Execution Templates: Caching Control Plane Decisions for Strong Scaling of Data Analytics

Omid Mashayekhi, Hang Qu, Chinmayee Shah, and Philip Levis, *Stanford University*

https://www.usenix.org/conference/atc17/technical-sessions/presentation/mashayekhi

This paper is included in the Proceedings of the
2017 USENIX Annual Technical Conference (USENIX ATC '17).
July 12–14, 2017 • Santa Clara, CA, USA
ISBN 978-1-931971-38-6

Execution Templates: Caching Control Plane Decisions for Strong Scaling of Data Analytics

Omid Mashayekhi Hang Qu Chinmayee Shah Philip Levis
Stanford University

Abstract

Control planes of cloud frameworks trade off between scheduling granularity and performance. Centralized systems schedule at task granularity, but only schedule a few thousand tasks per second. Distributed systems schedule hundreds of thousands of tasks per second but changing the schedule is costly.

We present *execution templates*, a control plane abstraction that can schedule hundreds of thousands of tasks per second while supporting fine-grained, per-task scheduling decisions. Execution templates leverage a program's repetitive control flow to cache blocks of frequently-executed tasks. Executing a task in a template requires sending a single message. Large-scale scheduling changes install new templates, while small changes apply edits to existing templates.

Evaluations of execution templates in Nimbus, a data analytics framework, find that they provide the fine-grained scheduling flexibility of centralized control planes while matching the strong scaling of distributed ones. Execution templates support complex, real-world applications, such as a fluid simulation with a triply nested loop and data dependent branches.

1 Introduction

As data analytics have transitioned from file I/O [1, 9] to in-memory processing [26, 28, 42], systems have focused on optimizing CPU performance [30]. Spark 2.0, for example, reports 10x speedups over prior versions with new code generation layers [38]. Introducing data-parallel optimizations such as vectorization, branch flattening, and prediction can in some cases be faster than hand-written C [32, 41]. GPU-based computations [2, 3] improve performance further.

Computational speedups, however, demand a higher task throughput from the control plane. This creates a tension between task throughput and dynamic, fine-grained scheduling. Available systems cannot fulfill both

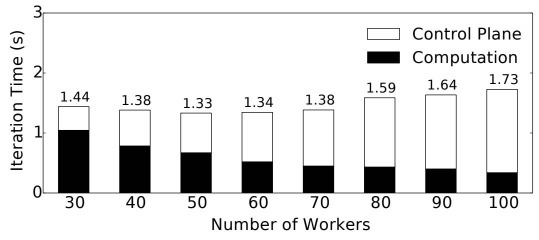


Figure 1: The control plane is a bottleneck in modern analytics workloads. Increasingly parallelizing logistic regression on 100GB of data with Spark 2.0's MLlib reduces computation time (black bars) but control overhead outstrip these gains, *increasing* completion time.

requirements simultaneously. Today, frameworks adopt one of two design points to schedule their computations across workers. One is a centralized controller model, and the other is a distributed data flow model.

In the first model, systems such as Spark [42] use a centralized control plane, with a single node that dispatches small computations to worker nodes. Centralization allows a framework to quickly reschedule, respond to faults, and mitigate stragglers *reactively*, but as CPU performance improves the control plane becomes a bottleneck. Figure 1 shows the performance of Spark 2.0's MLlib logistic regression running on 30–100 workers. While computation time decreases with more workers, these improvements do not reduce overall completion time. Spark spends more time in the control plane, spawning and scheduling computations. While there is a huge body of work for scheduling *multiple* jobs within a cluster [6, 10, 11, 19, 23, 31, 35], these approaches do not help when a *single* job has a higher task throughput than what the control plane can handle, as in Figure 1.

The second approach, used by systems such as Naiad [28] and TensorFlow [3], is to use a fully distributed control plane. When a job starts, these systems install data flow graphs on each node, which then independently execute and exchange data. By distributing the control plane and turning it into data flow, these frameworks achieve strong scalability at hundreds of thousands of tasks per second. However, data flow graphs describe a static schedule. Even small changes, such as rescheduling a task between two nodes, requires stopping the job, recompiling the flow graph and reinstalling it on every node. As a result, in practice, these systems mitigate stragglers only *proactively* by launching backup workers, which requires extra resource allocation even for non-straggling tasks [3].

This paper presents a new point in the design space, an abstraction called *execution templates*. Execution templates schedule at the same per-task granularity as centralized schedulers. They do so while imposing the same minimal control overhead as distributed execution plans.

Execution templates leverage the fact that long-running jobs (e.g. machine learning, graph processing) are repetitive, running the same computation many times [37]. Logically, a framework using execution templates centrally schedules at task granularity. As it generates and schedules tasks, however, the system caches its decisions and state in templates. The next time the job reaches the same point in the program, the system executes from the template rather than resend all of the tasks. Depending on how much system state has changed since the template was installed, a controller can immediately *instantiate* the template (i.e. execute without modification), *edit* the template by changing some of its tasks, or *install* a new version of template. Templates are not bound to a static control flow and support data-dependent branches; controllers *patch* system state dynamically at runtime if needed. We call this abstraction a template because it caches some information (e.g., dependencies) but instantiation requires parameters (e.g., task IDs).

Using execution templates, a centralized controller can generate and schedule hundreds of thousands of low-latency tasks per second. We have implemented execution templates in Nimbus, an analytics framework designed to support high performance computations. This paper makes five contributions:

1. Execution templates, a control plane abstraction that schedules high task throughput jobs at task granularity (Section 2).
2. A definition of the requirements execution templates place on a control plane and the design of Nimbus, a framework that meets these requirements (Section 3).
3. Details on how execution templates are implemented in Nimbus, including program analyses to generate and install efficient templates, validation and patching templates to meet their preconditions, and dynamic edits for in-place template changes (Section 4).

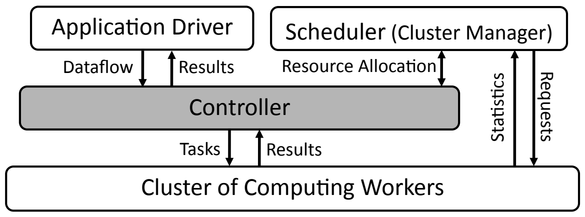


Figure 2: Generalized architecture of a cloud computing system: a *driver* program specifies the application logic to a *controller*, which can either directly assign tasks to workers or request resources from a *cluster manager*. Execution templates operate within a controller.

