FAST, DISTRIBUTED COMPUTATIONS IN THE CLOUD

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Omid Mashayekhi

June 2017

This dissertation is online at: http://purl.stanford.edu/nw054dq9276

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Philip Levis, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Alex Aiken

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Mendel Rosenblum

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

Today, cloud computing frameworks adopt one of two strategies to schedule their computations over hundreds to thousands of machines. In the centralized strategy, a controller dispatches computation units, called tasks, to worker nodes. Centralization allows a framework to quickly reschedule, respond to faults, and mitigate stragglers. However, a centralized controller can only schedule a few thousand tasks per second and becomes a bottleneck. In the distributed strategy, there is no central node. 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 can execute hundreds of thousands of tasks per second, and do not have a control plane bottleneck. However, data flow graphs describe a static schedule; even small changes, such as migrating a task between two nodes, requires stopping all nodes, recompiling the flow graph and reinstalling it on every node. This leads to high latency or wasteful resource utilization for rescheduling, fault recovery, or straggler mitigation.

This dissertation presents a new, third strategy, called *execution templates*. Execution templates leverage a program's repetitive control flow to cache control plane decisions in templates. A template is a parametrisable block of tasks: it caches some information (e.g., task dependencies) but instantiation requires some parameters (e.g., task identifiers). Executing the cached tasks in a template requires sending a single message that loads the new parameters. Large-scale scheduling changes install new templates, while small changes apply edits to existing templates. Execution templates are not bound to a static control flow and efficiently capture nested loops and data dependent branches. Evaluation of execution templates in Nimbus, a cloud

computing framework, shows that they provide the fine-grained scheduling flexibility of centralized control planes while matching the performance of the distributed ones. Execution templates in Nimbus support not only the traditional data analytics, but also complex, scientific applications such as hybrid graphical simulations.

Acknowledgements

Graduate school at Stanford University was a once-in-a-lifetime opportunity. Not only I received the highest level of academic education, but also I obtained diverse and lifelong skills ranging from playing golf and windsurfing to social ballroom dancing.

I am thankful to my advisor, Philip Levis, for his guidance throughout my studies at Stanford. He is a brilliant researcher and the best teacher I have ever had. He made me a better system researcher and code developer. Thanks also to the rest of my reading and defence committee – Alex Aiken, John, Gill, Mendel Rosenblum, Matei Zaharia – for their invaluable feedback and encouraging words.

I have been lucky to work alongside smart and hard working students. Specifically, the work presented in this dissertation is the result of direct collaborations with Chinmayee Shah and Hang Qu. I have also enjoyed working, and learned a lot from other fellow members of Stanford Information Networks Group (SING). Thanks to Behram, Ewen, Holly, Jae-Young, Judson, Kevin, Laurynas, Luke, Maria, Raejoon, Tahir; your friendship meant a lot to me during the graduate school.

The work presented here and my PhD career would not have been possible without financial support of Stanford Graduate Fellowship (SGF), National Science Foundation (CSR grant #1409847), and the Intel Science and Technology Center - Visual Computing. The experiments were made possible by a generous grant from the Amazon Web Services Educate program.

Last but not least, I would like to thank my family Abolfazl, Neda, Negin, Navid for their unconditional love and support in every aspect of my life. I consider myself very fortunate to have two wonderful parents. Their care, sacrifice and generosity built my foundation and enabled me to pursue my dreams.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Over the past decade, data center and cloud computing has transformed the scale and scope at which we can process data. Being able to harness hundreds to thousands of machines in order to process hundreds of gigabytes to petabytes of information has enabled big data analytics. Modern life is entwined with the technologies powered by big data analytics: search engines and information retrieval [39, 74, 89, 110], social networks [33, 93], speech and image recognition [69, 47], and natural language processing [37, 116], to name a few. Collecting and processing massive datasets has rekindled interest in many traditional algorithms and techniques such as neural networks [90, 66, 120], machine learning [45, 108, 87, 83, 53], and graph processing algorithms [100, 111, 30, 124, 82]. Most of these techniques are decades old, but big data has deployed them for practical applications.

These applications run in data centers against a high-level API of cloud computing frameworks. Application developers write simple programs. The framework transforms the program into many smaller computation units, called *tasks*, and seamlessly distributes them over the cluster. Cloud computing frameworks hide the intricacies of the underlying cluster from the programmers. They provide elastic scalability [6, 48], multi-tenant execution and resource sharing [117, 67, 99, 50], load balancing and straggler mitigation [29, 130, 26], as well as fault recovery [48, 128].

The performance characteristics of cloud frameworks have evolved over the time. They have transitioned from file I/O processing [6, 48] to in-memory processing [86,

94, 128] with transparent optimization layers [121, 1, 101, 127, 102]. This has made tasks orders of magnitude faster for many applications: optimized tasks that operate on in-memory data are as fast as tens of milliseconds. Furthermore, the straggler mitigation techniques [97], and strong scaling gains for interactive and stream processing [94, 24] has encouraged breaking up tasks in to even smaller units for faster completion times. Overall, the trend shows increasing need for supporting applications with milliseconds long tasks [99, 94].

