Decoupling the Control Plane from Program Control Flow for Flexibility and Performance in Cloud Computing

Hang Qu
Stanford University
Stanford, California
quhang@stanford.edu

Omid Mashayekhi
Stanford University
Stanford, California
omidmsk@gmail.com

Chinmayee Shah
Stanford University
Stanford, California
chshah@stanford.edu

Philip Levis
Stanford University
Stanford, California
pal@cs.stanford.edu

ABSTRACT
Existing cloud computing control planes do not scale to more than a few hundred cores, while frameworks without control planes scale but take seconds to reschedule a job. We propose an asynchronous control plane for cloud computing systems, in which a central controller can dynamically reschedule jobs but worker nodes never block on communication with the controller. By decoupling control plane traffic from program control flow in this way, an asynchronous control plane can scale to run millions of computations per second while being able to reschedule computations within milliseconds.

We show that an asynchronous control plane can match the scalability and performance of TensorFlow and MPI-based programs while rescheduling individual tasks in milliseconds. Scheduling an individual task takes 1 µs, such that a 1,152 core cluster can schedule over 120 million tasks/second and this scales linearly with the number of cores. The ability to schedule huge numbers of tasks allows jobs to be divided into very large numbers of tiny tasks, whose improved load balancing can speed up computations 2.1-2.3x.

CCS CONCEPTS
• Computer systems organization → Cloud computing; • Computing methodologies → Parallel algorithms;

KEYWORDS
Cloud Framework Control Planes, Distributed Scheduling, Centralized Control

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ACM Reference Format:
ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3190508.3190516

Figure 1: The execution time for running one thousand 1ms tasks on each core increases as the Nimbus controller drives more cores, because the controller synchronously spawns tasks on workers.

1 INTRODUCTION
Cloud frameworks make programming easier by taking care of many of the complexities of distributed programming, such as scheduling computations, load balancing, and recovering from failures. In this paper, we refer to the systems and protocols for these services as a framework’s control plane.

A typical cloud framework control plane that uses a fully centralized design can dispatch fewer than 10,000 tasks per second [24, 32]: jobs can either be big (thousands of cores) or fast (tasks take tens of milliseconds) but not both [24, 37]. Furthermore, parallelizing a job across more cores splits it into smaller pieces, reducing task duration. This tradeoff introduces a fundamental limit on how much these systems can parallelize a job. Increasing parallelization has many benefits, including improved fault tolerance, load balancing, and faster performance [17, 27, 29, 30, 43]. In practice, however, increasing parallelism past even a few hundred cores quickly hits the control plane’s scaling limit, making jobs run slower [24, 32].

A recent proposal to improve control plane scalability, taken by Nimbus [24] and Drizzle [40], is to distribute the control plane across worker nodes. Distributed control planes can handle an order of magnitude higher load, scheduling up to 250,000 tasks per second. This higher task rate allows them to run jobs on more cores, running jobs 5-8 times faster. This improvement, however, is only a constant factor and does not remove the fundamental limit. Figure 1 shows how this limit manifests: job completion time increases under Nimbus as a job grows to run on more cores. Although the per-worker CPU time remains fixed, at 1 second, completion time grows...
from 1.6 to 3.7 seconds as the control plane is unable to issue more than 250,000 tasks/second.

A second approach is to remove the control plane entirely. Systems such as TensorFlow [6] and Naiad [26] use a dataflow model that implicitly triggers computations when data arrives, while systems such as MPI [4] have programmers explicitly place data transfer calls in their code. By imposing no control overhead, these frameworks’ scalability is limited only by their applications as with classical high-performance computing workloads. Rebalancing load or migrating tasks, however, requires effectively killing and restarting a computation by generating a new execution plan and installing it on every node. This takes tens of seconds, stalling computations for hundreds of iterations and precluding interactive jobs. A cluster manager, such as Borg [41], cannot allocate or deallocate resources from jobs in these frameworks dynamically as other jobs appear and complete. For example, if a high priority, low-latency job arrives in a cluster, the cluster manager cannot pull cores from existing jobs without stopping them for seconds.

Recent technology trends have made this tension between scalability and flexibility increasingly acute. Increases in RAM allowed analytics to transition from disk-based systems [2, 10] to in-memory processing [26, 43]. Freed from slow I/O, the performance of applications in these frameworks can be CPU-bound [31], motivating aggressive code optimizations [5, 33]. By executing tasks faster, these improvements further increase the load on a control plane; all of them report results from a single machine.

This paper proposes a new control plane design that breaks the existing tradeoff between scalability and flexibility. It allows jobs to run extremely short tasks (<1ms) on thousands of cores and reschedule computations in milliseconds. The ability to run short tasks on thousands of cores is valuable in computations that iterate until they converge, such as machine learning and scientific computing. Removing the control plane bottleneck allows these computations to scale out while still supporting fault tolerance and dynamic load balancing.

The key insight of the design is that the bottleneck in existing control planes is due to synchronous operations between workers and a controller. Existing control planes are tightly integrated with the control flow of their programs, requiring nodes to block on communication with a central controller node at certain points in the program, such as spawning new tasks or resolving data dependencies. As programs run faster and on more cores, the latency of these synchronous operations increases, causing worker CPUs to fall idle.

