Canary: A Scheduling Architecture for High Performance Cloud Computing

Hang Qu, Omid Mashayekhi, David Terei, Philip Levis
Stanford University
{*quhang, omidm*}@*stanford.edu*, {*dterei, pal*}@*cs.stanford.edu*

Abstract

We present Canary, a scheduling architecture that allows high performance analytics workloads to scale out to run on thousands of cores. Canary is motivated by the observation that a central scheduler is a bottleneck for high performance codes: a handful of multicore workers can execute tasks faster than a controller can schedule them.

The key insight in Canary is to reverse the responsibilities between controllers and workers. Rather than dispatch tasks to workers, which then fetch data as necessary, in Canary the controller assigns data partitions to workers, which then spawn and schedule tasks locally.

We evaluate three benchmark applications in Canary on up to 64 servers and $1,152$ cores on Amazon EC2. Canary achieves up to $9-90\times$ speedup over Spark and up to $4\times$ speedup over GraphX, a highly optimized graph analytics engine. While current centralized schedulers can schedule 2,500 tasks/second, each Canary worker can schedule 136,000 tasks/second per core and experiments show this scales out linearly, with 64 workers scheduling over 120 million tasks per second, allowing Canary to support optimized jobs running on thousands of cores.

1 Introduction

Data analytics frameworks such as Spark [36] and Naiad [24] provide high-level programming abstractions that operate on datasets far too large for a single server's memory. These frameworks parallelize an application (called a *job*) across tens or hundreds of servers by breaking it into many small *tasks*. Many data analytics applications are CPU-bound [27]; breaking them into small tasks lets them use more cores, and parallelizing across more cores reduces completion time.

Micropartitioning each computational step (a *stage*) into multiple tasks per core can reduce completion time further. Cloud analytics applications have highly variable task execution times: some tasks can run twice or ten times as long as others, due to data skew or variations in node performance. [4] If each stage is micropartitioned into multiple tasks per core, busy cores can shed tasks to idle ones. Furthermore, the runtime can interleave computation and communication, sending the results of one task while computing the results of a second. [31]

But increasing the number of tasks has a cost. Although an analytics job is parallelized across many servers (called *workers*), a single server, the *controller*, is responsible for scheduling tasks for many jobs to a shared cluster of workers. A modern analytics controller can schedule 1,500-2,500 tasks per second. As the number of workers and tasks increases, the scheduler becomes a bottleneck. [28]

Systems such as Sparrow [28], Omega [32], Apollo [6], and Mercury [19] try to sidestep the scheduler bottleneck by allowing each job to have its own private scheduler. Through a variety of mechanisms (monitoring worker load, batching, power of two choices, central load databases), these systems allow each scheduler to find and use idle cores in a shared cluster while preventing flash mobs or other synchronized behaviors.

But even a single optimized job can easily saturate a scheduler. In Section 2, we show a C++ implementation of a standard analytics benchmark that executes three tasks per core every 470ms. Tasks take on average 150ms to complete, so each core can run 6 tasks/second. Modern Amazon EC2 instances (m4.10xlarge) have 40 cores: a single job running on 8 workers (320 cores) can execute 2,000 tasks/second, close to the limit of a modern controller. As CPU-bound analytics workloads increasingly call into GPUs or native C libraries to improve performance and the number of cores per processor increases, scheduling and controllers are emerging as a bottleneck.

MPI-based high performance computing applications can scale to run on tens of thousands of cores with tasks that are as short as tens or hundreds of microseconds by having no controller at all. Without a controller, however, no node knows the overall network and application state. Accordingly, MPI applications are notoriously difficult

to load balance, and so use one task per core and are carefully designed so that each stage's tasks have near-uniform execution times, in some cases even by running empty loops. Furthermore, the lack of a central controller makes MPI applications brittle to faults and failures.

This paper proposes a new scheduling architecture, called Canary, that can support orders of magnitude more tasks per second than existing approaches. The key insight behind Canary is to reverse the responsibilities between controllers and workers. Rather than have a controller schedule tasks to workers, which are then responsible for fetching data, in Canary the controller only decides how data are distributed on workers. Each worker locally spawns its own tasks based on the current program position and the data partitions the controller has assigned to it. The controller remains responsible for load balancing and fault tolerance, while workers take on the responsibility of scheduling and executing the job in a distributed way.

Canary is implemented in a C-based data analytics framework written from scratch because the performance of existing open-source frameworks falls far behind a fully optimized implementation [23]. The inefficiency of these frameworks prevents understanding realistic workload characteristics because their tasks run so slowly. In Section 2 we show how simple C++ re-implementations of analytics benchmarks in this framework can speed them up by factors of 4-90 over Spark [36] and GraphX [14].

This paper evaluates Canary's scalability using three applications, logistic regression, k-means clustering and PageRank. Our results show that using Canary, a single core can schedule 136,000 tasks/second, and a 64-node, 1152-core cluster can schedule 120 million tasks per second. This allows Canary to run benchmarks orders of magnitude faster as the scheduler is no longer a bottleneck. This paper makes three research contributions:

- an analysis that shows centralized scheduling cannot scale because scheduler load grows quadratically with the number of cores,
- a novel scheduling architecture which scales to support orders of magnitude more tasks per second by decoupling data placement management and task execution scheduling, and
- optimized implementations of three data analytics applications that demonstrate the benefits of a highly scalable scheduler.

Section 2 gives an overview of the performance bottlenecks of existing analytics frameworks and why their scheduling architecture limits the scale at which their applications can run. Section 3 presents the Canary system architecture, Section 4 presents the design of Canary workers, and Section 5 evaluates its performance and scalability. Section 6 describes the related work that Canary extends and builds on, and Section 7 concludes.

2 Motivation: Scalable Scheduling

This section gives an overview of analytics frameworks and their current bottlenecks. It shows how numerous optimizations to analytics have a corresponding cost: the *task rate*, or number of tasks per second the job executes, goes up, increasing load on the scheduler. Optimizing analytics codes (e.g., using C rather than functional Scala code) can improve performance and increase the task rate by a factor of 90. Scaling out a CPU-bound workload to more cores causes the task rate to increase quadratically with the number of cores used. Finally, breaking a job into smaller tasks reduces job completion time by allowing more flexible load balancing, but also increases the task rate. Together, these results mean that existing schedulers can barely keep up with a handful of well-tuned workers. This motivates the need for a new scheduling architecture that can handle orders of magnitude more tasks per second than current schedulers do.

2.1 The Cost of Fast Analytics

The power of MapReduce [9], Pregel [22], Spark [36] and other distributed computation frameworks is in great part due to their ability to scale out to run on many nodes in parallel. For I/O-centric workloads, which MapReduce targets, individual tasks can take tens or hundreds of seconds [7] and so even when scaled out to large numbers of machines the aggregate number of tasks per second the system must schedule is low. Furthermore, because these workloads are I/O bound, rather than compute bound, the increasing number of cores per CPU does not greatly increase task rate.