Speedup in computations demands a higher task throughput from a framework. Also, it imposes lower latency requirements for dynamic scheduling decisions, such as reactively changing the task partitioning due to changes in resources or stragglers [117, 67, 99, 109]. Current frameworks cannot deliver high task throughput requirements without sacrificing fast, dynamic scheduling. Centralized frameworks provide low latency, dynamic scheduling but have limited task throughput [48, 6, 128]. Distributed frameworks support high task throughput, but either have static scheduling or impose a high latency for dynamic changes [94, 22].

This dissertation introduces a new abstraction, called *execution templates*, that provides high task throughput with fast, dynamic scheduling. Execution templates take advantage of the fact that long running applications are iterative in nature; different iterations compute similar tasks with only minor differences. Caching the tasks in parametrisable blocks increases task throughput. The cached blocks can be modified at task granularity to keep the cost of scheduling changes proportional to the extent of scheduling differences from one iteration to the other. This makes dynamic scheduling feasible with execution templates. Also, execution templates provide a patching mechanism to capture applications with dynamic program control flow such as nested loops and data dependent branches.

Execution templates are implemented in Nimbus, an analytics framework designed for fast computations. Nimbus supports traditional data analytics applications, as well as complex, scientific computations such as hybrid graphical simulations. Evaluations show that execution templates enable Nimbus to match the performance of distributed frameworks, while providing fast, dynamic scheduling decisions similar to centralized frameworks.

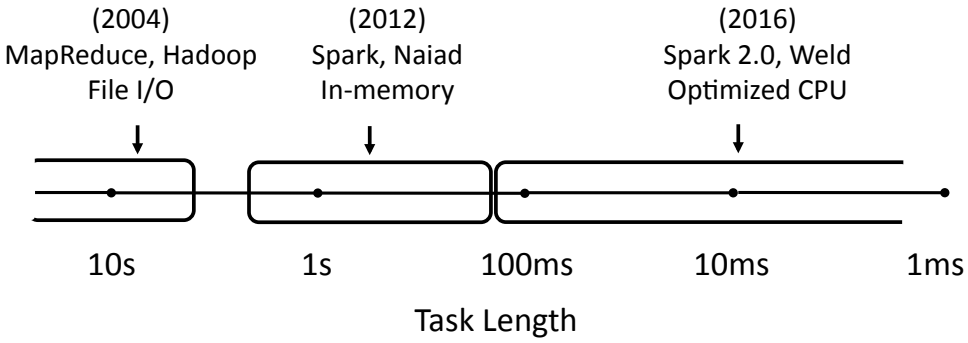


Figure 1.1: Cloud computing frameworks have evolved from file I/O to in-memory processing with CPU optimization layers. As a result, the computation units, called tasks, have become 2–3 orders of magnitude faster.

1.1 Evolution of Cloud Computing

Each task in a framework operates on a partition of data, and might mutate it or produce new data objects. As frameworks transitioned from file I/O to in-memory processing with CPU optimization layers, tasks have become orders of magnitude faster. Figure 1.1 shows the evolution of cloud computing frameworks and changes in their performance characteristics over time.

MapReduce [48] is the first general purpose and fault tolerant cloud computing system. Although there had been a few parallel programming systems before MapReduce, they are based on restricted programming models, tailored for small clusters, and leave the failure handling details to the programmers [80, 35, 62]. Each MapReduce task reads a chunk of data off disk as input, performs a computation, and then pushes the intermediate and final results back to disk. Using this file-backed model, MapReduce can process petabytes of data despite the failures and main memory limitations. However, the tasks are I/O-bound: progress is limited by disk speed.

More recently, systems such as Spark [128] and Naiad [94] have observed that many applications, when distributed across enough workers, can keep their entire working set in memory. Instead of having each task read from and write to disk, as frameworks such as MapReduce [48] and Hadoop [6] do, these systems operate on in-memory state. This allows multiple computations over in-memory data, for iterative and interactive applications. As a result, they have shown, orders of magnitude task

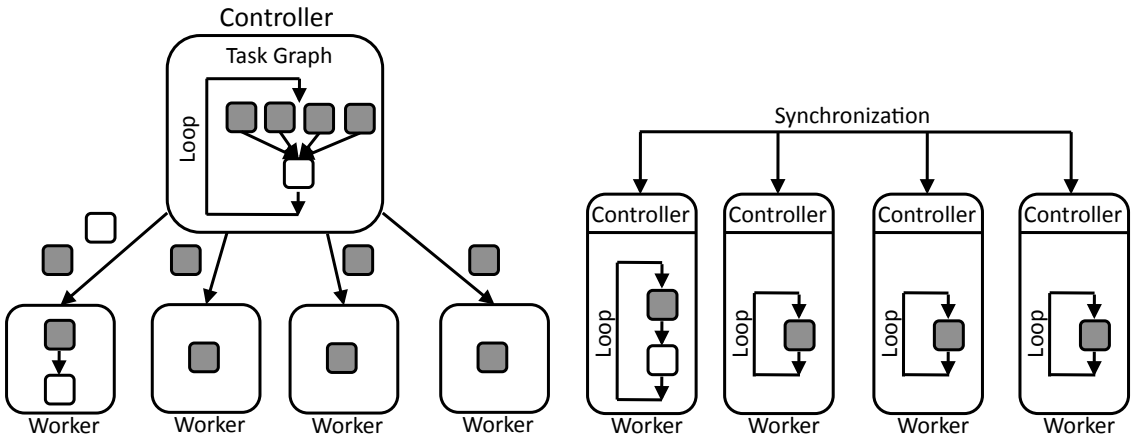
speedups. Nowadays, commercialized clusters [3, 13, 17] elastically scale to hundreds of gigabytes of memory capacity, and enable increasing number of applications to fit their entire dataset in memory [10].