This paper proposes using an asynchronous control plane. With an asynchronous control plane, a central controller tells worker nodes how to redistribute work, but workers locally decide, between themselves, when to redistribute. An asynchronous control plane can reschedule jobs, adapt to changes in load, add workers, and remove workers just as quickly and flexibly as a synchronous one. But when a job is stably running on a fixed set of workers, an asynchronous control plane exchanges only occasional heartbeat messages to monitor worker status. The control plane’s traffic is completely decoupled from the control flow of the program, so running a program faster does not increase load at the controller.

Current synchronous control planes such as Spark execute 8,000 tasks per second; distributed ones such as Nimbus and Drizzle can execute 250,000 tasks/second. Canary, a framework we have designed with an asynchronous control plane, can execute over 100,000 tasks/second on each core, and this scales linearly with the number of cores. Experimental results on 1,152 cores show it schedules 120 million tasks per second. Jobs using an asynchronous control plane can run up to an order of magnitude faster than on prior systems. At the same time, the ability to split computations into huge numbers of tiny tasks with introducing substantial overhead allows an asynchronous control plane to efficiently balance load at runtime, achieving a 2–3× speedup over highly optimized MPI codes.

This paper makes four research contributions:

- the concept of an asynchronous control plane, which has the scheduling flexibility of a centralized controller yet scales as well as a dataflow system;
- the program abstraction of task recipes, which allow worker nodes to independently spawn and schedule their own tasks as well as atomically migrate computations;
- the data abstraction of a partition map, which allows a central controller to decide how to distribute computations while maximizing data locality; and
- experimental evaluations of an asynchronous control plane that show it can match the performance and scalability of frameworks with no control plane, such as TensorFlow or MPI, while simultaneously matching the runtime flexibility of centralized controller designs, redistributing tasks within a job 1,000 times faster than TensorFlow.

## 2 CLOUD FRAMEWORK CONTROL PLANES

A cloud framework control plane splits a job into dependent computation units of tasks that can be scheduled to different nodes, and assigns those tasks to resources allocated by a cluster manager. Worker nodes that run on separate servers execute the tasks, while one controller node coordinates those workers and communicates with the cluster manager. Note that different jobs can run on the same worker. The responsibility of the controller varies in different control plane designs. For example, a Spark [43] controller is involved in every task scheduling, while a MPI [4] controller only launches processes on workers at the start of a job. Mesos [16] and YARN [39] are two example cluster managers that use this architecture.

Current cloud framework control planes do not scale to run jobs with short tasks on many cores [24, 32, 37, 40]. The source of this scalability bottleneck is their use of synchronous operations between the controller and workers as illustrated in Figure 2. That is, at certain points during job execution, workers block on communication with the controller, waiting for the controller’s control messages, and falling idle if the controller fails to send those messages. As controller load goes up, the latency of the synchronous operation increases, causing workers to block idle for longer periods. These synchronous operations are because the control plane and the program control flow are tightly coupled. A control plane obtains its control over a job by synchronizing with program control flow, injecting synchronous operations between the spawning and assignment of tasks.
A centralized control plane, as used in MapReduce [10], Dryad [17], Spark [43] and CIEL [27], requires one synchronous operation for every task (Figure 2a). A central controller node spawns all tasks in a job. A worker waits for the controller to send it new tasks, executes new tasks when requested, and falls idle while it waits. The effect of this synchronous operation is especially pronounced during an application barrier, when every worker waits for the controller to synchronously dispatch a new batch of tasks. This centralization limits the number of tasks per second the system can execute, as it is limited by the rate at which the controller can send commands to workers. The Spark controller, for example, dispatches fewer than 10,000 tasks per second [24, 32].

Ray [29] optimizes this centralized design by enabling a worker to spawn and execute some tasks locally. If a task accesses data on other workers, however, it must still perform synchronous operations with a controller.

Another approach to deal with this scalability bottleneck, used by systems such as Nimbus [24] and Drizzle [40] is to distribute a control plane across the workers and controller (Figure 2b). The key insight in these systems is to dispatch large blocks of tasks with a single synchronous operation. This reduces the number of synchronous operations, allowing the systems to scale better, but only by a constant factor. Workers still synchronously receive computations from the controller and fall idle if the controller cannot keep up.

TensorFlow [6], Naiad [26] and MPI [4] take this approach to its extreme, dispatching the entire job to workers all at once with a single synchronous operation that is required when the job starts (Figure 2c). These systems impose no control plane overhead at runtime, but are locked to a static execution distribution decided at the start of a job. Redistributing job execution, if possible, requires stopping all the nodes, regenerating the execution plans, reinstalling them on the nodes, and restarting from a checkpoint. Any changes in the task schedule (for the purpose of straggler mitigation or load balancing) are either very expensive, or impossible.

A long history of systems research has shown that asynchronous I/O, if managed carefully, can remove bottlenecks such as these [9, 28, 42]. Applying this principle to a cloud computing control plane means that a central controller still controls where tasks should execute, but these decisions may be applied asynchronously. Furthermore, to ensure that workers do not fall idle, they independently spawn their own tasks, asynchronously with respect to the controller. Achieving an asynchronous design such as this requires two mechanisms: a program representation that allows workers to spawn computations without any central coordination, and a scheduling mechanism in which workers can correctly execute computations despite having potentially inconsistent views of how computations are distributed.