4. An evaluation of execution templates on analytics benchmarks, comparing them with Spark's fine-grained scheduler and Naiad's high-throughput data flow graphs (Section 5).
5. An evaluation of Nimbus running a PhysBAM [12] particle-levelset water simulation [13] with tasks as short as 100μs. (Section 5).[1]

This paper does not examine the question of scheduling policy, e.g., how to best place tasks on nodes, whether by min-cost flow computations [16, 21], packing [17, 18], or other algorithms [6, 19, 23, 33] (Section 6). Instead, it looks at the *mechanism*: how can a control plane support high throughput, fine-grained decisions? Section 7 discusses how execution templates can be integrated into existing systems and concludes.

2 Execution Templates

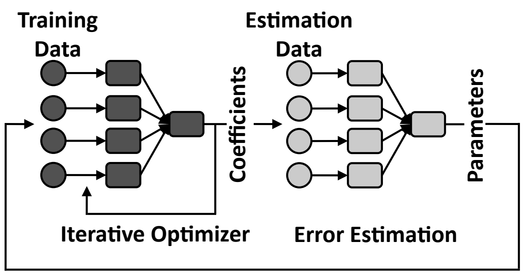
This section introduces execution templates and their characteristics. Figure 2 shows the general architecture of cloud computing systems. Execution templates operate on the controller and its interfaces.

Execution templates are motivated by the fact that long-running jobs are usually iterative and run same set of tasks repetitively [37] with minor changes. For example, Figure 3 shows the pseudocode and task graph for a training regression algorithm. The algorithm consists of a nested loop. The `Gradient` and `Estimate` operations can each generate many thousands of tasks. This graph structure is identical for each iteration, but the same vertex in two iterations can have different values across iterations, such as the `coeff` and `param` parameters. Furthermore, task identifiers change across iterations. With execution templates, the control plane can leverage the fixed structure to improve the performance.

[1]PhysBAM is an open-source simulation package that has received two Academy Awards and has been used in over 20 feature films.

```
while (error > threshold_e) {
   while (gradient > threshold_g) {
      // Optimization code block
      gradient = Gradient(tdata, coeff, param)
      coeff += gradient
   }
   // Estimation code block
   error = Estimate(edata, coeff, param)
   param = update_model(param, error)
}
```

(a) Driver program pseudocode.



(b) Iterative execution graph.

Figure 3: Task graph and driver program pseudocode of a training regression algorithm. It is iterative, with an outer loop for updating model parameters based on the estimation error, and an inner loop for optimizing the feature coefficients. The driver program has two basic blocks corresponding to inner and outer loops. `Gradient` and `Estimate` are both parallel operations that execute many tasks on partitions of data.

2.1 Abstraction

An execution template is a parameterizable list of tasks. The fixed structure of the template includes the list of tasks, their executable functions, task dependencies, relative ordering, and data access references. The parameter list includes the task identifiers and runtime parameters passed to each task.

To enable data dependent branches and nested loop structures, execution templates work at the granularity of basic blocks. A *basic block* is a code sequence in the driver program with only one entry point and no branches except the exit. For example, Figure 3 has two basic blocks, one for the inner loop and one for the outer loop operations. Note that loop unrolling and other batching techniques (e.g., as used in Drizzle [36]) cannot capture nested loops and data dependent branches.

Execution templates are *installed* and *instantiated* at run time. These two operations results in performance improvements in the control plane by caching and reusing repetitive control flow. Execution templates also support two special operations, *edits* and *patching*, which deal with scheduling changes and dynamic control flow. Each operation is discussed in the following subsections.

2.2 Installation and Instantiation

There are two types of execution templates, one for the driver-controller interface called a *controller template*, and one for the controller-worker interface called a *worker template*. Controller templates contain the complete list of tasks in a basic block across all of the worker nodes. They cache the results of creating tasks, dependency analysis, data lineage, bookkeeping for fault recovery, and assigning data partitions as task arguments. For every unique basic block, a driver program installs a controller template at the controller. The driver can then execute the same basic block again by telling the controller to instantiate the template.

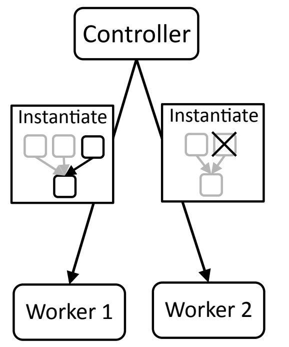
Where controller templates describes a basic block over the whole system, each worker template describes the portion of the basic block that runs on a particular worker. Workers cache the dependency information needed to execute the tasks and schedule them in the right order. Like TensorFlow [3], external dependencies such as data exchanges, reductions, or shuffles appear as tasks that complete when all data is transferred. Worker templates include metadata identifying where needed data objects in the system reside, so workers can directly exchange data and execute blocks of tasks without expensive controller lookups.

When a driver program instantiates a controller template, the controller makes a copy of the template and fills in all of the passed parameters. It then checks whether the prior assignment of tasks to workers matches existing worker templates. If so, it instantiates those templates on workers, passing the needed parameters. If the assignment has changed (e.g., due to scheduling away from a straggler or a failure), it either edits worker templates or installs new ones. In the steady state, when two iterations of a basic block run on the same set of n workers, the control plane sends $n+1$ messages: one from the driver to the controller and 1 from the controller to each of the n workers.

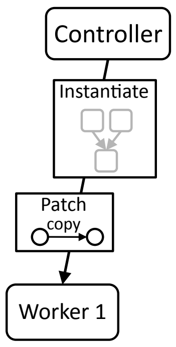
2.3 Edits

Execution templates have two mechanisms to make control plane overhead scale gracefully with the size of scheduling changes: installing new templates and editing existing ones. If the controller makes large changes to a worker's tasks, it can install a new worker template. Workers cache multiple worker templates, so a controller can move between several different schedules by invoking different sets of worker templates.

Edits allow a controller to change an existing worker template. Figure 4(a) shows how edits manifest in the



(a) Edit task from worker 2 to 1.



(b) Patching.