Tasks in in-memory cloud frameworks are CPU-bound: most of the time is spent processing data [98]. As a result, systems have begun focusing on optimizing CPU performance. Spark 2.0, for example, reports 10x speedups over prior versions with new code generation layers [121, 1]. Introducing data-parallel optimizations such as vectorization, branch flattening, and prediction can, in some cases, be faster than handwritten C [101, 127]. Runtimes for cross-library optimizations, e.g. Weld [102], and GPU-based computations [11, 22] improve performance even further.

1.2 Control Plane Implications

The speedup in computations has implications for the design and performance of the control plane of the cloud computing frameworks. The *control plane* transforms the driver program of the applications into individual tasks executing on the workers. The details of the control plane roles are framework dependent, but generally include generating and spawning tasks, coordinating global operations such as shuffles and reductions, tracking the execution progress, and providing load balancing and fault tolerance. The design goals for the control plane are, on one hand, high task throughput, and low latency, dynamic scheduling on the other. Today, frameworks adopt one of two strategies for their control plane design. One is a centralized controller model, and the other is a distributed data flow model. Figure 1.2 depicts these two strategies. Each instance of the application is launched as a *job* in the framework which consists of many tasks. The dataflow enforces the ordering among tasks, which is abstracted as a directed acyclic graph, called *task graph*.

In the centralized model, systems such as Spark [128] and MapReduce [48] use a single control node that dispatches tasks to worker nodes, as depicted in Figure 1.2(a). Centralization allows a framework to quickly reschedule, respond to faults, and mitigate stragglers *reactively*, but as tasks get shorter the control plane becomes a bottleneck. Available centralized controllers can only support a few thousand tasks per



(a) Centralized control plane model. (b) Distributed control plane model.

Figure 1.2: Two control plane design strategies in the cloud computing frameworks: (a) a centralized controller that allows fast, dynamic scheduling but has limited task throughput, (b) a distributed control plane that sustains orders of magnitude higher task throughput but suffers from high latency for scheduling changes.

second [99]. With the tasks in the order of milliseconds, even a cluster with few workers can quickly saturate the controller's capacity. While there is a huge body of work for scheduling *multiple* jobs within a cluster [36, 49, 50, 67, 78, 99, 117], these approaches do not help when a *single* job has a higher task throughput than what the control plane can handle.

The distributed model, used by systems such as Naiad [94] and TensorFlow [22], is a fully-distributed control plane as shown in Figure 1.2(b). When a job starts, these systems install data flow graphs on each worker, which then independently execute and exchange data. By distributing the control plane and turning it into data flow, these frameworks have strong scaling and can execute hundreds of thousands of tasks per second. However, data flow graphs describe a static schedule. Even small changes, such as migrating a task between two workers, require stopping the job, recompiling the flow graph and reinstalling it on every worker. 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 [22].

In summary, speedup in computations pressures the control plane to process tasks

Control Plane Design	Example Framework	Task Throughput (task per sec)	Scheduling Cost (per task)
Centralized	MapReduce [48] Hadoop [6] Spark [128]	$\approx 1,000$	$\approx 100\mu$s
Distributed	Naiad [94] TensorFlow [22]	$\approx 100,000$	$\approx 100,000\mu$s
Centralized w/ Execution Templates	Nimbus (§5)	$\approx 100,000$	$\approx 100\mu$s

Table 1.1: Current cloud computing frameworks have either a centralized control plane model with fast, dynamic scheduling but limited task throughput, or a distributed control plane model with orders of magnitude higher task throughput but very high scheduling cost. Execution templates (§4) introduced by this dissertation enable Nimbus (§5) to match the task throughput of a distributed framework, while providing the fast, dynamic scheduling similar to centralized frameworks.

at a higher rate to keep workers busy. Current frameworks fail to deliver the high task throughput required by the applications at scale without sacrificing low latency, dynamic scheduling. Table 1.1 summarizes these two design approaches for the control plane with example frameworks and their characteristics in terms of task throughput and dynamic scheduling cost.

1.3 Execution Templates

This dissertation presents a third strategy using an abstraction called *execution templates*. Execution templates schedule at the same per-task granularity as centralized schedulers do. 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 and graph processing) are iterative, running the same computation many times [119]. Machine learning algorithms, for example, typically iterate until the estimation error drops below a threshold.

Logically, a framework using execution templates centrally schedules at the task

granularity generated based on the driver program. As it generates and schedules tasks, however, the system caches its decisions and state in parametrisable blocks as templates. The next time the job reaches the same part of its program, the system executes from the templates rather than resend all of the tasks. We call this abstraction a template because it caches some information (e.g., dependencies) but instantiating it requires parameters (e.g., task IDs).

Once installed, execution templates provide three mechanisms:

1. **Instantiation** allows executing the tasks cached in the template by only loading a small set of parameters. The extent of changing parameters depends on the underlying framework and, for example, could be as simple as passing new task identifiers to each task. This mechanism helps improving the task throughput by avoiding reprocessing each individual task from scratch at the control plane for iterative workloads.