In contrast, modern analytics workloads are CPU-bound [27]. Improving their CPU performance makes their tasks run faster and jobs complete in less time. Table 1 shows the job completion time of a standard Spark benchmark, logistic regression, running on a single 18-core worker. One iteration of a standard Spark implementation, written in its functional language, takes 42 seconds. A highly optimized Spark implementation that uses Spark's lowest level API can run 16 times faster, in 2.6 seconds. Written in C, using the analytics framework we have described in Section 5.1, each iteration takes 470 milliseconds, running 90 times faster than the standard Spark implementation. In the C implementation, one iteration involves 3 tasks per partition. If there is one partition per core, a single 18-core worker executes more than 120 tasks per second.

	Time (s)	Tasks/sec
Spark (functional)	42.36	0.4
Spark (imperative)	6.52	2.8
Spark (low level API)	2.59	7.0
Canary	0.47	123.4

Table 1: Execution time and task rate of one iteration of a logistic regression job that processes 20GB of training data on a server with 18 physical cores. Each Spark iteration is 18 tasks (1 per core) while each Canary iteration is 58 tasks (3 tasks per core plus 4 global tasks). Spark (functional) **is the standard implementation which uses functional operators like** `map` **and** `zip`**.** Imperative **replaces the functional operators with loops.** Low level API **uses** `mapPartitions` **to manually iterate over data.** Canary **is a C++ implementation.**

2.2 The Cost of Scaling Out

Scaling out to more workers both increases the number of tasks that must be scheduled and decreases how long each of those tasks takes. As a result, the number of tasks per second that must be scheduled increases quadratically with the number of worker cores.

Suppose we have a CPU-bound analytics job which, if all data are in memory, takes C cycles to compute. If C is evenly distributed across W worker cores, then a scheduler will need to schedule one task per stage per core, or W tasks. If the job is CPU-bound and scales well, then parallelizing it across W cores will cause it to run W times faster; the computation will take $\frac{C}{W}$ time. A scheduler's load is therefore

$$\Theta\left(\frac{W}{\frac{C}{W}}\right) = \Theta\left(\frac{W^2}{C}\right) \quad (1)$$

tasks per second. As long as C does not change much with greater parallelism, the task rate a job generates for a scheduler scales with $\Theta(W^2)$.

2.3 The Cost of Utilization

Recent research [26] as well as experiences of core members of Google's datacenter software [8] have shown that *micropartitioning* a computational stage more finely than one partition per core has significant benefits. Micropartitioning enables better load balancing and faster completion times. If some tasks take longer than others (e.g., due to data skew), then workers with longer tasks can shed some of their other tasks to less heavily loaded workers. Prior work shows that micropartitioning a stage into 10-20 tasks/core has significant performance benefits, reducing completion time by a factor of 2-10 by keeping cores at high utilization.

Suppose each stage in a job is split into P tasks per core. Following the analysis above, this means the number of tasks that must be scheduled grows with $\Theta(P \cdot W^2)$.

2.4 The Need for a New Scheduling Architecture

Applying these analyses to the performance numbers presented above, a single worker running the logistic regression benchmark in Table 1 with $P = 10$ (a common setting), can execute 1,200 tasks per second. If the job were parallelized to run on two workers, completion time would be halved, but there would be twice as many tasks to execute, so the task rate would increase by a factor of 4, to 4,800 tasks per second. Scaling out the job to 8 workers would increase the task rate to $16 \cdot 4,800$, or nearly 80,000, tasks per second.

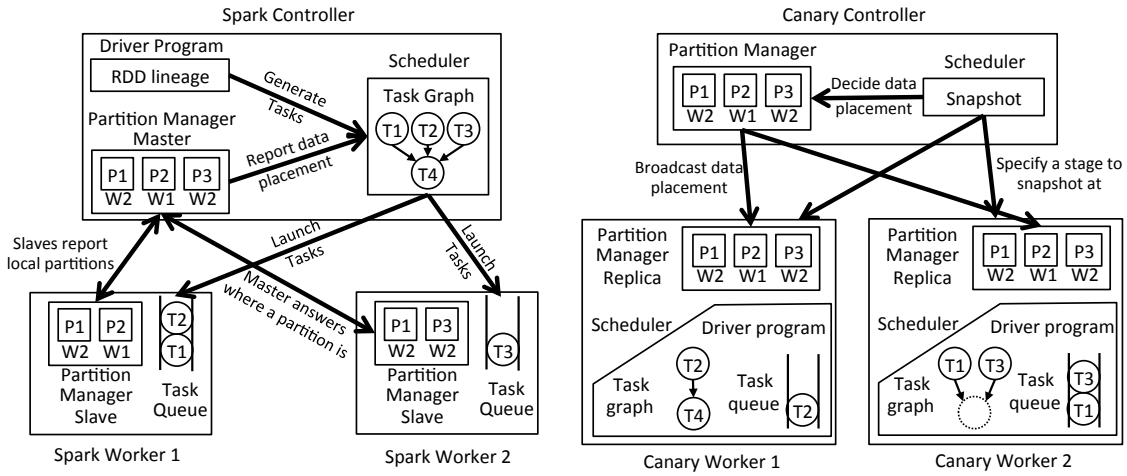
Modern analytics controllers can schedule 2,000 - 3,000 tasks per second.[1] If jobs run CPU-optimized code and try to parallelize to many modern multi-core machines, the scheduler quickly becomes a bottleneck; we defer a detailed presentation of our experimental results to Section 5, but for basic benchmarks written in Spark, the Spark scheduler is already a bottleneck for 32 18-core workers when $P = 1$, and setting $P = 10$ limits Spark to 3 workers.

If next generation frameworks are to scale out for high performance analytics workloads, they will need a new scheduling architecture. The next section describes such an architecture, whose scheduler's load is independent of the number of workers and instead only acts when load is rebalanced across workers.

3 System Overview

Canary is a distributed scheduler architecture for cloud data processing systems. The key insight behind Canary is that a worker can determine which tasks it should execute based on what data partitions it has in memory. By placing the data partitions a task uses onto a worker, the controller implicitly assigns that task to that worker. In the extreme case, if every worker knows the placement of every data partition and partitions never move, workers can run and complete an entire job without any controller coordination. This section describes the system architecture, execution model, and central controller. The next section provides details on how workers locally spawn and schedule tasks as well as how they exchange data and migrate partitions.

[1] Ousterhout et al. report the Spark scheduler can handle 1,500 tasks per second. Our measurements in Section 5 show that a recent Spark release (1.5.2, November 2015) improves the throughput to 2,600 tasks per second.