Figure 4: Edits and patches allow a framework to efficiently adapt templates to dynamic changes in the system. Edits dynamically modify a template structure in place, while patches move and copy data objects to match a template's preconditions. Grey denotes cached template information, while black denotes information sent over the network.

control plane: they modify already installed templates in place. Edits are used when the controller needs to make small changes to the schedule, e.g., migrate one of many partitions. Edits are included as metadata in a worker template instantiation message and modify its data structures. An edit can remove and add tasks. Edits keep the cost of dynamic scheduling proportional to the extent of changes. Together, installation and edits allow a controller to make fine-grained changes to how a basic block is distributed across workers.

2.4 Patching

Each worker template has a set of preconditions that must hold when the template is instantiated, for example requiring a replicated data object in local memory to have the most recent write. When a driver program instantiates a controller template, the system state may not meet the preconditions of the associated worker templates. This can happen because a basic block can be entered from many different points in the program. When a template is created, the controller may not even have seen all of these positions (e.g., an edge case covered by an if/else statement).

A controller uses patches to ensure correct execution in the presence of dynamic driver program control flow. Patches update and move data from one worker to another to satisfy the preconditions. For example, the worker templates for the inner loop in Figure 3(a) have the precondition that `param` needs to be in local memory. But there are two cases in which the controller might invoke the templates: the first iteration of the loop and subsequent iterations. In subsequent iterations, `param` is inductively already in local memory. However, on the first iteration, `param` exists only on the worker that calculated it. The controller therefore *patches* the inner loop template, sending directives to workers that copy `param` to each worker (Figure 4(b)).

Patches allow execution templates to efficiently handle dynamic program control flow. This is important when loop conditions are based on data, such as running until an error value falls below a threshold. There are two options to deal with the associated uncertainties in control flow. The controller can either ensure that the preconditions of every worker template always hold, or when a template is instantiated it can patch system state to match the preconditions. The first approach is prohibitively expensive, because it requires unnecessary and expensive data copies. For example, it would require immediately copying `param` in Figure 3(a) to every worker after it is calculated even if the outer loop terminates. A controller therefore has to react to the driver's stream of controller template instantiation requests and enforce the preconditions on the fly.

3 System Design

This section defines the requirements that execution templates place on a control plane and describes the design of a cloud computing framework, called Nimbus, that meets these requirements.

3.1 Control Plane Requirements

Conceptually, execution templates can be incorporated into any existing cloud framework. Incorporating them, however, assumes certain properties in the framework's control plane. We describe these requirements here, and defer a discussion of how they can be incorporated into existing systems to Section 7.

1. Workers maintain a queue of tasks and locally determine when tasks are runnable. Worker templates create many tasks on a worker, most of which are not immediately runnable because they depend on the output of prior tasks. A worker must be able to determine when these tasks are runnable without going through a central controller, which would become a bottleneck.

2. Workers can directly exchange data. Within a single template, one worker's output can be the input of tasks on other workers. As part of executing the template, the two workers need to exchange data without going through a central controller, which would become a bottleneck.

3. The controller schedules fine-grained tasks. Fine-grained tasks are a prerequisite to support fine-grained scheduling; they define the minimum scheduling change that a system can support.

3.2 Nimbus Architecture

Nimbus is an analytics framework that meets the three requirements. Nimbus's system architecture is designed to support execution templates and run computationally intensive jobs that operate on in-memory data across many nodes. Like Spark, Nimbus has a centralized controller node that receives tasks from a driver program. The controller dispatches these application tasks to workers. The controller is responsible for transforming tasks from a driver program into an execution plan, deciding on which workers to run which computations.

As it sends application tasks to workers, the controller inserts additional control tasks, such as tasks to copy data from one worker to another. These tasks explicitly name the workers involved in the transfer, such that workers can directly exchange data.

3.3 Nimbus Execution and Data Model

In Nimbus, a job is decomposed into *stages*. A stage is a computation over a set of input data and produces a set of output data. Each data set is partitioned into many *data objects* so that stages can be parallelized. Each stage typically executes many *tasks*, one per object, that operate in parallel. In addition to the identifiers specifying the data objects it accesses, each task can be passed parameters, such as model parameters or constants.

Nimbus tasks operate on mutable data objects. Supporting in-place modification of data avoids data copies and are crucial for computational and memory efficiency. In-place modification also has two crucial benefits for execution templates. First, multiple iterations of a loop access the same objects and reuse their identifiers. This means the data object identifiers can be cached in a template, rather than recomputed on each iteration. Second, mutable data objects reduce the overall number of objects in the system by a large constant factor, which improves lookup speeds.

Mutable objects mean there can be multiple copies and versions of an object in the system. For example, for the code in Figure 3(a), after the execution of the outer loop, there are *n* copies of `param`, one on each worker. However, one copy of `param`, has been written to, and has an updated value. Each data object in the system therefore combines an object identifier with a version number. The Nimbus controller ensures, through data copies, that tasks on a worker always read the latest value according to the program's control flow.

3.4 Nimbus Control Plane

The Nimbus control plane has four major commands. Data commands create and destroy data objects on workers. Copy commands copy data from one data object to another (either locally or over a network). File commands load and save data objects from durable storage. Finally, task commands tell the worker to execute an application function.

Commands have five fields: a unique identifier, a *read set* of data objects to read, a *write set* of data objects to write, a *before set* of the commands that must complete before this one can execute, and a binary blob of *parameters*. Task commands include a sixth field, which application function to execute.

A command's before set includes only other tasks on that worker. If there is a dependency on a remote command, this is encoded through a copy command. For example a task associated with the `update_model` operation in Figure 3(a) depends on the results of the parallel `Estimate` operation. The `update_model` task has *n* copy commands in its before set; one for each locally computed `error` in each partition.

Copy commands execute asynchronously and follow a push model. A sender starts transmitting an object as soon as the command's before set is satisfied. Because this uses asynchronous I/O it does not block a worker thread. Similarly, a worker asynchronously reads data into buffers as soon as it arrives. Once the before set of a task reading the data is satisfied, worker changes a pointer in the data object to point to the new buffer.