2. **Edits** enable fine-grained modifications in the templates by adding/removing individual tasks to/from a cached block. This mechanism brings in reactive load balancing in response to fine-grained scheduling changes despite the fact that the cached blocks might be based on an obsolete scheduling decision.

3. **Patching** adapts execution templates to dynamic changes in the program flow control. Each template has a set of preconditions that have to meet for the execution of its tasks. These preconditions depend on the underlying framework, and for example, could be the set of data objects needed in memory for the task executions. Patches are generated during run time if needed to match the system state with the preconditions. This mechanism allows dynamic program control flow to support nested loops and data dependent branches.

Depending on how much system state has changed since the template was installed, a controller can immediately *instantiate* the template, *edit* the template by changing some of its tasks, *patch* the template to satisfy preconditions, or *install* a new version of the template. Using execution templates, a centralized controller can generate and schedule hundreds of thousands of low-latency tasks per second through lightweight instantiation messages.

1.4 Nimbus: A Framework with Templates

To evaluate the benefits and tradeoffs of execution templates, we have implemented them in Nimbus, an analytics framework designed to support high performance computations. Nimbus embeds execution templates in its control plane and exploits a few novel optimizations to benefit the most from the mechanisms provided by templates. This dissertation presents the details of implementing execution templates in Nimbus, and lays out what it would take to incorporate them in other frameworks.

In order to show the generality of execution templates, we demonstrate that Nimbus can not only support traditional data analytics applications, but also is capable of running complex scientific applications, such as a graphical simulation. The program control flow of graphical simulations has nested loops and data dependent branches which leads to a nondeterministic and complex control flow. For example the driver program of a canonical water simulation has triply nested loops with eight basic blocks, two of which with multiple entry points. This is in contrast with the typically simple task graph of data analytics applications. The complex program control flow triggers subtle patching scenarios to match the templates with the changing state of the workers. This dissertation presents running these applications within a cloud framework, for the first time. Specifically, we have ported fluid simulations from PhysBAM library [51] in to Nimbus. PhysBAM is an open-source, physics-based, simulation package that has received two Academy Awards and has been used in over 20 feature films [20].

Distributing graphical simulations in a cloud framework is not straightforward. They require very different data and execution models than what current cloud computing systems provide. A graphical simulation uses multiple complex data models, such as a marker-and-cell grid [65] for the fluid volume, a dense particle field for the fluid surface [52], and a system of linear equations to ensure fluid does not disappear. These data structures are geometric in nature and computations on neighboring regions have tight dependencies. These requirements differ greatly from data tuples as in MapReduce [48], Spark [128], and Naiad [94] or graphs as in Pregel [88] and PowerGraph [60].This dissertation describes the design details that enable Nimbus to

automatically distribute a single-core grid-based and hybrid simulation. The novel data abstraction hides the distribution complexities from the simulation library developers as well as the driver program.

1.5 Thesis Statement and Contributions

This dissertation describes the design and implementation of a new scalable and flexible control plane for cloud computing frameworks. We argue that:

> Execution templates realize orders of magnitude higher task throughput than centralized frameworks, without sacrificing fine-grained, dynamic scheduling. By caching control plane decisions in parametrisable blocks, an execution template can dynamically schedule high performance computations. Execution templates are general enough to not only support traditional data analytics, but also complex applications with nested loops and data dependent branches that results in nondeterministic control flow.

The contributions of this dissertation are:

1. Evaluation and analysis of the existing cloud computing engines, demonstrating how the control plane is an emerging bottleneck for data analytics.

2. Execution templates, an abstraction for the control plane of cloud computing frameworks, that provide high task throughput and low cost, dynamic scheduling at the same time.

3. The design, implementation, and evaluation of Nimbus, a distributed cloud computing framework that embeds execution templates, 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.

4. An evaluation of execution templates implemented in Nimbus on analytics benchmarks, comparing them with the available frameworks with flexible centralized controllers, and high-throughput distributed data flow models.

5. A demonstration of a single-core PhysBAM [51] particle level-set fluid simulation [52] that Nimbus automatically distributes in the cloud showing execution templates in practice for complex applications with nested loops and data dependent branches.

1.6 Dissertation Outline

The rest of this dissertation is organized as follows. Chapter 2 provides a background of cloud computing systems and related work. Chapter 3 argues that control plane is becoming the bottleneck for cloud computing engines. Chapter 4 introduces execution templates and describes the mechanisms that they provide. Chapter 5 presents the design and implementation of Nimbus and the details of implementing execution templates in Nimbus. Chapter 6 evaluates execution templates and explores the performance of running data analytics under Nimbus, comparing the results with the performance of state of the art frameworks in regards to task throughput and task scheduling. Chapter 7 describes the data model and system design of Nimbus that enables automatic distribution of graphical simulations, and evaluates the performance of graphical simulations running under Nimbus. Finally, Chapter 8 discusses and concludes.

Chapter 2

Background and Related Work

This dissertation builds on a large history of prior work in large scale distributed and parallel computations. The related work can be divided into three major classes, each of which examines the problem from a different point in the software stack: cloud computing frameworks, cloud scheduling systems, and high performance computing.