(a) Existing centralized scheduling architecture. **(b) Canary scheduling architecture.**

Figure 1: The Canary controller distributes task generation, and task dependency analysis to workers. Canary workers infer what tasks to run based on what data partitions are local, reason about task dependency from a local driver program copy, and decide what tasks to run and when, all without the controller's help.

3.1 Architecture

A Canary controller is responsible for deciding how data partitions are distributed across workers, deciding when partitions should migrate between workers, and coordinating partition migrations to maintain program correctness. Each Canary worker has a copy of the driver program, which it uses to locally spawn and schedule tasks. It decides which tasks to spawn based on what data partitions are resident in its local memory, as determined by the controller. Controllers today decide task placement and leave it to workers to move partitions so they can execute those tasks. Canary inverts this model, as its controller decides partition placement and leaves it to workers to generate the tasks to compute on their data. Figure 1 shows the Canary architecture in comparison to existing scheduler architectures.

Canary's model makes two assumptions about how data partitions behave. First, it assumes that the set of data partitions changes slowly. For analytics workloads, which typically involve many iterations over in-memory data (e.g., graph algorithms, regression, signal processing), this assumption is valid. If workers are continually generating many new, short-lived partitions, then this activity could lead to significant load at the controller. Second, it assumes there are far fewer data partition migration events than tasks executions, that is, a worker will execute many tasks on a partition before possibly migrating it. This assumption is valid for any CPU-bound workload, as migrating a partition generally takes much more than than computing tasks on it; if there are many migration events, the workload is I/O, rather than compute, bound.

3.2 Execution Model

Canary's execution model is very similar to other cloud computing frameworks. A job is written as a *driver*, a sequential program whose variables are large datasets. Each variable has an associated partitioning function, which defines how many partitions the variable is split into. The program can apply parallel operations on these datasets, which perform the computation on the partitions in parallel. Just as in Spark, if an operator takes multiple variables as input, each of those variables must have the same partitioning. Otherwise, it is not possible to properly parallelize the computation. Moving data from one partitioning to another is a shuffle operation, just as in the MapReduce programming model.

Canary's execution and programming model differs from centralized schedulers in one important way: all variables are datasets. Because there is no single, centralized copy of the driver program, there is no notion of a scalar variable such as a loop counter or a parameter passed through a closure (as is common with Spark). Since the driver executes on every worker, such a variable would be replicated across them. Instead, a program can define a dataset that has a single partition; it is then up to the controller to decide where this partition resides.

The execution model allows loops or conditional branches based on runtime results, with one constraint. Every worker must make the same runtime control flow decision, i.e., running the same number of iterations and taking the same conditional branch. As explained in Section 4.4, this is so that workers have a consistent enough view of program execution that they can correctly migrate partitions and exchange data. Since there are no global

variables, this implicitly means that any conditional execution requires a barrier operation.

3.3 Controller

The Canary controller has four major responsibilities:

- compute and update the *partition map*, which specifies how partitions are distributed across workers,
- distribute the partition map to workers,
- coordinate worker execution so they migrate partitions safely when it changes the partition map, and
- decide when workers should store a snapshot of their partitions to durable storage for failure recovery.

A data partition is named by a (name,index) tuple. Name is the variable name in the program, and index is an integer in the range of $[0, n-1]$, where n is the number of partitions. The partition map is a table that specifies, for each partition, which worker has that partition in memory. In-memory partitions are not replicated. If a worker fails, the controller instructs other workers to reload the lost partitions from the last snapshot to durable storage, which is replicated. If the snapshot is old enough that it is not possible to recover the current state of a partition, the controller asks other workers to restore their partitions as well, rolling the entire computation back to the snapshot.

The controller periodically queries workers for load information, including how much time their CPU is idle and how much CPU time they have spent computing tasks on each partition. When a worker is completing stages significantly slower than the average, the controller chooses another (lightly loaded) worker and moves some of the slow worker's partitions to the lightly loaded one. It does this by updating the partition map, sending those updates to every worker, and sending control messages to the workers to perform the transfer. Section 4.4 describes how the workers independently coordinate when to safely perform the transfer among themselves.

The partition map is a table, so updates involve simple changes to the worker field of the rows for migrating partitions. One important property of distributing partition map updates is that they do not need to be strongly consistent: sequential consistency is sufficient. This is because migrations, since they rely on explicit commands from the controller, do not directly use the partition map. Instead, the principal role the partition map plays on a worker is to inform it whom it needs to exchange data with. In cases where a worker's partition map is stale or incorrect, the data exchange protocol described in Section 4.2 can detect this and update the stale partition map.

Workers autonomously generate tasks based on which partitions they have. It is therefore possible to distribute partitions in a way that will prevent a program from executing. For example, consider a stage S that has two inputs A and B, each of which has two partitions, for a total of 4 partitions: A_1, A_2, B_1, and B_2. If the controller places A_1 on one worker and A_2, B_1, and B_2 on the other, then no worker can compute $S(A_1, B_1)$. The controller therefore uses the driver program to compute a set of constraints on what partition placements are valid and always updates the partition map in a way that meets these constraints. In the example above, if one worker had all four partitions and was overloaded, the controller would decide to migrate either A_1 and B_1 or A_2 and B_2 simultaneously.

Tasks can compute on in-memory data much faster than it can be written to durable storage or replicated across workers. Therefore, rather than relying on replicating in-memory data partitions, Canary borrows techniques from high performance computing for durability and fault tolerance. This has the added benefit that it reduces memory use and so allows a cluster to process larger datasets. A controller periodically sends workers a snapshot command. The command specifies which stage the workers should snapshot at. The selected stage must be a stage that acts as a barrier, which all workers must complete before continuing. A shuffle stage, for example, which remaps data from one partitioning to another, is a barrier because all of the input partitions must be processed into output partitions before computation may continue. When the workers complete execution of this stage, they stop execution and store their partitions to durable storage.

3.4 Discussion

If a job is executing smoothly and is well balanced, a controller does little more than periodically collect performance information. This idleness is intentional, in order to prevent the controller from becoming a bottleneck. As a result, each worker core has much more responsibility: it must spawn and schedule tasks, coordinate partition migration, and transfer data with other workers. But because this work is parallelized across every worker core, it takes up at most a few percent of worker CPU cycles. The next section describes the abstractions and mechanisms workers use to achieve this behavior.

4 Workers

The key challenge in Canary is enabling workers, in a completely distributed way, to

- correctly spawn tasks and compute their scheduling dependencies,
- exchange data with other workers,
- maintain a consistent view of execution, and
- migrate partitions.