2.1 Making the Control Plane Asynchronous

Applying the above principles, we argue that an asynchronous control plane places four requirements on its program representation and scheduling model:

- **Distributed scheduling**: workers must spawn and execute tasks locally, requiring no communication with a central controller.
- **Atomic migration**: the output of a job must have each task execute exactly once, despite mid-job migrations.
- **Centralized control**: a centralized controller must be able to dynamically change how a job is distributed across workers without introducing any synchronous operation.
- **Maximized locality**: a controller must be able to very efficiently calculate data distributions, so that workers exchange minimal data as they execute.

The key challenge in satisfying these four requirements is the absence of synchronous operations. Without synchronous operations, the controller and workers cannot precisely know exactly where in the program control flow another worker is. Despite this, the control plane must be able to perform the above four functions.

3 ASYNCHRONOUS CONTROL PLANE ARCHITECTURE

The proposed control plane requires no synchronous operations between the controller and workers because they have cleanly divided responsibilities: a controller decides where to execute tasks and workers decide when to execute them.

On the worker side, an abstraction called task recipes, described in Section 4, describes when to run a task by specifying a pattern matched against the task’s input data. Using recipes, every worker spawns and executes tasks by examining the state of its local data objects. This requires no interaction with the controller. Task recipes do not move or fetch data: they trigger based on what data is present on the worker.
Task recipes implement distributed scheduling, as workers can spawn and execute tasks purely based on their local state. Ensuring atomic migration, however, requires a careful design of how preconditions are encoded as well as how data objects move between workers. No node in an asynchronous control plane has a global view of the execution state of a job, so workers manage atomic migration among themselves. Two workers can be in very different positions in the control flow of the program, yet must ensure that data objects transferred between them neither miss nor repeat computations: a worker who is ahead, must run older recipes on objects it receives, while a worker who is behind must be sure not to re-run its current recipes on objects it receives.

This section explains task recipes and the mechanisms by which workers using recipes can atomically migrate computations. It uses a logistic regression computation, shown in Figure 3, as a running example. The next section presents the partition map, the mechanism by which an asynchronous control plane implements centralized scheduling policies that maximize locality.

### 4.1 Execution Model

Task recipes assume an execution model typical to cloud frameworks. A job is broken up into many *stages*. Each stage typically represents a function (or series of functions) applied over one or more variables representing large, distributed *datasets*. Datasets are mutable and can be updated in place, avoiding the overhead of copying on each write required by systems such as Spark [43]. Each dataset can be broken up into many *partitions*. The number of partitions in a dataset defines the available degree of parallelism, i.e. the number of individual computational *tasks* a stage can be broken into. A *driver program* specifies a sequential program order, but the runtime may reorder tasks as long as the observed result is the same as the program order (just as how processors reorder instructions).

In addition to this standard execution model, task recipes add an additional mechanism: tasks can send and receive *messages*. Messages represent an intermediate result that is written by one task and read by another: once a message is read, there is no need to track it. A partition, in contrast, is shared data that can be read and modified by many tasks. The placement of partitions determines how tasks are assigned to workers, and its management by the control plane implicitly schedules a job. Messages, on the other hand, are dataflow between tasks and so implicitly define execution ordering. Furthermore, this dataflow can be between tasks potentially running on different workers, so can represent distributed dependencies.
4.2 Abstraction
A task recipe specifies three things:

- a function to run,
- which datasets the function reads and/or writes, and
- preconditions that must be met for the function to run.

Figure 3 shows an example of a logistic regression job and how it is represented with task recipes. The job has three datasets: the gradient G and weights W, which are not partitioned, and the training data TD, which is partitioned into TD1 and TD2.

A key property of task recipes is that if a dataset has N partitions, then a recipe accessing that dataset will trigger N times, once for each partition. For example, the Compute recipe accesses the training data TD, which has two partitions. This recipe therefore triggers twice, as shown in Figure 3a, once on the worker holding TD1 and once on the worker holding TD2. Task recipes enforce the same partitioning as other frameworks, that all datasets that a stage accesses must have the same number of partitions.

4.3 Preconditions
Datasets determine where a recipe triggers. Preconditions determine when it triggers. There are three types of preconditions:

- **Last input writer:** For each partition it reads or writes, the recipe specifies which recipe should have last written it. This enforces local write-read dependencies, so that a recipe always sees the correct version of its inputs.
- **Output readers:** For each partition it writes, the recipe specifies which recipes should have read it since the last write. This ensures that a partition is not overwritten until tasks have finished reading the old data.
- **Read messages:** The recipe specifies how many messages a recipe should read before it is ready to run. Unlike the other two preconditions, which specify local dependencies between tasks that run on the same worker, messages specify remote dependencies between tasks that can run on different workers.

Incorrect preconditions can lead to extremely hard to debug computational errors, so they are generated automatically from a sequential user program. We defer code examples to Section 6, but Figure 3 shows a block diagram of a logistic regression computation and the corresponding recipes with their preconditions.

A single recipe describes potentially many iterations of the same data-parallel computation. The loop shown in Figure 3a, for example, executes many times. Because the recipes in Figure 3b refer to prior executions of the loop, they cannot be used for the first iteration. There are therefore a parallel set of recipes (not shown) for the first iteration of a loop, with slightly different preconditions. For example, the Broadcast recipe expects W to be written by stage SN, which is the preceding input stage shown in Figure 7.