2.1 Cloud Computing Frameworks

Cloud computing frameworks facilitate distributed programming for data-centric computations. Applications are written using simple interfaces, and the framework automatically partitions and distributes them across many nodes. The difficult aspects of distributed programming are abstracted away from the application developers by seamlessly handling concurrent execution, resource allocation, elastic scaling, networking, scheduling, straggler mitigation, and fault recovery.

Figure 2.1 depicts the overall structure of cloud computing frameworks. Application developers specify an application logic by writing a simple driver program. Each instance of the application runs as a *job* in the framework. The *control plane* transforms each job into many smaller computation units, called *tasks*, and harnesses a cluster of *workers* to execute the tasks. The control plane could be distributed or centralized as depicted in Figure 1.2.

There are many flavors of cloud computing frameworks, ranging from generalized

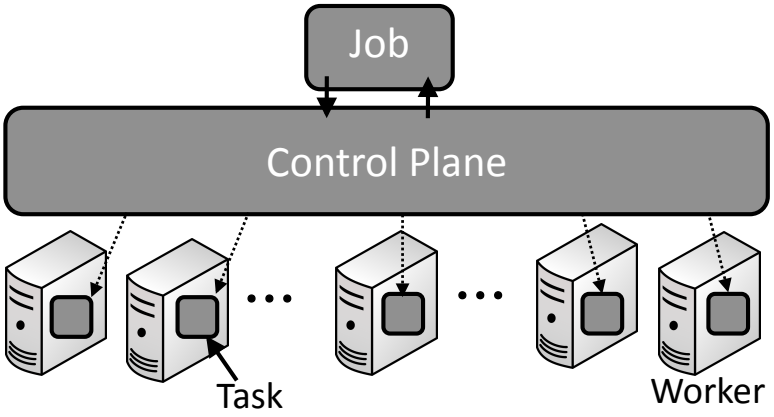


Figure 2.1: Each instance of an application runs as a *job* in the cloud computing framework. The *controller plane* transforms the driver program of a job into many smaller computation units, called *task*. Control plane assigns the task to a cluster of *workers* for execution.

systems to application specific frameworks. They differ in their architecture and design, and hence show various performance characteristics under different workloads.

2.1.1 Application Programming Interfaces

The application programming interface (API) of cloud computing frameworks abstracts the parallel operations with a high-level data model. The data model makes it easier for the programmers to specify the data flow of the application, and at the same time, gives frameworks the flexibility to distribute and schedule the computations, automatically. The level of abstraction and flexibilities vary from framework to framework. There is a tradeoff between simplicity and generality on one hand, and application domain optimizations, on the other. General data models fit many applications but cannot deliver the best performance in every case domains. In contrast, domain specific frameworks (e.g. for graph processing) restrict the interface in favor of workload dependent optimizations (e.g. vertex and edge caching).

Legacy Frameworks

MapReduce [48] introduced the first general-purpose application interface for cloud computations. Although there had been a few parallel programming systems before MapReduce, they were based on restricted programming models, were tailored for small clusters, and left the failure handling details to the programmers [80, 35, 62]. The goal of MapReduce was to leverage thousands of interconnected cheap PC's for large-scale, real-world computations [14]. For example, one of the major use cases was ranking the web for the purpose of search engine developed at Google based on the PageRank algorithm [100].

In MapReduce, applications are translated in to a series of *map* and *reduce* stages. The data object is modeled as a collection of *key-value* tuples. Each map stage receives a set of key-value pairs as input, and outputs a set of intermediate key-value pairs, which are fed to the reduce stage. The MapReduce runtime could dynamically execute many parallel mappers or reducers; however, all the intermediate tuples with a similar key are guaranteed to be aggregated in a single set for reduction. This requires a global barrier, and a *sort and shuffle* stage between the mappers and the reducers. The sort and shuffle is handled by the runtime, automatically. In MapReduce, data sets are accessed from and pushed to the disk, and so the execution survives node failures. For example, MapReduce at Google uses a replicated distributed file system, GFS [56], to store and manage the data sets.

MapReduce has been well received in the academia and industry, due to its simplicity and generality. Many distributed computations could be easily ported in to MapReduce abstraction. For example, Hadoop [6], the open source implementation of MapReduce in Java, and its application specific descendants [7, 96, 8, 5] are actively and commonly used in the industry for big data computations.

Dryad [72] provides a more general interface for coarse-grained data parallel applications. A Dryad program specifies computations as *vertices* that are connected with *channels* to form a dataflow graph. The Dryad runtime distributes the vertices among a cluster of computing machines and realizes data channels with files, TCP pipes, or shared memory accesses. Each vertex is a sequential program; however, Dryad schedules vertices concurrently on multiple machines and/or multiple cores of

a single machine.

The generalized vertex-channel abstraction allows application developers to specify arbitrary directed acyclic graph (DAG) for the communication patterns. This is in contrast with the general map and reduce stages in MapReduce that restrict dataflow patterns in favor of generality and simplicity. For many data intensive applications, choosing an application specific dataflow graph can greatly improve the performance. For example, implementing customized multi-level reduction schemes based on the intermediate data size and the number of reducing nodes could alleviate the communication channel bottlenecks and avoid running out of local memory space on the reducers [72].