The rest of this section explains how they do so. To ground these explanations in a concrete example, we use

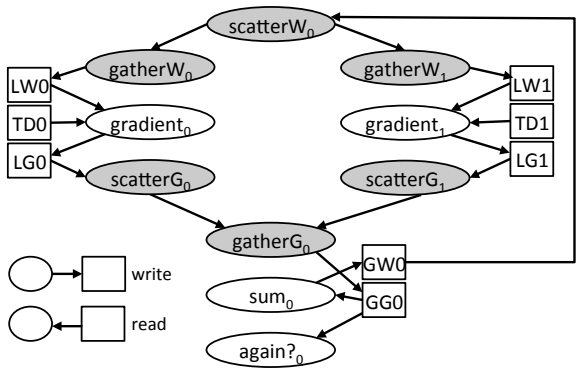


Figure 2: A logistic regression application implemented in Canary. Grey ovals are compute tasks, white ovals are communication tasks, and boxes are partitions. LW is the local weight, TD is training data, LG is the local gradient, GW is the global weight and GG is the global gradient.

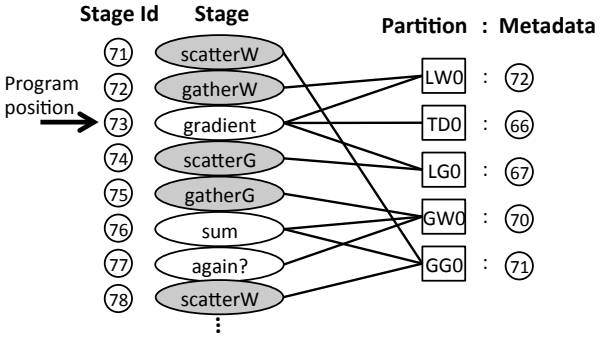


Figure 3: Metadata a worker keeps on the logistic regression application. Each stage has an identifier, and a worker keeps track of which was the last stage to access each partition.

a standard data analytics benchmark, logistic regression. Logistic regression iteratively computes a gradient value for each data partition, combines the gradient values to derive a global weight value, then uses the global weight value as a parameter for the next iteration.

Figure 2 shows the structure of this application in terms of stages and datasets. To calculate a gradient value for each training data partition, the `gradient` stage reads the `training_data` dataset and `local_weight` dataset and writes the `local_gradient` dataset. To sum up local gradient values to the global gradient value, a scatter stage packs local gradient values as intermediate chunks, and a gather stage sums up values in those intermediate chunks and writes to `global_gradient` dataset.

4.1 Program Representation and Execution

Each Canary worker has a full copy of the job's driver program. The worker enumerates the stages in the program and uses these as identifiers to specify program position. For each partition, the worker maintains which was the last stage that read from or wrote to that partition. Figure 3 shows the metadata a worker maintains for the logistic regression application.

A Canary worker borrows the concept of a "software processor" from the Legion framework [5]. In this approach, programs appear to run sequentially, but underneath the runtime may execute stages out of order to improve parallelism, just as a modern CPU does with instructions. The worker is therefore responsible for determining stage dependencies, for example, that the `scatterG` communication stage in Figure 3 cannot run until the `gradient` computational stage has completed. Given a driver program, the worker generates a dependency graph of its stages based on the datasets they read and write.

4.2 Data Communication

Canary provides two communication abstractions. The first is for data transfer. It is a scatter-gather operation, similar to MapReduce, Spark, and other frameworks. The operation takes two functions. The first, the scatter function, takes one or more datasets and produces one or more datasets as output. The key property of the scatter operation is that, while all of its input datasets must have the same partitioning and all of its output datasets must have the same partitioning, the input partitioning does not have to be the same as the output partitioning. Furthermore, unlike computational tasks, which only access one partition of a dataset when they execute, a scatter function accesses any partition (hence "scatter"). When the scatter completes, the worker uses the partition map to determine where it should send any non-empty scatter partitions. The gather function takes many partitions (the many pieces generated by different scatterers) and merges them into one.

Because a partition map can be stale, it is possible that a worker executing a scatter task will try to deliver intermediate results to the wrong worker. It is also possible for a worker to start receiving data for a partition it does not yet know it is responsible for. To handle these inconsistencies, each entry in the partition map has a version number associated with it. When the controller sends a partition map update, it increments the version number of any changed entries. Whenever a worker sends data to another worker, it includes the version number in its associated partition map entry. If a worker receives a data transfer for a partition it does not have and the sender has a newer version of the partition map entry, it concludes

that it will soon own the partition and so holds the scatter data. If the sender has an older version of the partition map entry, the receiver forwards the data to the correct destination and assumes the sender will be updated soon.

The second communication abstraction is for control. It is a reliable broadcast of a replicated boolean variable called a *signal*. Signal functions are used for conditions and loops. A signal function takes a dataset as input. This dataset must have a single partition (any other partitioning is a runtime error). It produces a boolean value as output. After the worker which has the input partition executes the function, it broadcasts the result to every other worker in the system. Other workers, when they reach a signal function, wait to hear the broadcast.

4.3 Spawning Local Tasks

The metadata that workers maintain is sufficient for them to quickly infer and spawn tasks locally. As a worker keeps track of the last stage to access each of its local partitions, it can, from its dependency graph, determine what are the next stages that will access it. For example, in Figure 3, the worker knows that the next stage to access `local_gradient` will be the `scatterG` function. Whenever the worker updates which stage last accessed a partition, it checks each of the next task(s) to access that dataset for whether all of their inputs are ready. If so, it spawns the task and adds it to a local run queue.

The key property of Canary's execution model that makes this possible is its communication primitives. Because any looping construct goes through a signal function, this acts as a barrier across the workers. Similarly, any results which escape a worker and can affect computations elsewhere must go through a scatter-gather, and so are a barrier. As a result, all of the workers agree on which iteration of a loop they are on, and correspondingly the position in the sequential program can be defined precisely enough to ensure that workers remain in sync with one another.

4.4 Partition Migration

Section 3.3 described how a controller tells two workers to migrate one or more data partitions. Because the controller does not know exactly where the workers are in their execution, it cannot tightly coordinate this exchange. Canary makes the assumption that a slight delay in migrating the partition is acceptable, since the worst that can happen is the job will run an additional iteration with a load imbalance.

When a worker receives a command to migrate a partition, it marks that partition with a flag. When the worker considers whether it can spawn a task, it always considers a flagged partition as not ready, and so will not spawn any new tasks that access that partition. Once there are no tasks that access the partition in the run queue and the last task to access it completes, the source worker begins the transfer to the destination. It includes, as metadata, the last stage to write to the partition. This tells the destination at what stage in the program the source worker is and so what the next tasks to execute on the partition are.

