apollo schedulers to optimize task scheduling by estimating task completion time, instead of based on instantaneous resource availability at the scheduling time. This also gives Apollo the extra benefit of masking resource-prefetching latency effectively, which is important for our target workload.

Although both adopting a distributed scheduling framework, Sparrow and Apollo differ in how they make scheduling decisions. Sparrow's sampling mechanism will schedule tasks on machines with the shortest queue, with no consideration for other factors affecting the completion time, such as locality. In contrast, Apollo schedulers optimize task scheduling by estimating task completion time that takes into account multiple factors such as load and locality. Sparrow uses reservation on multiple servers with late binding to alleviate its reliance on queue length, rather than task completion time. Such a reservation mechanism would introduce excessive initialization costs on multiple servers in our workload. Apollo introduces duplicate scheduling only as a correction mechanism; it is rarely triggered in our system.

While Omega and Sparrow have been evaluated using simulation and a 110-machine cluster respectively, our work distinguishes itself by showing Apollo's effectiveness in a real production environment at a truly large scale, with diverse workloads, and complex resource and job requirements.

Capacity management often goes hand-in-hand with scheduling. Capacity scheduler [2] in Hadoop/YARN uses global capacity queues to specify the share of resources for each job, which is similar to token-based resource guarantee implemented in Apollo. Apollo uses fine grained allocations and opportunistic scheduling to take advantage of idle resources gracefully. Resource management on local servers is also critical. Existing work leverages Linux containers [13], usage monitoring [27] and/or contention detection [31] to provide performance isolation. Apollo can accommodate any of those mechanisms.

Operator runtime estimation based on data statistics and operator semantics has been extensively studied in the database community for effective query optimization [19, 11, 6]. For distributed computing of arbitrary input and program, most effort (e.g., ParaTimer [21]) has been focusing on estimating the progress of running jobs based on runtime statistics. Apollo combines both static and runtime information and leverages program patterns (e.g., stage) to estimate task runtime. Apollo also includes a set of mechanisms to compensate inaccuracy whenever needed. For example, many existing works on outlier detection and straggler mitigation (e.g., LATE [30], Mantri [4], and Jockey [9]) are complementary to our work and can be integrated with the Apollo framework for reducing job latency.

7 Conclusion

In this paper, we present Apollo, a scalable and coordinated scheduling framework for cloud-scale computing. Apollo adopts a distributed and loosely coordinated scheduling architecture that scales well without sacrificing scheduling quality. Each Apollo scheduler considers various factors holistically and performs estimation-based scheduling to minimize task completion time. By maintaining a local task queue on each server, Apollo enables each scheduler to reason about future resource availability and implement a deferred correction mechanism to effectively adjust suboptimal decisions dynamically. To leverage idle system resources gracefully, opportunistic scheduling is used to maximize the overall system utilization. Apollo has been deployed on production clusters at Microsoft: it has been shown to achieve high utilization and low latency, while coping well with the dynamics in diverse workloads and large clusters.

Acknowledgements

We are grateful to our shepherd Andrew Warfield for his guidance in the revision process and to the anonymous reviewers for their insightful comments. David Chu and Jacob Lorch provided feedback that helped improve the paper. We would also like to thank the members of Microsoft SCOPE team for their contributions to the SCOPE distributed computation system, of which Apollo serves as the scheduling component. The following people contributed to our scheduling framework: Haochuan Fan, Jay Finger, Sapna Jain, Bikas Saha, Sergey Shelukhin, Sen Yang, Pavel Yatsuk, and Hongbo Zeng. Finally, we would like to thank Microsoft Big Data team members for their support and collaboration.

References

[1] Under the hood: Scheduling MapReduce jobs more efficiently with Corona. http://on.fb.me/TxUsYN, 2012. [Online; accessed 16-April-2014].

[2] Hadoop Capacity Scheduler. http://hadoop.apache.org/docs/r2.3.0/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html, 2014. [Online; accessed 16-April-2014].

[3] Hadoop Distributed File System (HDFS). http://hadoop.apache.org/, 2014. [Online; accessed 16-April-2014].

[4] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in MapReduce clusters using Mantri. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[5] N. Bruno, S. Agarwal, S. Kandula, B. Shi, M.-C. Wu, and J. Zhou. Recurring job optimization in scope. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 805–806, New York, NY, USA, 2012. ACM.

[6] S. Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 34–43. ACM, 1998.

[7] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[8] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12. ACM, 1987.

[9] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 99–112, New York, NY, USA, 2012. ACM.

[10] D. Gale and L. S. Shapley. College admissions and the stability of marriage. *American Mathematical Monthly*, 69(1):9–15, Jan. 1962.

[11] P. Gassner, G. M. Lohman, K. B. Schiefer, and Y. Wang. Query optimization in the IBM DB2 family. *IEEE Data Eng. Bull.*, 16(4):4–18, 1993.

[12] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[13] M. Helsley. LXC: Linux container tools. *IBM devloperWorks Technical Library*, 2009.

[14] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 22–22, Berkeley, CA, USA, 2011. USENIX Association.

[15] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[16] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 261–276, New York, NY, USA, 2009. ACM.

[17] H.-T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[18] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[19] L. F. Mackert and G. M. Lohman. *R* optimizer validation and performance evaluation for local queries*, volume 15. ACM, 1986.

[20] M. L. Massie, B. N. Chun, and D. E. Culler. The ganglia distributed monitoring system: Design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.

[21] K. Morton, M. Balazinska, and D. Grossman. ParaTimer: A progress indicator for MapReduce DAGs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 507–518. ACM, 2010.

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

[23] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: Flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 351–364, New York, NY, USA, 2013. ACM.

[24] G. Staples. TORQUE resource manager. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 8. ACM, 2006.

[25] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: The Condor experience. *Concurrency and Computation: Practice and Experience*, 17(2-4):323–356, 2005.

[26] R. Van Renesse, K. P. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems (TOCS)*, 21(2):164–206, 2003.

[27] V. K. Vavilapalli. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proc. SOCC*, 2013.

[28] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 265–278, New York, NY, USA, 2010. ACM.

[29] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[30] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *OSDI*, volume 8, page 7, 2008.

[31] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. CPI2: CPU performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 379–391, New York, NY, USA, 2013. ACM.

[32] J. Zhou, N. Bruno, M.-C. Wu, P.-Å. Larson, R. Chaiken, and D. Shakib. SCOPE: Parallel databases meet MapReduce. *VLDB J.*, 21(5):611–636, 2012.