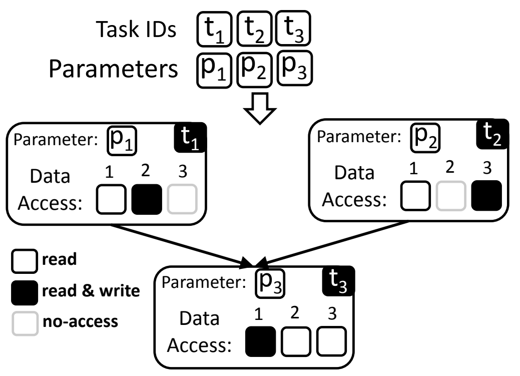
4 Implementation

This section describes how Nimbus implements execution templates and their operations. The Nimbus code, including execution templates, is publicly available at `https://github.com/omidm/nimbus`. Nimbus core library is about 35,000 semicolons of C++ code and supports tasks written in C++. In addition to machine learning and graph processing applications, the repository includes graphical simulations ported to Nimbus.

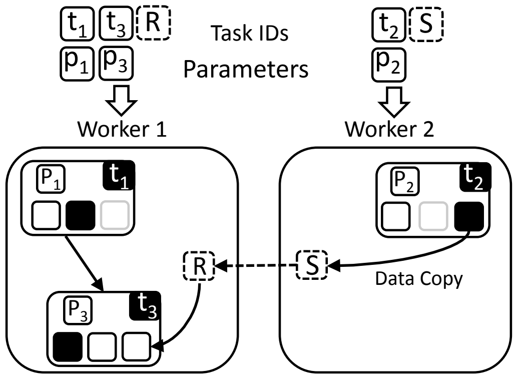
4.1 Installation and Instantiation

Template installation begins with the driver sending a start template message to the controller at the beginning of a basic block. In the current implementation, a programmer explicitly marks the basic block in the driver program. For example, in Figure 3(a), the only change in the driver program to support templates is extra annotations before and after basic blocks; one can also use other automatic approaches such as static program analysis. As the controller receives tasks, it simultaneously schedules them normally and stores them in a temporary task graph data structure.

At the end of the basic block, the driver sends a template finish message. On receiving a finish message, the controller takes the task graph and post-processes it into



(a) A **controller template** represents the common structure of a task graph metadata. It stores task dependencies and data access patterns. It is invoked by filling in task identifiers and parameters to each task.



(b) Each **worker template** stores the common structure of a task graph for execution including the data copies among workers. It is invoked by passing the task identifiers, and parameters to each task.

Figure 5: Controller template and worker templates for a simple task graph.

an optimized, table-based data structure. Pointers are turned into indexes for fast lookups into arrays of values.

Controller templates cache the read set, write set, and function identifier. A template instantiation message includes an array of command identifiers and a block of task parameters. Within a template, task identifiers index into this array. The one time cost of generating the ordered indices keeps the successive instantiations efficient. Figure 5(a) shows the instantiation of a controller template with new set of task identifiers and parameters.

Once it has generated the controller template, the controller generates the associated worker templates. Worker templates have two halves. The first half exists at the controller and represents the entire execution across all of the workers. This centralized half allows the controller to cache how the template's tasks are distributed across workers and track the preconditions for generating patches when needed.

Each worker template has preconditions that list which data objects at each worker must hold the latest update to that object. Not all data objects are required to be up to date: a data object might be used for writing intermediate data and be updated within the worker template itself. For example, in Figure 5(b), the third data object on worker 1 does not need to have the latest update at the beginning of the worker template; the data copy within the worker template updates it.

The second half of the worker template is distributed across the workers and caches the per-worker local command graph which they locally schedule. The controller installs worker templates very similarly to how the driver installs controller templates. And like controller templates, instantiation passes an array of task identifiers and parameters. Figure 5(b) shows a set of worker templates for controller template in Figure 5(a).

4.2 Patching

Before instantiating a worker template, controller must *validate* whether the template's preconditions hold and patch the worker's state if not. Validating and patching must be fast, because they are sequential control plane overhead that cannot be parallelized. Making them fast is challenging, however, when there are many workers, data objects, and tasks, because they require checking a great deal of state.

Nimbus uses two optimizations to keep validation and patching fast. The first optimization relates to template generation. When generating a worker template, Nimbus ensures that the precondition of the template holds when it finishes. By doing so, it ensures that tight inner loops, which dominate execution time and control plane traffic, automatically validate and need no patching. As an example, in Figure 5(b), this adds a data copy of object 1 to worker 2 at the end of the template.

Second, workers cache patches and the controller can invoke these patches much like a template. When a worker template fails validation, the controller checks a lookup table of prior patches indexed by what executed before that template. If the cached patch will correctly patch the template, it sends a single command to the worker to instantiate the patch. When patches require multiple data copies, the cache helps reduce the networking overhead at the controller. We have found that the patch cache has a very high hit rate in practice because control flow, while dynamic, is typically not very complex.

4.3 Edits

Whenever a controller instantiates a worker template, it can attach a list of edits for that template to apply before

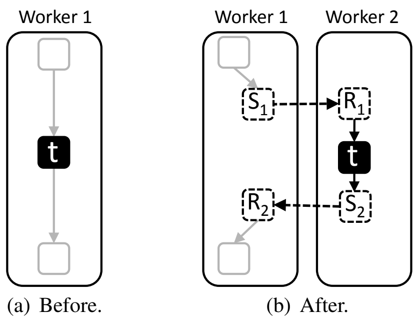


Figure 6: Edits to reschedule a task. The controller removes the task from worker 1's template and adds two data copy commands (S_1, R_2). It adds the task and two data copy commands (R_1, S_2) to worker 2's template.

instantiation. Each edit specifies either a new task to include or a task to remove. Edits are usually limited to the actual tasks being added or removed, because in cases when there are dependencies with other tasks, tasks are exchanged with data copy commands. Figure 6 shows, for example, how a task's entry in a before set is replaced by a data receive command. As long as the data receive command is assigned the same index within the command identifier array, other commands do not need to change. Using edits, minor changes in scheduling have very small costs. The cost scales linearly with the size of the change.

4.4 Fault Recovery

Nimbus implements a checkpoint recovery mechanism. Although a controller keeps the full lineage for every data object in the system, for iterative computations we found that linage-based recovery [42] is essentially identical to checkpointing because there are frequent synchronization points around shared global values. Any lineage recovery beyond a synchronization point requires regeneration of every data object, which is a checkpoint.