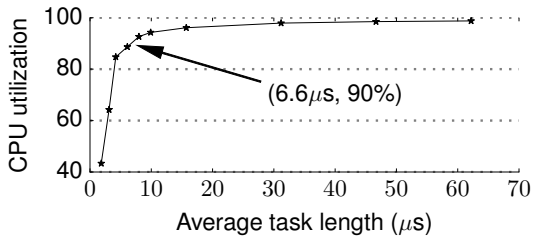


Figure 4: Canary can schedule 136K tasks/sec per-core while consuming less than 10% of available CPU time. This allows a worker with 18 cores to schedule 2.4M tasks/sec. In the benchmark, tasks run on average for 6.6μs and take 0.7μs to schedule. In comparison, a Spark controller has a peak scheduling throughput of 2.6K tasks/sec.

5 Evaluation

This section evaluates Canary by measuring its task scheduling throughput and how high scheduler throughput improves end-to-end application performance. It compares Canary's performance with GraphX and Spark, two analytics frameworks used heavily in practice today.

5.1 Implementation and Methodology

The Canary implementation this section evaluates is a from-scratch analytics computing engine written in C++. The entire framework is 4K semicolons. We have implemented numerous applications in the framework, including both the three benchmarks used in the evaluation as well as several scientific computing applications that we do not mention due to space constraints. Figure 1 shows its overall software architecture and data structures.

All experiments use Amazon EC2 [1] `C4` instances, a server configuration optimized for compute-intensive workloads. `C4` instances use Intel Xeon E5-2666 v3 (Haswell) processors with hyperthreading and 10GbE networking. Workers run on `c4.8xlarge` instances, containing 18 cores and 60GB of memory, while the controller runs on a `c4.large` instance, which has 1 core and 3.75GB of memory.

To evaluate application performance using Canary in comparison to systems that schedule centrally, we use three standard analytics benchmark applications: logistic

Figure 5: Canary scheduling throughput grows almost linearly with the number of workers, reaching 127M tasks/sec with 64 workers (1,152 cores). The high throughput allows jobs to be split into as many tasks as needed without concern for the scheduler overhead.

regression, k-means clustering and PageRank. For logistic regression and k-means clustering we compare Canary against hand-optimized Spark implementations (the low level API row in Table 1). For PageRank, we compare Canary against GraphX, an API built on top of Spark that is optimized for graph applications. Both the GraphX and Canary PageRank implementations use the same vertex cut partitioning algorithm, which uses a 2D partitioning of the sparse edge adjacency matrix and guarantees that each vertex has at most $2 \times \sqrt{numPartitions}$ mirror vertices.

We tuned each application's input data size so that Spark or GraphX uses about 80% of worker memory when running on a cluster with 8 workers. For logistic regression and k-means clustering, this means the training data is 1 billion vectors of 20 64-bit floating point numbers (160GB on disk). For PageRank, the graph has 5 million vertices and 635 million edges generated using the same graph parameters used to evaluate Pregel [22].

For Canary we report the iteration time as the *average* of all iteration times except the first and last iterations. For Spark and GraphX we instead use the *median* of all iterations to filter out the overhead of the JVM warm up, just-in-time compilation, and garbage collection. The median therefore more accurately reports the execution time of only the tasks themselves. Because Spark and GraphX have a long tail in their iteration times, using the average for Canary and the median for Spark and GraphX means the reported speedups are lower than what would be seen in production. Logistic regression and k-means clustering run for 50 iterations, while PageRank runs for 20 iterations.

5.2 Task Scheduling Throughput

We first measure the scheduling throughput of Canary by running a micro-benchmark that runs one billion stages. Each stage consists of tasks that busy loop on their data partition. The job assigns 10 data partitions to each core. Because each Canary worker core runs its own scheduler, having that scheduler completely utilize the core would prevent any forward progress on the job. We therefore measure scheduling throughput as the task rate that a core can schedule while using at most 10% of the core's CPU cycles, leaving $> 90\%$ of CPU cycles for application code.

Figure 4 shows the results. Canary takes 700ns to schedule a task. Therefore, when requiring the scheduler use at most 10% of CPU cycles, Canary can support tasks as short as 6.6μs. This allows a single worker core to schedule 136,000 tasks per second, and one 18-core worker node to schedule 2.4 million tasks per second. For 1ms tasks, the Canary scheduler uses only 0.07% of CPU cycles, leaving 99.93% for application code.

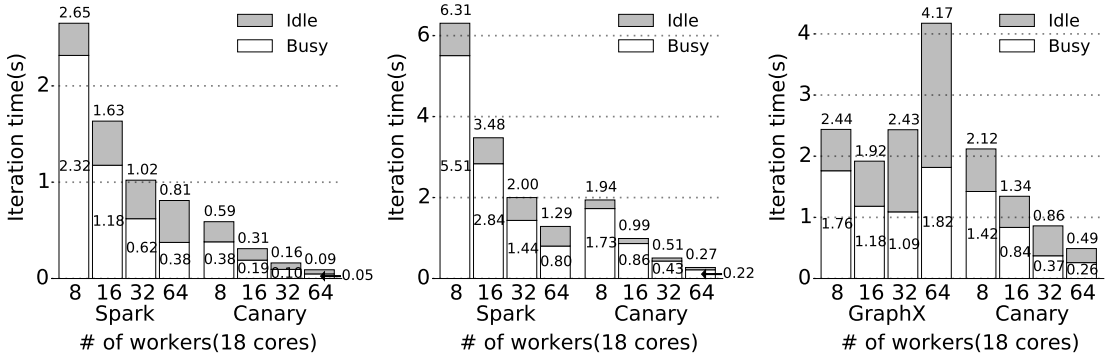
Canary is able to schedule tasks so quickly because all of the scheduling logic executes on not only the same node, but even the same process as application code. Therefore, generating, scheduling, and executing tasks requires neither network I/O nor even inter-process communication. Scheduling involves traversing the task data structure for runnable tasks and inserting them into a queue, while dispatch removes the next task from the queue.

Figure 5 shows the same micro-benchmark running across multiple workers. Canary's scheduling throughput scales linearly with the number of workers for up to 64 workers (1,152 cores). Task scheduling is completely local and requires no communication with the controller or other workers. With 64 workers, Canary can schedule 127 million tasks per second while using at most 10% of CPU cycles. This 127 million is less than the scheduling throughput of 64 independent workers, which would be 154 million (2.4 million per worker from Figure 4 times 64 workers). The difference comes from the difference in task execution time on the workers. When running on multiple nodes, the application sends data over the network for scatter-gather operations to compute whether to execute another iteration.

5.3 Application Performance

Running CPU-bound applications on more cores should reduce completion time until the task durations shrink enough that I/O time becomes significant. To evaluate the performance benefit of scaling jobs out to run on more cores, we examine the performance of Spark, GraphX and Canary as a job with a fixed input data size running on an increasing number of worker nodes. As noted in Section 5.1, the input sizes are such that they can fit in RAM on 8 workers.