Higher Level Programming Interfaces

The key-value or vertex-channel models, while general, are tedious to manage at application level. Hence, there have been numerous efforts for increasing programmer productivity in writing applications for cloud frameworks. For example, Pig [96], Hive [7], and Sawzall [104] are domain-specific languages for SQL queries on top of Hadoop and MapReduce. FlumeJava [42] provides a Java library for MapReduce pipelines through high level classes called *parallel collections*. The entire pipeline or multiple pipelines of MapReduce are implemented within a single Java program.

DryadLINQ [125] builds on top of Dryad and provides a set of language extensions and a novel programming model. Specifically, it provides a data model based on strongly typed .Net objects, and supports a hybrid imperative and declarative interface in high-level programming languages. The compiler and runtime enable high-level implementation of many graph and machine learning algorithms, as well as SQL queries, without losing significant performance compared to a direct, application specific implementation.

CIEL [95] and Optimus [79] introduce more flexible control flow mechanisms to the Dryad interface. A CIEL program can have data-dependent branches during run time, where the execution DAG is generated dynamically as tasks execute. Optimus extends CIEL operators for more general dynamic task graph rewriting in DryadLINQ.

Frameworks for In-Memory Computations

For application that can fit the entire work set in memory, file processing models induce unnecessary I/O cost. Unlike MapReduce and Dryad which rely on non-volatile storage for reading input data blocks and writing intermediate/output results for fault resilience, more recent frameworks provide interfaces that allow in-memory operations. Piccolo [106] allows parallel user defined functions to read and mutate distributed, in-memory key-value stores. Piccolo relies on checkpointing and rollback mechanisms to revive from machine failures.

Spark [128] introduced Resilient Distributed Datasets (RDD) abstraction for in-memory computations with efficient fault recovery mechanisms. Spark's immutable data model lends itself to lineage-based, parallel fault recovery. Spark provides a high-level programming interface for coarse-grained data parallel operations on RDDs such as *map*, *reduce*, and *groupBy*. There are applications specific libraries built on top of Spark for SQL queries [123, 31], graph processing [61], machine learning algorithms [9, 8, 19], and stream processing [129]. Simplicity and generality has made Spark one of the most common frameworks for application developments among data scientists [10]. Spark support interfaces in high-level languages such as Scala and Python, as well as lower level managed languages such as Java.

Naiad [94] is another framework for in-memory computations. It introduces timely dataflow abstraction to unify batch processing, iterative computations, and stream processing in a distributed manner. The distributed, event-based runtime handles low latency computation and incremental computing, very efficiently. Similar to Spark, Naiad provides high-level and coarse-grained data parallel operations, and also allows user defined, arbitrary dataflow graphs similar to Dryad in terms of vertices and edges. Vertices exchange messages along edges and implement callback functions for message send/receive events. Each message has an associated *timestamp* that reflects the program context or loop counter. The runtime synchronizes the operations among nodes through timestamps.

Transparent Code Optimizations

Since the inception of general purpose frameworks for in-memory computations [128, 94], numerous workload traces from academia and industry have shown that many applications have become CPU-bound: the majority of the time was spent at the machines, using CPU power to compute on the in-memory data [98]. However, due to high level abstractions in general purpose frameworks, most of the CPU cycles were wasted. For example, 70% of the Spark applications are written in Scala [127]. Scala code is compiled into Java bytecodes executed by the JVM. The translated bytecodes are not necessarily the most optimized version possible. Also, JVM adds extra overhead for garbage collection and increases memory footprint due to memory expansion. Overall, the Scala code could be two orders of magnitude slower than code written directly in native code. We explore this inefficiency in details, in Chapter 3.

This trend has led to a wave of research on optimizing the CPU performance in cloud computing frameworks. Programming models such as DimWitted [85] and DMLL [40], within the database and parallel computing communities, have explored the computational inefficiency of Spark code, proposing new programming models and frameworks to replace it. Spark 2.0 has reported 10x speedups for common SQL benchmarks compared to older Spark versions [121] through transparent optimizations for code generation layers [31]. Specifically, the second generation Tungsten engine, introduced in Spark 2.0, emits optimized bytecode at runtime that collapses the entire query into a single function, eliminating virtual function calls and leveraging CPU registers for intermediate data [122].

There is also ongoing research on a common intermediate language for Spark that provides a glossary of data-parallel optimizations (including vectorization, branch flattening and, prediction), suggesting performance in some cases even faster than hand-written C [101, 127]. Weld [102] takes the optimization one step further for cross library data access patterns. The results show prominent speedups even for libraries implemented in lower level languages by leveraging the application dependent data layout information.

Domain Specific Frameworks

In addition to general purpose cloud frameworks, many domain specific systems have been developed for 1) stream processing, 2) machine learning and neural networks and 3) graph processing.

MillWheel [24] and Storm [115] are designed for low latency, stream processing applications. With MillWheel, application developers specify a directed graph of computing nodes. The runtime distributes the nodes and ensures persistent flow of records among them, despite machine failures, through checkpointing of batched records and atomic transactions. Each record has a logical time associated with it, that simplifies time-based queries over the records. Strom (used internally at Twitter) has a similar abstraction, and builds on top of ZooKeeper [71] for distribution and consistency guarantees.