4.4 Exactly-once Execution and Asynchrony
When a data partition moves from one worker to another, they must exchange sufficient information to ensure that the task from a given stage executes exactly once and messages are delivered correctly. For example, suppose that the worker holding the gradient G in Figure 3 is running slowly, and so G is moved to another worker after it has received G1 and G2 but before Reduce has run. These messages must be re-routed to the new holder of G so it can trigger the Reduce recipe. However, if G is moved after Reduce has run, then the messages should not be re-routed and Reduce should not run again.

To ensure that tasks execute exactly once, when workers transfer a data partition they include the access history metadata relevant to preconditions, the last writer and how many recipes have read it. The last writer ensures both that a modifying recipe does not re-execute and that it is not missed, since other recipes cannot yet trigger. This approach depends on the stage number being consistent across all workers, so they can precisely specify stages across the entire cluster.

When a worker is notified that it should migrate a partition, it must first wait for all currently executing tasks that access the partition to complete, so that its last writer and number of readers are in a consistent state. It marks the partition as busy, so that no new recipes are triggered from it. Then, when all outstanding tasks complete, it begins the transfer. It can delay setting the busy flag, to defer transmission, but generally will trigger at the first opportunity.

The asynchrony between control plane commands to migrate data and the actual migration allows workers to distributedly schedule tasks without ever blocking on a central controller.

Exactly-once execution also depends on the correct delivery of messages. We defer a discussion of this mechanism to Section 5.4, which describes the control plane.

5 PARTITION MAP
A partition map describes how a job should be distributed across workers, and is used as the mechanism for the controller to signal workers how to reschedule job execution. This section defines a partition map (Section 5.1), describes the control interface between the controller and workers (Section 5.2), discusses how a partition map relates to data locality (Section 5.3) message delivery (Section 5.4), and illustrates how to use a partition map to express scheduling algorithms (Section 5.5).
worker receives all updates in the same order but not necessarily at the same time. Consequently the partition map copy on a particular worker can be stale. The controller attaches an increasing version number to each update, so that workers can tell which worker’s partition map copy is more stale.

5.3 Maximize Data Locality

There are two requirements to ensure that the data distribution given by a partition map maximizes data locality. First, every partition should have a single physical copy. Otherwise, every write to a partition needs to be synchronized to all its copies, causing poor data locality. Second, the case when the input partitions to the same task are distributed on multiple workers should be eliminated, because executing the task requires at least one of the partitions be migrated.

The controller satisfies the second requirement by always updating a partition map under the constraints that the input partitions to each possible task in a job are assigned to the same worker. The execution model of task recipes is intentionally designed to make the constraints explicit and achievable: if a stage reads or writes multiple datasets, a task in the stage only reads or writes the datasets’ partitions that have the same index, so those partitions are constrained to be assigned to the same worker.

Figure 5 gives an example job with four stages and five datasets. Stage W only accesses Dataset 1, and runs independent tasks on each partition of Dataset 1. So Stage W imposes no constraints on the partition map. Stage X accesses both Dataset 2 and Dataset 3, so partition (2, j) and partition (3, j) will be input to the jth task of Stage X and should be assigned to the same worker.

The controller computes the constraints by analyzing the dataset access of all stages written in the driver program. The analysis divides datasets into sets, by putting two datasets into the same set if any stage takes both datasets as input, and concludes that partitions should be assigned to the same worker when they have the same partition index, and their datasets end up in the same set.

5.4 Message Delivery

A strawman solution for message delivery is that a worker decides where to send a message by looking up which worker stores an
This subsection illustrates how to express scheduling algorithms using the partition map as the mechanism. The controller runs a scheduling function to generate partition map updates that describe how to reschedule job execution. Listing 1 gives the scheduling function signature. Common scheduling functions, e.g. for straggler mitigation and dynamic worker membership, are given by the framework, but a user can associate customized scheduling functions to a job. A scheduling function is optionally invoked before a job starts (to generate the initial partition map), after a worker is added, before a worker is removed, at a fixed time interval or when requested by a user.

Listing 2 gives a scheduling function for straggler mitigation. It calls `findStragglerWorker` to determine a straggler worker, and then `getNextWorker` to move the partition of the straggler to the most idle worker. Listing 3 shows a scheduling function that migrates one partition from a dying worker to other workers in round robin.

5.5 Expressing Scheduling Algorithms

This subsection illustrates how to express scheduling algorithms using the partition map as the mechanism. The controller runs a scheduling function to generate partition map updates that describe how to reschedule job execution. Listing 1 gives the scheduling function signature. Common scheduling functions, e.g. for straggler mitigation and dynamic worker membership, are given by the framework, but a user can associate customized scheduling functions to a job. A scheduling function is optionally invoked before a job starts (to generate the initial partition map), after a worker is added, before a worker is removed, at a fixed time interval or when requested by a user.

Listing 2 gives a scheduling function for straggler mitigation. It calls `findStragglerWorker` to determine a straggler worker, which could be implemented in many ways, e.g. if the worker’s CPU utilization is much higher than average, or if the background processes on the worker are using significant CPU cycles. Note that

Updates computePartitionMapUpdates(
    WorkerTable worker_table,
    PartitionTable partition_table,
    Set<WorkerId> dying_workers);

Listing 1: The signature of a scheduling function that generates partition map updates. It takes three inputs: a worker table that describes each worker’s resource usage, a partition table that specifies which worker stores each partition and whether the partition is being migrated, and a set of workers that are being shut down by the cluster manager.