Nimbus automatically inserts checkpoints into the task stream from a driver program. When a checkpoint triggers, the controller waits until all worker task queues drain, stores a snapshot of the current execution graph, and requests every worker to write its live data objects to durable storage.

When a controller determines a worker has failed (it stops sending periodic heartbeat messages or workers depending on its data fall idle), it sends a halt command to every worker. On receiving the command, workers terminate all ongoing tasks, flush their queues, and respond back. Then, the controller sends commands to load the latest checkpoint into memory, reverts to the stored execution graph snapshot, and restarts execution.

5 Evaluation

This section evaluates execution templates in Nimbus, comparing them with Spark's fine-grained centralized scheduler, Naiad's high-throughput distributed data flow graphs [2], and application-level MPI messaging. In summary, the results show:

- Execution templates allow Nimbus to schedule hundreds of thousands of tasks per second, imposing a control overhead competitive with Naiad's distributed data flow graphs.
- Execution templates allow Nimbus to schedule at task granularity, providing a runtime flexibility and adaptivity equivalent to Spark's centralized scheduler.
- Execution templates are expressive enough to support complex, high-performance applications, such as a particle-levelset water simulation with a triply nested, data dependent loop and tasks as short as 100μs.

5.1 Methodology

All experiments use Amazon EC2 compute-optimized instances since they are the cheapest option for compute-bound workloads. Worker nodes use `c3.2xlarge` instances with 8 virtual cores and 15GB of RAM. Controllers run on a more powerful `c3.4xlarge` instance to show how jobs bottleneck on the controller even when it has more resources. All nodes are allocated in a single placement group and so have full bisection bandwidth.

We compare the performance of Nimbus with Spark 2.0 and Naiad 0.4.2 using two machine learning benchmarks, logistic regression and k-means clustering. Because our goal is to measure the task throughput and scheduling granularity of the control plane, we factor out language differences between the three frameworks and have them run tasks of equal duration. We chose the task duration as the fastest of the three frameworks, as it evaluates the highest task throughput. Nimbus tasks run 8 times faster than Spark's MLlib due to Spark using a JVM (a 4x slowdown) and its immutable data requiring copies (a 2x slowdown). Nimbus tasks run 3 times faster than Naiad due to Naiad's use of the CLR. To show that tasks in Naiad and Spark run as fast as C++ ones, we label them *Naiad-opt* and *Spark-opt*. This is done by replacing the task computations with a spin wait as long as C++ tasks.

The Naiad and Nimbus implementations of k-means and logistic regression include application-level two-level reduction trees. Application-level reductions in Spark harm completion time because they add more tasks that bottleneck at the controller.

[2] TensorFlow's control plane design is very similar to Naiad's which results in very close performance and behavior.

	Per-task cost
Installing controller template	25μs
Installing worker template on controller	15μs
Installing worker template on worker	9μs
Nimbus task scheduling	134μs
Spark task scheduling	166μs

Table 1: Template installation is fast compared to scheduling. The 49μs per-task cost is evenly split between the controller and worker templates. Installing a new worker template has a per-task cost of 24μs, and 18% overhead on centrally scheduling that task.

	Per-task cost
Instantiate controller template	0.2μs
Instantiate worker template	
with auto-validation	1.7μs
with explicit-validation	7.3μs

Table 2: Template instantiation is fast. For the common case of a template automatically validating (repeated execution of a loop), instantiation takes 1.9μs/task: Nimbus can schedule over 500,0000 tasks/sec. If dynamic control flow requires a full validation, it takes 7.5μs/task and Nimbus can schedule 130,000 tasks/second.

5.2 Micro-Benchmarks

This section presents micro-benchmark performance results. These results are from a logistic regression job with a single controller template with 8,000 tasks, split into 100 worker templates with 80 tasks each.

Table 1 shows the costs of template installation. We report the per-task costs because they scale with the number of tasks (there are individual task messages). We also report the cost of centrally scheduling a task in Spark and Nimbus to give context. Installing a template has a one-time cost of installing the controller template and the potentially repeated cost of installing worker templates. Adding a task to a controller template takes 25μs. Adding it to a worker template takes 24μs. In comparison to scheduling a task (134μs), this cost is small. Installing all templates has an overhead of 36% on centrally scheduling tasks.

Table 2 shows the costs of template instantiation. There are two cases for the worker template. In the first (common) case, the template validates automatically because it is instantiated after the same template. Since Nimbus ensures that a template, on completion, meets its preconditions, in this case the controller can skip validation. In the second case, a different worker template is instantiated after the previous one, and controller must fully validate the template. When executing the inner loop of a computation, Nimbus's scheduling throughput is over 500,000 tasks/second (0.2μs + 1.7μs per task).

	Cost
Nimbus single edit	$\approx 41\mu s$
Nimbus rescheduling 5% of tasks (800 edits)	35ms
Nimbus complete installation (8000 tasks)	203ms
Naiad any scheduling change	230ms

Table 3: A single edit to the logistic regression job takes 41μs in Nimbus, and the cost scales linearly with the number of edits. Edits are less expensive than full installation when rescheduling as high as 5% of the tasks in templates. Any change in Naiad induces the full cost of data flow installation.

Table 3 shows the costs of edits. A single edit (removing or adding a task) takes 41μs[3]. Edits allow controllers to inexpensively make small-scale changes to worker templates. For example, 800 edits (e.g., rescheduling 5% of the tasks) takes 35ms, fraction of complete installation cost. The cost of installing physical graphs on Naiad, caused by any change to the schedule, is about 230ms.

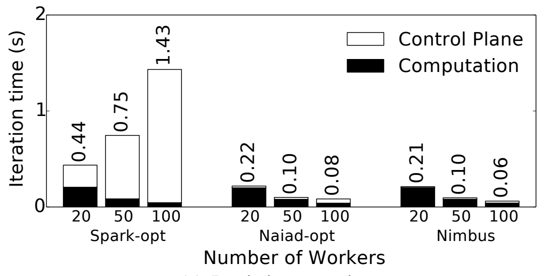
5.3 Control Plane Performance

This section evaluates the scalability of execution templates and their impact on job completion time. Figure 7 shows the results of running logistic regression and k-means clustering over a 100GB input once data has been loaded and templates have been installed. We observed negligible variance in iteration times and report the average of 30 iterations.