With the wide popularity of machine learning algorithms and neural networks among data scientists, there have been efforts to design specialized systems for these workloads. DistBelief [47] introduced a system design for learning algorithms with billions of model features. For these applications, updating the entire feature vector on every node (as in general purpose systems such as Spark and Naiad) is not feasible. However, each node only accesses a subset of feature vector. DistBelief keeps a centralized version of the features, which then nodes read and write partially. Parameter Server [84] extends this idea to asynchronous operations and more flexible consistency models. This is important for computations at scale, since with only synchronous implementations the progress speed is harshly affected by the tail latency. In Parameter Server, users can choose the consistency model based on the requirement of the workload to tradeoff between convergence rate and the progress speed.

TensorFlow [22] builds on top of DistBelief and Parameter Server, but provides generalized interface for heterogeneous clusters including multicore CPUs, GPUs, and specialized ASICs known as Tensor Processing Units. TensorFlow provides a reach library of algorithms with a customizable consistency model, such that programmers can easily deploy complex algorithms concisely. With TensorFlow the code developed for testing and production remains the same, greatly increasing programmer

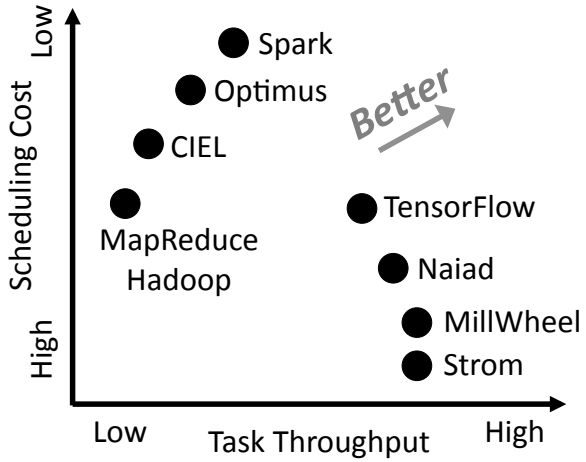


Figure 2.2: Relative position of prior cloud frameworks in terms of task throughput and dynamic task scheduling cost.

productivity.

Systems such as Pregel [88], PowerGraph [60] and GraphX [61] provide abstractions for expressing, and techniques for accelerating graph computations. Pregel introduces supersteps composed of vertex programs and messages between neighboring vertices. PowerGraph introduces vertex-cut partitioning to balance computation and reduce communication in natural graphs with a power-law degree distribution. GraphX builds on top Spark platform, and introduces caching and compression techniques to manage and exchange graph data efficiently.

2.1.2 Cloud Control Plane Design

There are two design philosophies for the control plane: 1) centralized controller, or 2) distributed dataflow. In the first strategy, frameworks have a centralized node that generates and assigns tasks to the workers [48, 6, 128, 95, 79]. This design has a limited tasks throughput, but allows low cost reactive scheduling crucial for straggler mitigation. In the second strategy, each worker installs a distributed data flow that generates and executes tasks locally [24, 115, 94, 22]. This design allows orders of magnitude higher task throughput, but any schedule change (e.g. migrating few tasks among workers) induces significant runtime overhead for installing new dataflow on

each worker node. Figure 2.2 shows the relative position of prior frameworks in the design scope with respect to task throughput and dynamic scheduling cost. In the following, we explore the details of each control plane design point in the context of example systems.

Centralized Controller

Systems such as MapReduce [48], Hadoop [6], CIEL [95], Optimus [79], and Spark [128] keep the entire control flow of a job 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. However, centralization limits the tasks throughput to few thousand tasks per second [99]. Generating and transmitting each tasks to workers is bound by the computation power for dependency analysis at the controller, and networking bandwidth for metadata transmission from controller to workers.

In MapReduce [48] (and for this matter Hadoop [6]), a centralized *master* partitions the *map* and *reduce* stages in to parallel partitions and assigns each task to a *slave*. Master also coordinates *sort and shuffle* stage among slaves by partitioning the intermediate key domain and assigning each sorting task to a slave. Masters throughput in generating and assigning the tasks is limited, however the centralization allows low cost tasks migration and straggler mitigation [26] by efficiently assigning backup slaves for the straggling tasks [130, 29].

CIEL [95] and Optimus [79] extend the centralized controller functionalities by adding dynamic control flow. In these frameworks, tasks can spawn other tasks to the controller for execution, and so in a sense, the task graph is generated dynamically during execution. This adds extra overhead at the controller for dynamically tracking and generating the tasks graph at the runtime. Spark [128] controller supports similar functionalities, and in addition, keeps track of a lineage for each data object in the system for efficient, fine-grained fault recovery. Upon failure, instead of rewinding back to a checkpoint, Spark controller reconstructs only the missing data objects on the failed worker by re-executing the tasks on the lineage. Spark controller handles only a few thousand tasks per second [99].

Execution templates, described in Chapter 4, borrow the same centralized control plane design strategy from prior systems to keep the dynamic scheduling and control flow in place. Also, execution templates add caching functionalities to a centralized controller to improve the limited task throughput by memoizing the task assignment decisions and caching tasks on the workers.

Distributed Dataflow

Systems such as MillWheel [24], Strom [115], Naiad [94], and TensorFlow [22] deploy a decentralized control plane. In these systems, each computing node installs a dataflow graph locally and generates and executes tasks independently. If there is synchronization required among nodes, they directly exchange data objects through lightweight messaging mechanisms. This design scales well and can support hundreds of thousands of tasks per second.