Updates updates;
if (!haveStragglerWorker(worker_table))
    return updates;
WorkerId straggler = findStragglerWorker(worker_table);
PartitionEntry entry = selectOnePartition(
    worker_table[straggler].partitions);
worker = getMostIdleWorker(worker_table);
updates.add(entry.dataset_id,
    entry.partition_index,
    worker);
return updates;

Listing 2: A scheduling function that migrates one partition on a straggler worker to the most idle worker.

Updates updates;
WorkerId worker = getFirstWorker(worker_table);
for (entry in partition_table)
    if (!entry.in_migration &&
        entry.worker_id in dying_workers) {
        updates.add(entry.dataset_id,
            entry.partition_index,
            worker);
        worker = getNextWorker(worker_table, worker);
    }
return updates;

Listing 3: A scheduling function that migrates partitions from dying_workers to other workers in round robin.

Figure 6: Workers ensures correct message delivery when not all partition map copies on workers are up-to-date. TD1 is a dataset partition. When some worker has a stale partition map copy (i.e. N2 in Figure (a) and N3 in Figure (b)), workers use the version number of partition map updates to decide whether to hold or forward received messages.

input partition of the receiving task. While a program sends a message to a task, the system translates the destination to be one of the receiving task’s input partitions. The constraints of a partition map mean the input partitions to a particular task are always stored on the same worker, so every input partition is equivalent.

This strawman solution can deliver a message to the wrong worker, if the lookup used a stale partition map copy. Figure 6 illustrates how to ensure correct message delivery using the version number of partition map updates. When a worker sends a message, it includes the version number of the latest partition map update it receives in the message. If a worker receives a message that has a newer version than the version number the worker has, it holds the message. If the message has an older version, the receiver forwards the message to the correct destination. This simple algorithm ensures that the message will eventually arrive at the worker that holds the destination partition. Another corner case is that after a worker receives a message, later partition map updates may indicate the message should be sent to another worker. So a worker buffers messages it receives, and checks whether buffered messages should be sent again whenever receiving a partition map update.

Lost messages are detected by per-hop acknowledgements. If an acknowledgement is not received within a time period, this is interpreted as a worker failure. Because datasets are in-memory and mutable, there is no replica and a job cannot proceed if a worker fails. A worker detecting a failure informs the controller, which restarts the application from the last checkpoint.

(a) A worker holds a message if the message has a newer version number than the receiver.

(b) A worker forwards a message if the receiver has a newer version number than the message. When forwarding a message, the worker rewrites the version number in the message.
Again?  

Update  

Reduce  

Input  

Broadcast  

Compute  

Reduce  

Update  

Output  

Figure 7: The sequential program for the example logistic regression computation, which is constructed by the driver program (omitted). The computation loops until the Again stage computes False. Every rectangle is a data-parallel computation, i.e. a stage. Inside the rectangle is the stage’s function, which retrieves a partition by calling GetPartition, gets the partition index by calling PartitionIndex, and optionally passes messages to tasks in the next stage.

6 IMPLEMENTATION

We implement recipes and the asynchronous control plane in an in-memory cloud framework called Canary. This section describes Canary’s programming abstraction and how Canary achieves fault tolerance.

6.1 Programming Abstraction

Writing recipes and their preconditions manually is tedious and error-prone. Instead, Canary asks a user to write a driver program that constructs another sequential program of data-parallel computations (i.e. stages) with optional loops or branches, and then translates the sequential program to recipes. Figure 7 shows the sequential program for the example logistic regression computation, which is constructed by the user’s driver program. The remaining text describes features of the programming abstraction.

Data-dependent control flow. The loop in Figure 7 is iteratively executed until the Again stage computes False. The Again stage should have one task and compute one boolean value, so that there is a unique boolean decision for each iteration. The driver program checks this when constructing the Again stage, by ensuring every dataset the stage reads or writes has only one partition.

Distributed driver program execution. Before a job starts, every worker receives a driver program copy from the controller, and independently runs the copy to construct the sequential program of stages and translate it to recipes. The driver program is not running during job execution, so it must describe all computations before the job starts. In contrast, a centralized driver program as in Spark, CIEL and Ray causes centralized performance bottleneck. The benefit of a centralized driver program is that it can run arbitrary user codes during job execution to decide what stages to execute, and is more flexible for describing highly irregular computations as in reinforcement learning.

Weak data model. Canary makes minimal assumption of what is stored in a dataset. A dataset partition can store any C++ object that can be serialized, and the class type of the object is chosen by the user. The weak data model enables working on geometric data that can be serialized, and the class type of the object is chosen by the user. The weak data model enables working on geometric data and reusing existing data structures, which is handy for scientific computing applications.

Message passing. Canary’s data model does not specify what operations a dataset provides, so a driver program has to specify, e.g. how to repartition a dataset into another using a partitioning function, or how to reduce multiple writes to a dataset partition.

To this end, Canary supports passing messages between tasks for implementing repartitioning, reduction and broadcasting. Concurrent writes to the same dataset partition are modeled as passing multiple messages (representing the writes) to a task, and the task decides how to reduce the messages and write to the partition. In Figure 7, every task in the Compute stage generates a local gradient (Gi). Instead of writing the local gradients to partition G, the task passes Gi as a message. The Reduce task receives all Gi messages, sums them up, and writes it to partition G. The reduction logic is highly reusable and is implemented as an application library. For simplicity, a stage’s function only sends messages to tasks in the next stage and the receiving task is identified by the index of its input partitions. The receiving task’s function calls Recv to retrieve messages in a vector ordered by the indexes of the sending tasks.