Figure 6 shows the results for running the jobs on 8, 16, 32, and 64 workers (144, 288, 576 and 1152 cores). At low scale-outs, e.g., with 8 workers, Canary runs logistic



(a) Logistic regression. **(b) K-means clustering.** **(c) PageRank.**

Figure 6: Iteration execution time while keeping the size of the input fixed as the number of workers increases. Each core runs one task in a stage. Canary scales out logistic regression and k-means clustering linearly and accelerates PageRank up to 64 workers. Spark scales out worse due to scheduler load as well as data transfer.

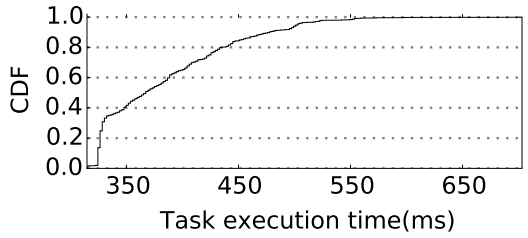
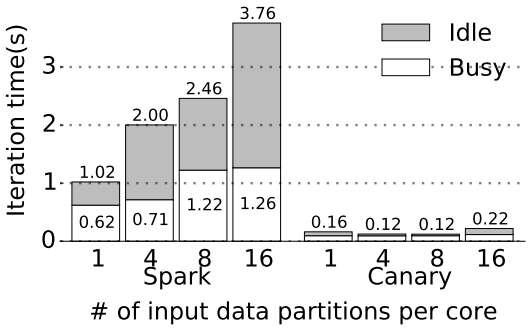
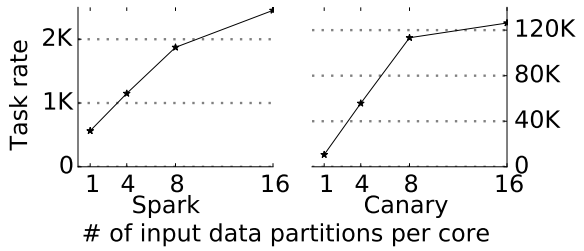


Figure 7: Execution time distribution for calculation tasks when Canary runs logistic regression using one input data partition per core on 8 18-core workers. The slowest task takes twice as long as the average task length despite processing the same amount of data.



(a) Iteration execution time. Spark slows down with more partitions per core due to scheduler load as well as data transfers. Canary speeds up due to better load balancing, up until 16 partitions/core when data I/O becomes a bottleneck.



(b) Task rates. Spark saturates at 2.6K tasks/sec, bottlenecking the job. Canary schedules up to 120K tasks/second, and bottlenecks at 16 partitions/core on application communication.

Figure 8: Logistic regression on 32 workers as the number of partitions per core increases.

regression 4.5× faster, k-means clustering 3.25× faster, and PageRank 15% faster. As the application scales out and runs faster, the performance improvements increase: logistic regression runs 9× faster, k-means cluster runs 4.8× faster, and PageRank runs 8.5× faster.

For Spark, this trend is because although Spark's computation time shrinks similarly to Canary's, the idle time shrinks much less quickly. Spark's idle CPU time is caused by task scheduling latency. At the start of an iteration, workers wait for the controller to dispatch tasks. In the case of 64 workers and 1,152 cores, the controller takes about 0.5sec to schedule enough tasks to keep all cores busy, given the controller can only schedule 2.6K tasks/sec.

The 5-10× speedup of Canary over Spark shows how fast analytics combined with scaling out demands high task scheduling throughput. When running on 64 workers, Canary executes 36,000 tasks/sec on average for logistic regression and 13,000 tasks/sec for k-means clustering.

For PageRank, Canary runs only 15% faster than

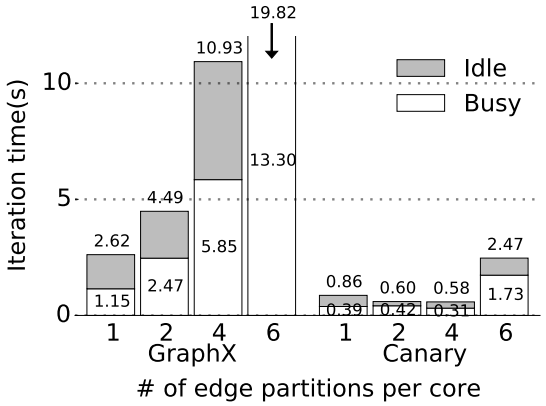


Figure 9: PageRank execution on 32 workers as partitions per core increases. GraphX is unable to handle additional partitions per core due to scheduling and data transfer bottlenecks. Canary speeds up by 32% due to better load balancing and because smaller partitions have better cache behavior.

GraphX on 8 workers; GraphX is an optimized system whose task execution speeds are close to C++ code. PageRank is a harder application to scale out because as the number of partitions goes up, the number of graph vertices that must be communicated increases (the vertex-cut algorithm creates more mirror vertices) and so I/O becomes an increasing portion of execution time. GraphX runs fastest with 16 workers, which have a total task rate of 450 tasks/sec. Canary, in contrast, continues to run faster as it scales out. With 64 workers, Canary runs 3.9× faster than GraphX with 16 workers, and 8.5× as fast as GraphX with 64 workers. At this speed, Canary executes 12,000 tasks/sec.

5.4 Micropartitioning

The experimental results in Figure 6 shows that Canary does not fully utilize its cores. Logistic regression and k-means cores spend 18-44% of their time idle despite having very little network I/O. Task execution time variance causes this CPU idle time. These experiments have one partition per core, and so slowest task determines execution time. Figure 7 shows the CDF of task execution time for logistic regression. The fastest core is twice as fast as the slowest one. This variance is caused by non-uniform memory access (NUMA). Each worker has two sockets, and on-socket memory throughput is twice that of off-socket memory. Because Linux's allocation policy maps physical pages across the two sockets, task execution time can vary by a factor of 2: the fastest tasks have all local pages and the slowest have all remote pages.

The variance in task execution time suggests that micropartitioning a stage and load balancing across cores will reduce completion time as this variance will average out. Figure 8a shows how long it takes to complete one iteration of logistic regression on 32 workers with micropartitioning.

Spark slows down with more micropartitions. Figure 8b shows the task rates for Spark and Canary in this experiment. At 16 partitions/core, the Spark controller is at its maximum scheduling capacity of 2,500 tasks/second. The controller occupies a larger and larger fraction of execution time, as cores fall idle. With 16 partitions per core, CPUs are idle 67% of the time as they wait to receive tasks to execute. In contrast, Canary schedules 120,000 tasks/second and the scheduler is not the bottleneck.