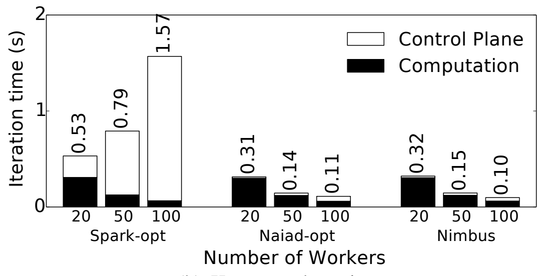
Nimbus and Naiad have equivalent performance. With 20 workers, an iteration of logistic regression takes 210-220ms; with 100 workers it takes 60-80ms. The slightly longer time for Naiad with 100 workers (80ms) is due to the Naiad runtime issuing many callbacks for the small data partitions; this is a minor performance issue and can be ignored. For k-means clustering, an iteration across 20 nodes takes 310-320ms and an iteration across 100 nodes takes 100-110ms. Completion time shrinks slower than the rate of increased parallelism because reductions do not parallelize.

Running over 20 workers, Spark's completion time is 70-100% longer than Nimbus and Naiad. With greater parallelism (more workers), the performance difference increases: Naiad and Nimbus run proportionally faster and Spark runs slower. Over 100 workers, Spark's completion time is 15-23 times longer than Nimbus. The dif-

[3]It is greater than the cost of installing a task in a worker template (29μs) due to the necessary changes in the task graph and inserting extra copy tasks (see Figure 6).



(a) Logistic regression



(b) K-means clustering

Figure 7: Iteration time of logistic regression and k-means for a data set of size 100GB. Nimbus executes tasks implemented in C++. Spark-opt and Naiad-opt show the performance when the computations are replaced with spin-wait as fast as tasks in C++. Execution templates helps centralized controller of Nimbus scale out almost linearly.

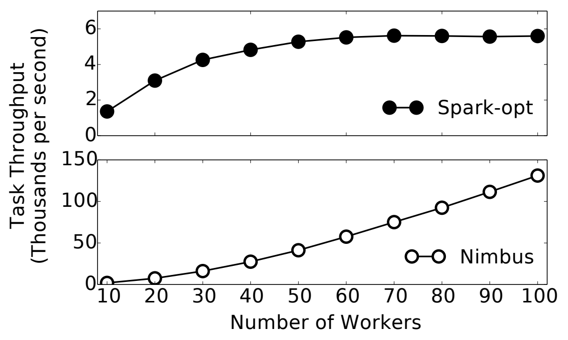


Figure 8: Task throughput of Nimbus and Spark as the number of workers increases. Spark saturates at about 6,000 tasks per second, while Nimbus grows to adapt to the number of tasks required for more parallelism. Note that the y-axis scale is different in the plots.

ference is entirely due to the control plane. Spark workers spend most of the time idle, waiting for the Spark controller to send them tasks.

Figure 8 shows the rate at which Nimbus and Spark schedule logistic regression tasks as the number of workers increases. Spark quickly bottlenecks at 6,000 tasks per second. Nimbus scales to support the increasing task throughput: a single iteration over 100 workers takes 60ms and executes 8,000 tasks, which is 128,000 tasks/second (25% of Nimbus's maximum throughput). Note that greater parallelism increases the task rate superlinearly because it simultaneously creates more tasks and makes those tasks shorter.

5.4 Dynamic Scheduling

Figure 9 shows the scenario of running a logistic regression job over 100 workers and rescheduling 5% of tasks every 5 iterations. The incremental edit cost lets Nimbus finish 20 iterations almost twice as fast as Naiad. Even if there is a single task rescheduling, Naiad would

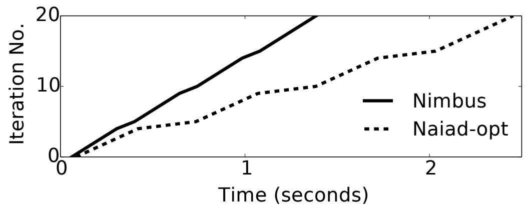


Figure 9: Logistic regression over 100 workers with task rescheduling every 5 iterations. Nimbus's edits have negligible overhead, while Naiad requires complete data flow installation for any scheduling change.

behave similarly; however, Nimbus's overhead remains negligible even in such an extreme scenario. Note that, current Naiad implementation does not support any data flow flexibility once the job starts, so the curve here is simulated from the numbers in Table 3 and Figure 7(a).

Figure 10 shows the time per iteration of logistic regression in Nimbus as a cluster manager adjusts the available resources. The run starts with templates disabled: the control plane overhead of a centralized scheduler dominates iteration time: each iteration takes 1.07s. At iteration 10, the driver starts using templates. Iteration 10 takes ≈ 1.3s, as installing each of the 8,000 tasks in the controller template adds 25μs (Table 2). On iteration 11, the controller template has been installed, and the controller generates its half of the worker template as it continues to send individual tasks to workers. This iteration is faster because the control traffic between the driver and controller is a single instantiation message. On iteration 12, the controller half of the worker templates has been installed, and the controller sends tasks to and installs templates on the workers. On iteration 13, templates are fully installed and an iteration takes 60ms (as in Figure 7(a)), with minimal control plane overhead.

At iteration 20, the cluster resource manager revokes 50 workers from the job's allocation. On this iteration, the controller regenerates the controller half of the