6.2 Fault Tolerance

Canary periodically checkpoints all the partitions of a job. The controller monitors whether workers fail using periodic heartbeat messages. If any worker running a job is down, the controller cleans up the job’s execution on all workers, and reruns the job from the last checkpoint. To build a checkpoint, the controller first pauses all workers which report back the furthest stage the worker has run, then instructs all workers to complete all stages before that stage, and finally asks workers to store all partitions in the job to a persistent storage. This forms a consistent checkpoint at the barrier stage. The barrier stage introduces additional idle time, but simplifies debugging because only tasks before the barrier stage have executed.

Checkpoint-based failure recovery renews the execution on every worker back to the last checkpoint when a failure happens, while lineage-based failure recovery as in Spark only needs to recompute lost partitions. But the cost of lineage-based failure
Decoupling the Control Plane from Program Control Flow

Figure 8: Iterations per second of a logistic regression computation over 100GB of data as it is parallelized across more cores in Spark, Nimbus, TensorFlow and Canary. On 1,024 cores, every task takes 2.6ms. Canary and TensorFlow scale almost linearly, while Nimbus and Spark bottleneck at their controllers.

Figure 9: How many 1-millisecond tasks Spark, Nimbus, TensorFlow and Canary can run as the number of cores increases. Both Spark and Nimbus bottleneck at their controller, while TensorFlow’s and Canary’s continue to scale up to 1,024 cores.

recovery in CPU-intensive jobs outweighs the benefit, because it requires every partition to be copied before modifying it.

7 EVALUATION

We evaluate the benefits of an asynchronous control plane by answering four questions:

(1) Can an asynchronous control plane scale to parallelize computations as highly as dataflow systems that have no control plane?
(2) What limits the maximum task rate an asynchronous control plane can support?
(3) Can the ability to run large numbers of tiny tasks reduce job completion time?
(4) Can an asynchronous control plane reschedule tasks as quickly as a synchronous one?

To answer these questions, we implemented an asynchronous control plane in Canary, an in-memory cloud framework written in C++. We compare Canary with a full sample of the existing design points in cloud frameworks. Spark release 2.0 is an example of a centralized control plane, Nimbus is an example of a distributed control plane, and Open MPI release 1.10 as well as TensorFlow release 1.0 are examples of no control plane.

In most experiments, both the controller and workers use Amazon EC2 [1] c4.4xlarge instance, a server configuration optimized for compute-intensive workloads. The instance has an Intel Xeon E5-2666 processor with 8 cores and 30GB of memory, with 10GbE links. The scientific computing applications use the larger c4.8xlarge instances for each worker, which has 18 cores and 60GB of memory.

In summary, we find that:

• Canary scales as well as TensorFlow, linearly scaling its performance in experiments using up to 1,024 cores. Cost considerations prevented us from evaluating at larger scales. Canary can also match the performance of MPI implementations of three data analytics benchmarks.
• The task rate is limited by the CPU cycles on each worker. Scheduling a task takes approximately 1µs. Capping the scheduling overhead at 10%, an individual core can scale to 100,000 tasks/second. This rate scales linearly with the number of cores: 1,152 cores can schedule 120 million tasks/second.
• The ability to split a job into many tasks per core allows Canary to easily balance non-uniform load. This allows Canary to run several scientific computing benchmarks 2.1-2.3× faster than MPI.
• An asynchronous control plane can reschedule a job in milliseconds, on par with a synchronous one and three orders of magnitude faster than TensorFlow, which has to recompile and reinstall the dataflows on all workers.

7.1 Control Plane Scalability

We evaluate the scalability of an asynchronous control plane, comparing it with the existing design points of Spark, Nimbus, and TensorFlow. The scalability of cloud framework jobs depends not only on the control plane but also CPU efficiency [25]: if the framework generates very slow code, then cores can execute few tasks per second and the control plane does not become the bottleneck. To isolate differences in generated code performance from the performance of their control planes, we use the same methodology as prior work, measuring the task duration of each framework and then replacing application tasks with spin loops that run for the same duration as the fastest framework.

To evaluate the performance effect an asynchronous control plane has on end-to-end application performance, we compare the performance of three data analytics benchmarks implemented in Canary and in MPI. As MPI has no control plane and uses highly optimized C code, it represents the best-case performance for these benchmarks.

7.1.1 Strong Scaling. Figure 8 shows the application performance of a fixed size logistic regression job as it is parallelized across more cores, measured in iterations per second. This experiment evaluates the strong scaling of Spark, Nimbus, TensorFlow,
and Canary. The job processes 100GB of data, which is split into 10 partitions per core.

Performance for both Canary and TensorFlow scales linearly up to 1,024 cores. Nimbus cannot scale much beyond 512 cores, which causes its maximum performance to be 50% slower than Canary and TensorFlow on 1,024 cores. Spark’s performance cannot scale much beyond 256 cores, and at 512 cores it begins to decline, which aligns with the results reported by Nimbus [24]: TensorFlow and Canary execute over 6 times faster.