When there is one task per core (576 tasks), Canary runs over six times faster than Spark, and micropartitioning improves Canary's performance. Running with 16 partitions per core slows down the application as the amount of communication needed for the scatter/gather stage begins to dominate, but 4 or 8 partitions per core cut job completion time by 25%.

Micropartitioning has an additional benefit: smaller partitions can be transferred faster and pipelined with execution. For example, $P = 8$, each partition is approximately 135MB, takes 80ms to transfer, and a k-means computation takes 50ms. A slow worker can transfer partitions while it computes on others, overlapping these operations and reducing idle time.

Finally, micropartitions also improve performance through better cache locality, especially for workloads that are not simple scans of data, such as PageRank. Figure 9 shows how increasing the number of partitions per core affects completion time for GraphX and Canary. For GraphX, micropartitioning quickly overwhelms the scheduler and forces it to use inefficient marshalling procedures for larger amounts of data, such that its completion time goes up nearly linearly. For Canary, PageRank's performance is dominated by the step that scans over a list of edges and fills mirror vertex values into a hash table. Micropartitioning shrinks this hash table and reduces cache pressure, cutting completion time by 33% with 4 partitions per core. At 6 partitions per core, performance begins to degrade as the number of mirror vertices increases and so requires much more I/O and data replication.

6 Related Work

Scheduling jobs to share a cluster is a broad research topic, with prior work examining how to mitigate stragglers [34, 30], how to optimize networking use [3, 11, 17], or how to isolate performance between tenants [21].

This paper focuses on improving scheduling throughput. Earlier work like YARN [33] and Mesos [15] takes a centralized approach, and suffers from scalability challenges when serving large-scale clusters. To increase

scheduling throughput, distributed scheduling systems [28, 32, 6, 19] use multiple servers to schedule jobs in parallel by coordinating how these distributed servers allocate shared resources. Recent work [10, 12] proposes a hybrid approach, e.g., Hawk [10] schedules long-running batch processing jobs centrally to improve scheduling decisions, but uses distributed scheduling for short jobs to improve scheduling throughput. All of these prior approaches, however, assume that any single job is scheduled by a single controller. In contrast, Canary examines the case when even a single job's scheduling load is too great for a central controller.

State-of-art data analytics frameworks are designed without the awareness that a job ever needs such a high task rate as this paper explains, and their system designs perform poorly when significantly increasing the task rate within a job. Dryad [16, 35] and Spark [36] use a centralized controller to schedule tasks in a same job, and the controller can become a performance bottleneck. GraphX [14] is built on top of Spark and shares the same problem. Naiad [24] does not have the controller bottleneck, but uses a distributed progress tracking mechanism that decides when a notification can be delivered to a vertex to trigger its execution. The mechanism is required to execute the programming model (timely dataflow) correctly, and incurs significant overhead when the task rate is high enough due to more progress updates being exchanged.

The idea that the controller places data and then workers generate tasks locally is similar with how graph analytics frameworks, e.g. GraphLab [20], PowerGraph [13], and Galois [25], schedule distributed graph computations. These systems place vertices and edges on workers, and run a vertex program on each vertex, which performs computation, exchanges data between other vertex programs, and synchronizes through global data structures. Both the execution model and the scheduling design are tightly coupled with iterative graph algorithms, and cannot extend to general analytics workloads. For example, GraphLab [20] and PowerGraph [13] use distributed locking to schedule distributed graph computations, and lock neighboring vertices when the state of a vertex is being accessed, so as to avoid conflicting data access.

The idea of giving each worker a driver program copy such that workers understand application workflow is similar to the SIMD (single instruction, multiple data) programming paradigm, such as used in MPI (message passing interface). [2] MPI is widely used in supercomputing to scale scientific computations to tens of thousands of cores. In MPI, execution is described as processes running the same driver program as a continuous workflow. Those processes coordinate execution through exchanging messages, and block when waiting for messages. MPI's lack of a central controller or any scheduling at all means it scales out very well, but at the cost of poor load balancing and fault tolerance. Unlike MPI, Canary is an example of a MTC (many-task computing) system [29], which breaks execution into tasks that do not have to run in sequential order (as long as they respect data dependencies). Charm++ [18] and Legion [5] are other MTC systems, designed for supercomputers. Canary represents a midpoint in this design space, borrowing its distributed control flow from supercomputing but its centralized load balancing from cloud computing.

7 Discussion and Conclusion

In the past few years, led by systems such as Spark, GraphX, Galois, and Naiad, cloud computing systems have transitioned from being disk and I/O-centric to being CPU-bound computations on in-memory data. High performance computing (HPC) has spent decades exploring how to run massively parallel CPU-bound workloads on in-memory data, but has done so on supercomputers, which have different performance and reliability tradeoffs than cloud computing systems. For example, while floating point workloads dominate HPC, cloud systems have a mix of floating point and fixed point computations.

Canary is a first attempt to borrow key techniques from each field. From high performance computing it borrows distributed scheduling to support fast codes, while it borrows central control for fault tolerance from cloud computing. A Canary controller is responsible for deciding how data partitions are distributed across workers. Each worker core locally spawns and schedules tasks to execute based what partitions it has. Evaluations show that Canary can schedule up to 2.4M tasks/sec per worker, and the task scheduling throughput increases linearly up to 64 workers, with 1,152 cores scheduling 120M tasks/sec. This scheduling performance allows Canary to run high performance codes that are up to 90 times faster than standard Spark implementations and 9 times faster than highly optimized Spark implementations. High performance scheduling also allows Canary to further reduce job completion time by micropartitioning datasets.

Spark showed it was possible to achieve orders of magnitude improvements over disk-centric frameworks such as Hadoop. Canary shows that further orders of magnitude improvements are possible, but doing so requires new architectures and further research.

References

[1] Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2.

[2] Open MPI: Open Source High Performance Computing. http://www.open-mpi.org.

[3] AHMAD, F., CHAKRADHAR, S. T., RAGHUNATHAN, A., AND VIJAYKUMAR, T. N. ShuffleWatcher: Shuffle-aware Scheduling

in Multi-tenant MapReduce Clusters. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (2014), USENIX ATC'14, USENIX Association, pp. 1–12.

[4] ANANTHANARAYANAN, G., KANDULA, S., GREENBERG, A., STOICA, I., LU, Y., SAHA, B., AND HARRIS, E. Reining in the Outliers in Map-reduce Clusters Using Mantri. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (2010), OSDI'10, USENIX Association, pp. 1–16.

[5] BAUER, M., TREICHLER, S., SLAUGHTER, E., AND AIKEN, A. Legion: Expressing Locality and Independence with Logical Regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2012), SC '12, IEEE Computer Society Press, pp. 66:1–66:11.

[6] BOUTIN, E., EKANAYAKE, J., LIN, W., SHI, B., ZHOU, J., QIAN, Z., WU, M., AND ZHOU, L. Apollo: Scalable and Coordinated Scheduling for Cloud-scale Computing. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (2014), OSDI'14, USENIX Association, pp. 285–300.