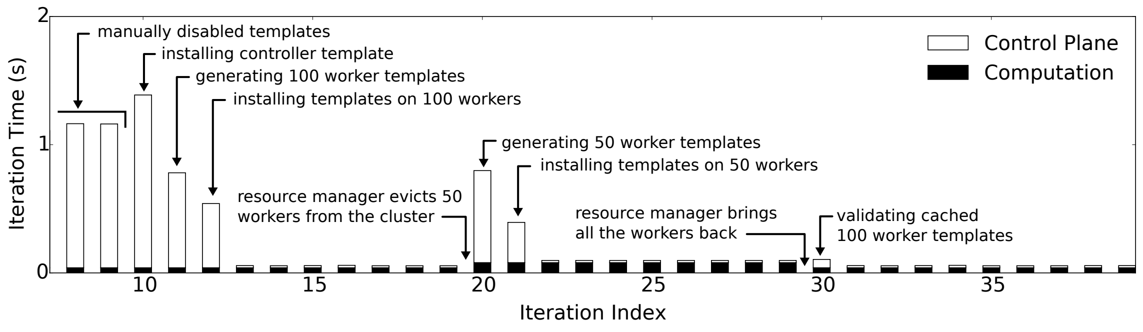
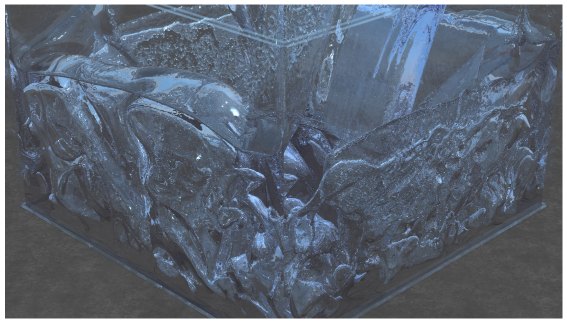
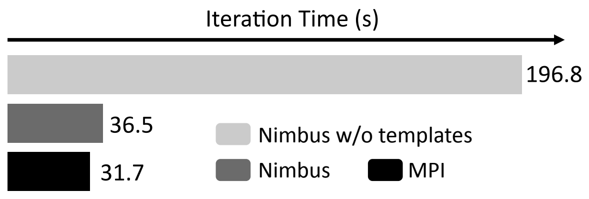


Figure 10: Execution templates can schedule jobs with high task throughputs while dynamically adapting as resources change. This experiment shows the control overheads as a cluster resource manager allocates 100 nodes to a job, revokes 50 of the nodes, then later returns them.



(a) Still of water pouring into a glass bowl.



(b) Iteration time of the main outer loop.

Figure 11: PhysBAM water simulation.

worker template, rescheduling tasks from evicted workers to remaining workers. On iteration 21, the controller installs new worker templates on the 50 workers. Computation time doubles because each worker is performing twice the work.

At iteration 30, the cluster resource manager restores the 50 workers to the job's allocation. The controller reverts to using the original worker templates and so does not need to install templates. However, on this first iteration, it needs to validate the templates. After this explicit validation, each iterations takes 60ms.

5.5 Complex Applications

To evaluate if execution templates can handle full applications with complex control flows, we use PhysBAM, an open-source computer graphics simulation library [12]. We ported PhysBAM to Nimbus, wrapping PhysBAM functions inside tasks and interfacing PhysBAM data objects (level sets, mark-and-cell grids, particles) into Nimbus.

We ran a canonical particle-levelset fluid simulation benchmark, water being poured into a glass [13]. This is the same core simulation used for the ocean in The Perfect Storm and the river in Brave. It has a triply-nested loop with 21 different computational stages that access over 40 different variables. The driver program has 8 basic blocks, three of them require extra data copies for auto validation, and two of them have non-deterministic entry points with different patch sets. Systems with static data flow (e.g., Naiad) cannot run this simulation efficiently because the termination conditions of its two inner loops are based on data values. Without data dependent branches, each loop instance must run as many iterations as the longest instance, which is wasteful when the loop converges faster.

We ran a 1024^3 cell simulation (512GB-1TB of RAM) on 64 workers. The median task length is 13ms, 10% of tasks are <3ms and some tasks are as short as 100μs. Figure 11 shows the results of running the simulation with PhysBAM's hand-tuned MPI libraries, in Nimbus without templates and in Nimbus with templates. The MPI libraries cannot rebalance load, and in practice developers rarely use them due to their brittle behavior and lack of fault tolerance. Without templates, the central controller becomes the bottleneck and the simulation takes 520% longer than MPI. With templates, the simulation runs within 15% of the MPI implementation, while providing fine-grained scheduling, automatic fault tolerance, and adaptive load balancing.

6 Related Work

Execution templates build on a large body of prior work that can be divided into three major categories: cloud frameworks, cloud schedulers, and high performance computing.

Cloud frameworks schedule tasks from a single job. Systems such as CIEL [29], Spark [42] and Optimus [24] keep all execution state on a central controller, dynamically dispatching tasks as workers become ready. This gives the controller an accurate, global view of the job's progress, allowing it to quickly respond to failures, changes in available resources, and system performance. Execution templates borrow this model, but cache control plane decisions to drastically increase task throughput for strong scalability.

Systems such as Naiad [28] and TensorFlow [3] take the opposite approach, statically installing an execution plan on workers so the workers can locally generate tasks and directly exchange data. Execution templates borrow this idea of installing execution plans at runtime but generalize it to support multiple active plans and dynamic control flow. Furthermore, execution templates maintain fine-grained scheduling by allowing a controller to edit the current execution plan.

Frameworks such as Dryad [20], DryadLINQ [40], and FlumeJava [8], as well as programming models such as DimWitted [25], DMLL [7] and Spark optimizations [32, 41, 4, 39, 38] focus on abstractions for parallel computations that enable optimizations and high performance, in some cases faster than hand-written C. This paper examines a different but complementary question: how can a framework's runtime scale to support the resulting fast computations across many nodes?

Cloud schedulers (also called cluster managers) schedule tasks from many concurrent jobs across a collection of worker nodes. Because these schedulers have global knowledge of all of the tasks in the system, they can efficiently multiplex jobs across resources[17], improve job completion time [14], fairly allocate resources across jobs [15], follow other policies [6, 11, 19], or allow multiple algorithms to operate on shared state [33].

Traditional centralized schedulers have transitioned to distributed or hybrid models. In Sparrow [31], each job runs its own independent scheduler that monitors the load on workers. These schedulers independently make good cooperative scheduling decisions based on mechanisms and principles derived from the power of two choices [27]. Tarcil uses a coarser grained approach, in which multiple schedulers maintain copies of the full cluster state, whose access is kept efficient through optimistic concurrency control because conflicts are rare [11]. Hawk's hybrid approach centrally schedules long-running jobs for efficiency and distributes short job scheduling for low latency [10]. Mercury allows multiple schedulers to request resources from a shared pool and then schedule tasks on their resources [23].

These distributed and hybrid schedulers address the problem of when the combined task rate of multiple jobs is greater than what a centralized scheduler can handle. Execution templates solve a similar, but different problem, when the control plane bottlenecks a *single* job. Like Sparrow, a framework using execution templates requests allocation from its cluster manager.