7.1.2 Weak Scaling. To determine whether the scalability results in Figure 8 are due to the control plane or other application aspects of these frameworks, we measured control plane performance with a weak scaling test. In this experiment, every task runs for 1 millisecond. We measure how many tasks a framework can execute per second by scaling the number of cores. Ideally, each core can execute 1,000 tasks per second, and the overall task rate should be 1,000 tasks/second.

Figure 9 plots the task rate of each framework as the number of cores increases. Canary and TensorFlow scale linearly. Canary and TensorFlow are able to achieve 80% of the ideal performance, executing 800 tasks/second per core. This 20% performance loss is due to application-level communication in the logistic regression job (reducing the gradient and broadcasting the weights). The synchronous control plane communication in Spark and Nimbus limit them to 8,000 and 250,000 tasks/second, while Canary and TensorFlow easily scale up to 750,000 tasks/second.

7.1.3 End-to-end Application Performance. Because it places all functionality in the control of a programmer, carefully optimized and designed MPI code represents the best-case performance for distributed computing. To evaluate the performance of applications written in Canary, we compare three data analytics benchmarks written in Canary with their MPI counterparts. Both use the same C++ task implementations. The MPI implementations use MPI’s optimized communication libraries for reduction and broadcasts, using asynchronous communication primitives (e.g. MPI_Isend, MPI_Irecv) wherever possible.

We use three data analytics benchmarks: logistic regression, k-means clustering and PageRank. Logistic regression is shown in Figure 3. K-means clustering has a higher computation density than logistic regression. PageRank uses a standard vertex-cut algorithm.

The input data sizes of the benchmarks are chosen to fill the RAM of 8 nodes: 240GB of training data in logistic regression and K-means clustering and a 1.6 million edge sparse graph for PageRank.

Figure 10 compares the iteration times of running the benchmarks on Canary and MPI with 256-1024 cores. Canary matches the performance of MPI. The error bars show the minimum and maximum across 3 runs with 100 iterations per run. A node has 8 cores.

7.1.4 What is the Limit? To explore the scalability limit of the asynchronous control plane, we measure how fast Canary executes independent empty tasks. We cap the fraction of CPU time that can be spent on scheduling to 10%; in this setting, applications would still receive 90% of the CPU cycles. Figure 11 shows the results. Each individual task takes 1μs to schedule. Using 10% of CPU cycles, each worker core can execute 100,000 tasks/second. Since scheduling requires no inter-node communication, this scales linearly with the number of cores. Running on 1,152 cores, Canary can execute more than 120 million tasks per second.

Figure 11: The task rate of Canary grows linearly with the number of cores, reaching more than 120 million tasks per second on 1,152 cores, because every core spawns and executes tasks in parallel.
7.2 Benefits of Tiny Tasks

We evaluate whether the ability to split a job into tiny tasks can improve application performance and examine two complex scientific computing applications. Unlike the data analytics benchmarks, these scientific computing applications have imbalanced computation load on different partitions, so splitting tasks finer improves load balancing between cores. Note that MPI’s process-based execution model performs poorly when there are more processes than cores, and so cannot subdivide jobs finer than the number of cores. OpenMP-MPI hybrid [35] might achieve similar speedup but requires manually parallelizing computations.

The fluidanimate benchmark in PARSEC [7] is a particle simulation, which simulates 600 million particles in a Cartesian grid of 1100 × 380 × 1100. It has 4 stages per iteration. Lassen [3] tracks a wave moving on an unstructured mesh of 5 billion cells, and has 7 stages per iteration. When porting these applications to Canary, we reuse their existing data structures, but have to break execution into stages, and rewrite the communication logic. These experiments use c4.8xlarge instances with 18 cores per node, because a larger instance helps balance load between cores. We choose the input data sizes so that up to 80% worker memory is used for each worker.

As shown in Table 1, splitting tasks four times finer enables Canary to reduce PARSEC’s iteration time by 2.30× and Lassen’s iteration time by 2.14×. The speedup is due to higher CPU utilization rather than faster computation because both MPI and Canary execute the same computation codes. In the experiment, load balancing happens between cores on the same node, so no inter-node communication is needed.

7.3 Rescheduling Job Execution

We measure how fast Canary reschedules job execution to mitigate stragglers and dynamically use more or fewer nodes, using TensorFlow as a reference.

Straggler mitigation. Figure 12 shows the first use case of straggler mitigation. We manually introduce a straggler node by launching one computation-intensive thread on each of its cores, which will slow down execution by roughly 50%. A Canary controller detects a straggler node if the node’s background processes use significant CPU cycles, and then migrates partitions away from the worker and evenly distributes the partitions to other workers. TensorFlow cannot automatically mitigate stragglers, and its rescheduling is achieved by manually configuring it to rerun the job without using the straggler node. This experiment assumes partitions are replicated on workers, which is a common storage policy in the cloud, so the partition data on the straggler worker does not need to be transferred.

As shown in Figure 12, a straggler node increases the iteration time from 0.66s to 1.19s. After the straggler node is removed from the job, the iteration time drops to 0.73s. However, TensorFlow takes 5.6s to complete the rescheduling. As a larger cluster is used and stragglers happen more often, TensorFlow will lag behind.