[7] CHEN, Y., ALSPAUGH, S., AND KATZ, R. Interactive Analytical Processing in Big Data Systems: A Cross-industry Study of MapReduce Workloads. *Proc. VLDB Endow. 5*, 12 (Aug. 2012), 1802–1813.

[8] DEAN, J. Building Software Systems at Google and Lessons Learned, Slide 99. http://research.google.com/people/jeff/Stanford-DL-Nov-2010.pdf.

[9] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6* (2004), OSDI'04, USENIX Association, pp. 10–10.

[10] DELGADO, P., DINU, F., KERMARREC, A.-M., AND ZWAENEPOEL, W. Hawk: Hybrid Datacenter Scheduling. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference* (2015), USENIX ATC '15, USENIX Association, pp. 499–510.

[11] GANDHI, R., HU, Y. C., KOH, C.-K., LIU, H., AND ZHANG, M. Rubik: Unlocking the Power of Locality and End-point Flexibility in Cloud Scale Load Balancing. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference* (2015), USENIX ATC '15, USENIX Association, pp. 473–484.

[12] GODER, A., SPIRIDONOV, A., AND WANG, Y. Bistro: Scheduling Data-parallel Jobs Against Live Production Systems. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference* (2015), USENIX ATC '15, USENIX Association, pp. 459–471.

[13] GONZALEZ, J. E., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (2012), OSDI'12, USENIX Association, pp. 17–30.

[14] GONZALEZ, J. E., XIN, R. S., DAVE, A., CRANKSHAW, D., FRANKLIN, M. J., AND STOICA, I. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (2014), OSDI'14, USENIX Association, pp. 599–613.

[15] HINDMAN, B., KONWINSKI, A., ZAHARIA, M., GHODSI, A., JOSEPH, A. D., KATZ, R., SHENKER, S., AND STOICA, I. 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* (2011), NSDI'11, USENIX Association, pp. 295–308.

[16] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (2007), EuroSys '07, ACM, pp. 59–72.

[17] JALAPARTI, V., BODIK, P., MENACHE, I., RAO, S., MAKARYCHEV, K., AND CAESAR, M. Network-Aware Scheduling for Data-Parallel Jobs: Plan When You Can. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (2015), SIGCOMM '15, ACM, pp. 407–420.

[18] KALE, L. V., AND KRISHNAN, S. *CHARM++: A Portable Concurrent Object Oriented System Based on C++*, vol. 28. ACM, 1993.

[19] KARANASOS, K., RAO, S., CURINO, C., DOUGLAS, C., CHALIPARAMBIL, K., FUMAROLA, G. M., HEDDAYA, S., RAMAKRISHNAN, R., AND SAKALANAGA, S. Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference* (2015), USENIX ATC '15, USENIX Association, pp. 485–497.

[20] LOW, Y., BICKSON, D., GONZALEZ, J., GUESTRIN, C., KYROLA, A., AND HELLERSTEIN, J. M. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proc. VLDB Endow. 5*, 8 (Apr. 2012), 716–727.

[21] MACE, J., BODIK, P., FONSECA, R., AND MUSUVATHI, M. Retro: Targeted Resource Management in Multi-tenant Distributed Systems. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (2015), NSDI'15, USENIX Association, pp. 589–603.

[22] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: A System for Large-scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (2010), SIGMOD '10, ACM, pp. 135–146.

[23] MCSHERRY, F., ISARD, M., AND MURRAY, D. G. Scalability! But at What Cost? In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems* (2015), HOTOS'15, USENIX Association, pp. 14–14.

[24] MURRAY, D. G., MCSHERRY, F., ISAACS, R., ISARD, M., BARHAM, P., AND ABADI, M. Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), SOSP '13, ACM, pp. 439–455.

[25] NGUYEN, D., LENHARTH, A., AND PINGALI, K. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), SOSP '13, ACM, pp. 456–471.

[26] OUSTERHOUT, K., PANDA, A., ROSEN, J., VENKATARAMAN, S., XIN, R., RATNASAMY, S., SHENKER, S., AND STOICA, I. The Case for Tiny Tasks in Compute Clusters. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems* (2013), HotOS'13, USENIX Association, pp. 14–14.

[27] OUSTERHOUT, K., RASTI, R., RATNASAMY, S., SHENKER, S., AND CHUN, B.-G. Making Sense of Performance in Data Analytics Frameworks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (2015), NSDI'15, USENIX Association, pp. 293–307.

[28] OUSTERHOUT, K., WENDELL, P., ZAHARIA, M., AND STOICA, I. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), SOSP '13, ACM, pp. 69–84.

[29] RAICU, I., FOSTER, I., AND ZHAO, Y. Many-task Computing for Grids and Supercomputers. In *Many-Task Computing on Grids and Supercomputers, 2008. MTAGS 2008. Workshop on* (Nov 2008), pp. 1–11.

[30] REN, X., ANANTHANARAYANAN, G., WIERMAN, A., AND YU, M. Hopper: Decentralized Speculation-aware Cluster Scheduling at Scale. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (2015), SIGCOMM '15, ACM, pp. 379–392.

[31] SANCHO, J., BARKER, K., KERBYSON, D., AND DAVIS, K. Quantifying the Potential Benefit of Overlapping Communication and Computation in Large-Scale Scientific Applications. In *SC 2006 Conference, Proceedings of the ACM/IEEE* (Nov 2006), pp. 17–17.

[32] SCHWARZKOPF, M., KONWINSKI, A., ABD-EL-MALEK, M., AND WILKES, J. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), EuroSys '13, ACM, pp. 351–364.

[33] VAVILAPALLI, V. K., MURTHY, A. C., DOUGLAS, C., AGARWAL, S., KONAR, M., EVANS, R., GRAVES, T., LOWE, J., SHAH, H., SETH, S., SAHA, B., CURINO, C., O'MALLEY, O., RADIA, S., REED, B., AND BALDESCHWIELER, E. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing* (2013), SOCC '13, ACM, pp. 5:1–5:16.

[34] XU, Y., MUSGRAVE, Z., NOBLE, B., AND BAILEY, M. Bobtail: Avoiding Long Tails in the Cloud. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (2013), nsdi'13, USENIX Association, pp. 329–342.

[35] YU, Y., ISARD, M., FETTERLY, D., BUDIU, M., ERLINGSSON, U., GUNDA, P. K., AND CURREY, J. DryadLINQ: A System for General-purpose Distributed Data-parallel Computing Using a High-level Language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (2008), OSDI'08, USENIX Association, pp. 1–14.

[36] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. 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* (2012), NSDI'12, USENIX Association, pp. 2–2.