High performance computing (HPC) embraces the idea that an application should be responsible for its own scheduling as it has the greatest knowledge about its own performance and behavior. HPC systems stretch from very low-level interfaces, such as MPI [34], which is effectively a high performance messaging layer with some support for common operations such as reduction. Partitioning and scheduling, however, is completely an application decision, and MPI provides very little support for load balancing or fault recovery. HPC frameworks such as Charm++ [22] and Legion [5] provide powerful abstractions to decouple control flow, computation and communication, similar to cloud frameworks. Their fundamental difference, however, is that these HPC systems only provide mechanisms; applications are expected to provide their own policies.

7 Discussion and Conclusion

Analytics frameworks today provide either fine-grained scheduling or high task throughput but not both. Execution templates enable a framework to provide both simultaneously. By caching task graphs on the controller and workers, execution templates are able to schedule half a million tasks per second (Table 2). At the same time, controllers can cheaply edit templates in response to scheduling changes (Table 3). Finally, patches allow execution templates to support dynamic control flow.

Execution templates are a general control plane abstraction. However, the requirements listed in Section 3 are simpler to incorporate in some systems than others. Incorporating execution templates into Spark requires two changes: workers need to queue tasks and resolving dependencies locally and workers need to be able to exchange data directly (not go through the controller for lookups). Naiad's data flow graphs as well as TensorFlow's can be thought of as an extreme case of execution templates, in which the flow graph describes a very large, long-running basic block. Allowing a driver to store multiple graphs, edit them, and dynamically trigger them would bring most of the benefits.

Acknowledgments This work was funded by the National Science Foundation (CSR grant #1409847) and conducted in conjunction with the Intel Science and Technology Center - Visual Computing. The experiments were made possible by a generous grant from the Amazon Web Services Educate program.

References

[1] Apache Hadoop. http://wiki.apache.org/hadoop.

[2] Facebook AI Research open sources deep-learning modules for Torch. https://research.facebook.com/blog/fair-open-sources-deep-learning-modules-for-torch/.

[3] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Savannah, Georgia, USA*, 2016.

[4] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM, 2015.

[5] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11. IEEE, 2012.

[6] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. Apollo: scalable and coordinated scheduling for cloud-scale computing. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 285–300, 2014.

[7] K. J. Brown, H. Lee, T. Rompf, A. K. Sujeeth, C. De Sa, C. Aberger, and K. Olukotun. Have abstraction and eat performance, too: Optimized heterogeneous computing with parallel patterns. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 194–205. ACM, 2016.

[8] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. In *ACM Sigplan Notices*, volume 45, pages 363–375. ACM, 2010.

[9] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[10] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel. Hawk: hybrid datacenter scheduling. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 499–510, 2015.

[11] C. Delimitrou, D. Sanchez, and C. Kozyrakis. Tarcil: reconciling scheduling speed and quality in large shared clusters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 97–110. ACM, 2015.

[12] P. Dubey, P. Hanrahan, R. Fedkiw, M. Lentine, and C. Schroeder. Physbam: Physically based simulation. In *ACM SIGGRAPH 2011 Courses*, SIGGRAPH '11, pages 10:1–10:22, New York, NY, USA, 2011. ACM.

[13] D. Enright, R. Fedkiw, J. Ferziger, and I. Mitchell. A hybrid particle level set method for improved interface capturing. *Journal of Computational Physics*, 183(1):83–116, 2002.

[14] M. R. Garey, D. S. Johnson, and R. Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of operations research*, 1(2):117–129, 1976.

[15] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, volume 11, pages 24–24, 2011.

[16] I. Gog, M. Schwarzkopf, A. Gleave, R. N. M. Watson, and S. Hand. Firmament: Fast, centralized cluster scheduling at scale. In *To appear in Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*. USENIX, 2016.

[17] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 455–466. ACM, 2014.

[18] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni. Do the hard stuff first: Scheduling dependent computations in data-analytics clusters. *arXiv preprint arXiv:1604.07371*, 2016.

[19] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[20] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 59–72. ACM, 2007.

[21] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 261–276. ACM, 2009.

[22] L. V. Kale and S. Krishnan. *CHARM++: a portable concurrent object oriented system based on C++*, volume 28. ACM, 1993.

[23] K. Karanasos, S. Rao, C. Curino, C. Douglas, K. Chaliparambil, G. M. Fumarola, S. Heddaya, R. Ramakrishnan, and S. Sakalanaga. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 485–497, 2015.

[24] Q. Ke, M. Isard, and Y. Yu. Optimus: a dynamic rewriting framework for data-parallel execution plans. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 15–28. ACM, 2013.

[25] J. Liu and S. J. Wright. Asynchronous stochastic coordinate descent: Parallelism and convergence properties. *SIAM Journal on Optimization*, 25(1):351–376, 2015.

[26] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, 2014.

[27] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[28] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013.

[29] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. Ciel: A universal execution engine for distributed dataflow computing. In *NSDI*, volume 11, pages 9–9, 2011.

[30] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, B.-G. Chun, and V. ICSI. Making sense of performance in data analytics frameworks. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 293–307, 2015.

[31] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 69–84. ACM, 2013.

[32] S. Palkar, J. Thomas, and M. Zaharia. Nested vector language: Roofline performance for data parallel code. `http://livinglab.mit.edu/wp-content/uploads/2016/01/nvl-poster.pdf`.

[33] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 351–364. ACM, 2013.

[34] M. Snir. *MPI–the Complete Reference: The MPI core*, volume 1. MIT press, 1998.

[35] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[36] S. Venkataraman, A. Panda, K. Ousterhout, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica. Drizzle: Fast and adaptable stream processing at scale.

[37] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *Proceedings of the 13th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2016.

[38] R. Xin. Technical Preview of Apache Spark 2.0 Now on Databricks. `https://databricks.com/blog/2016/05/11/apache-spark-2-0-technical-preview-easier-faster-and-smarter.html`.

[39] R. Xin and J. Rosen. Project Tungsten: Bringing Apache Spark Closer to Bare Metal. `https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html`.

[40] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, volume 8, pages 1–14, 2008.

[41] M. Zaharia. New developments in spark and rethinking apis for big data. http://platformlab.stanford.edu/Seminar%20Talks/stanford-seminar.pdf.

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