Canary’s rescheduling overhead. We further investigate Canary’s overhead during the rescheduling by measuring how long Canary takes to migrate empty partitions between workers. When there are 128 workers and 8 partitions per worker, Canary takes 0.94ms to migrate one empty partition, and 2.60ms to migrate all 8 partitions of a worker to other workers. Since partitions store no data, the reported time mainly consists of two parts: (1) transmitting partition map updates from the controller to workers, and (2) exchanging the metadata of what recipes have read or modified the data, the reported time mainly consists of two parts: (1) transmitting partition map updates from the controller to workers, and (2) exchanging the metadata of what recipes have read or modified the data, the reported time mainly consists of two parts: (1) transmitting partition map updates from the controller to workers, and (2) exchanging the metadata of what recipes have read or modified the data, the reported time mainly consists of two parts: (1) transmitting partition map updates from the controller to workers, and (2) exchanging the metadata of what recipes have read or modified the data. This is the time cost for TensorFlow to reschedule a single task.

TensorFlow’s rescheduling overhead. We report how much time TensorFlow takes to compute and install dataflows on workers as shown in Figure 13. This is the time cost for TensorFlow to recompute and reinstall dataflows on workers.
update the variable. Regardless of the simple computation pattern, it takes 0.93s to compute the dataflows for 16 workers, and 0.58s to install the dataflows on those workers. The computation and installation time increases linearly with the number of workers. Note the job uses in-graph replication instead of between-graph replication, because the latter only works for a limited set of applications such as distributed machine learning based on a parameter server [21].

Using more or fewer nodes. Figure 14 shows how Canary redistributes job execution to dynamically use more or fewer nodes i.e. the number of worker nodes available to a k-means clustering job changes over time. The job initially runs on 32 workers, and takes 1.2s per iteration. Then, 32 workers are added to Canary’s allocation. As a result, half training data of 120GB are shipped to added workers in 15 seconds. Computation continues during the rescheduling, so both the iteration during the rescheduling and the next iteration are slower. After the rescheduling, the job runs twice as fast. A reverse procedure happens when the cluster size is reduced. Another command is issued to the controller to condense data back to 32 workers, after which the controller shuts down removed workers.

8 RELATED WORK

Distributed resource scheduling. Distributed scheduling systems [8, 11, 19, 32] solve the problem of allocating resources to tasks in parallel, and their design enables multiples nodes to decide the best resources for tasks without tight coupling. However, they do not deal with the dependencies of tasks in a job, since only a cloud framework knows how tasks exchange data and depend on each other. Therefore, those resource scheduling systems assume tasks in a job are always scheduled by the same node, but this paper suggests the task execution rate within a job can be too high for a single node to handle.

Graph processing frameworks. First, many graph processing frameworks (Pregel [23], GPS [36], and Mizan [20]) use the Bulk Synchronous Parallel (BSP) model [38], and rely on global barriers to ensure correct execution, and that is highly inefficient for scheduling dependent tasks that frequently exchange data.

Second, graph processing frameworks can avoid global barriers by introducing the asynchronous execution model, which allows vertices to run different numbers of supersteps without tight synchronization. For example, GraphLab [22] and PowerGraph [14] achieve this by allowing a vertex to pull data from others, and use distributed locking to resolve conflicted data access. Giraph Uncuffed [15] achieves this with a push-based model, and replaces global barriers by local barriers. Asynchronous execution accelerates graph algorithms, but does not generally work for other application domains.

Third, graph processing frameworks often rely on static graph partitioning algorithms to balance load, i.e. they assign computation to nodes statically. Mizan [20] migrates vertices between nodes during global barrier phases in BSP. In the asynchronous execution model, it is still unclear how to implement dynamic load balancing.

HPC frameworks. High performance computing frameworks can handle fine-grained computations efficiently, but their solutions involve manual effort to deal with correct execution, and only work when running on a fixed set of nodes offering uniform and stable performance. For example, MPI [4] runs the same program on distributed processes, coordinating execution through message passing. MPI scales well as there is no global state or central control, instead the heroic application developer has to implement all logic beyond communication channels. Charm++ [18] models an job as distributed objects that can trigger events that are executed on each other. A user has to reason about the event triggering order so as to guarantee correct execution.

Programming abstractions. Guarded command language describes non-deterministic programs by associating a predicate with each statement about when the statement can run. [12] Stateful dataflow graphs are an imperative abstraction where a task can access partitions and exchange messages. [13] The distributed memory model in Piccolo allows tasks to modify partitions in-place; task can modify any partition, and concurrent writes are resolved through accumulative operations. [34] Canary’s programming abstraction borrows ideas from each of these prior programming models, adapting them to implement a control plane for distributed task scheduling.

9 CONCLUSION

This paper demonstrates that a cloud computing framework can run tiny tasks as short as milliseconds on hundreds of cores, and at the same time reschedule job execution in milliseconds. The key insight is to remove the synchronous operations in the control plane, so adding runtime control capability to a cloud framework imposes negligible overhead. Implementing an asynchronous control plane requires significant changes to existing cloud framework design. First, a program abstraction, called task recipes, avoids a central program control flow, so that workers spawn and execute tasks without any interaction with the controller. Second, using a partition map as the scheduling mechanism, a controller retains centralized scheduling while workers can asynchronously apply the controller’s scheduling decisions.

ACKNOWLEDGMENTS

We thank our shepherd, Tim Harris, and the anonymous reviewers for their valuable feedback. We thank David Terei for contributing to an earlier draft of this paper. This work was funded by the