However, scheduling changes in decentralized systems have a significant cost. Since there is no centralized coordinator, any change requires stopping all nodes, installing new dataflow, and resuming execution from a checkpoint. As a result, in practice, these systems only deal *proactively* with the stragglers through two major techniques. In the first approach, stragglers are avoided through meticulous engineering [94]. However, this does not align with the notion of cloud as a service [3, 17, 13] where there is no direct access by the developers to the cluster hardware. The second approach launches backup workers for straggler mitigation [22]. This is not appealing from a cluster utilization perspective, as it requires at least doubling resources for synchronous operations.

Decentralized systems primarily emerged for low latency, stream processing applications. In MillWheel [24] and Storm [115], the stream processing systems deployed at Google and Twitter, the dataflow graph defines a set of computing nodes, each receiving a portion of the input stream. Application driver program explicitly defines the flow of streams among the nodes, such that there is no need for on-the-fly task assignments or coordination in the control plane. Fault recovery is handled through frequent and persistent checkpointing at the nodes.

In Naiad [94], the control plane supports loops in the driver program, as well.

Each message exchanged among nodes has a *timestamp*. The timestamp specifies the instance of the loop in the driver program. The control plane uses lightweight broadcast messages with monotonically increasing timestamps to mark the boundaries of each iteration of a loop. Nodes synchronize the execution according to the timestamps. Naiad does not support dynamic dataflow changes, so it tries to prevent stragglers by meticulous engineering. For example, Naiad's deployment at Microsoft data centers [94] disables TCP's Nagle's algorithms (for delayed acknowledgments) and reduces the default TCP acknowledgement timeout. These help improve the throughput for the bursty traffic during broadcasts. Also, Naiad implementation is hand tuned to minimize object allocations such that the garbage collector is activated less frequently. Note that the garbage collector could suspend the process, which leads to random stragglers in the cluster.

TensorFlow [22] turns the driver program in to a dataflow graph depending on the available resources. The *Master* node translates the cross node dependencies to a pair of send/receive tasks on the nodes. Also, TensorFlow allows data dependent branches through *multiplexing nodes* where every node takes a same branch based on a data value. Master could launch backup workers for each computation. The backup workers help with the tail latency. For asynchronous operations, the computation proceeds with only a subset of the fastest workers. For synchronous operations, at each stage, the result from the faster node between each primary and backup worker is used for the following stages.

Execution templates borrow the idea of installing dataflow 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 installed dataflow in place.

2.2 Cluster Scheduling Systems

Cloud schedulers (also called cluster managers) schedule tasks from many concurrent applications across a cluster of worker nodes. Each instance of an application runs as a *job* in cloud computing frameworks. A cloud framework could run multiple

concurrent jobs, and many cloud frameworks could share the resources in a cluster. Cloud schedulers have global knowledge of all of the jobs in the cluster, and control placement of each task on the worker nodes. Schedulers efficiently multiplex jobs across resources to improve cluster utilization [117, 67, 63], improve job completion time [54], fairly allocate resources across jobs [58], follow other policies [36, 50, 99], or allow multiple algorithms to operate on shared state [109]. There has been extensive work on the cluster schedulers, and there are numerous systems varying from system architecture or scheduling policy standpoints.

2.2.1 Scheduling Architectures

Initial cloud schedulers are centralized. In systems such as Mesos [67] and YARN [117], a centralized scheduler monitors the entire cluster and places tasks on the worker nodes for executions. Implementing scheduling policies such as job priorities, fairness, and resource allocation is straightforward in centralized schedulers with the global knowledge and control over the entire system. However, the limited task throughput of this architecture is a bottleneck at scale.

Increasing task rates from in-memory analytics workloads and high, aggregate task throughput from many concurrent jobs [97] have led some systems to take a distributed scheduling architecture. Sparrow [99] introduced the first distributed scheduling model. In Sparrow, jobs could connect to any scheduling node from a pool of schedulers. Sparrow schedulers are stateless, making it straightforward to scale by elastically adding or removing nodes. Each scheduler monitors the cluster independently; however, they still make good cooperative scheduling decisions based on mechanisms and principles derived from the power of two choices [92].

To optimize cluster utilizations with better global decisions, more recent systems have taken a hybrid approach. Tarcil [50] 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. Hawk's hybrid approach centrally schedules long-running jobs for efficiency and distributes short job scheduling for low latency [49]. Finally, Mercury [78] allows multiple schedulers to

request resources ("containers") from a shared pool and then schedule tasks on their resources independently.

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 for a *single* job. As we describe in Chapter 3, with optimized data analytics, the control plane of cloud frameworks become a bottleneck even for a single job at scale.

2.2.2 Scheduling Policies

Cluster schedulers implement various scheduling policies including min-cost flow computations [59, 73], packing [63, 64], or other algorithms [36, 67, 78, 109]. This dissertation does not examine the question of scheduling policy. Instead, it provides solutions to enable a control plane to support high task throughput with fine-grained scheduling decisions. Such a control plane could then benefit from any of the aforementioned scheduling policies. Scaling the control plane is orthogonal to the scheduling policies.

2.3 High Performance Computing

High performance computing (HPC) embraces the idea that an application should be responsible for its own task scheduling, as it has the greatest knowledge about its own performance and behavior. This approach is motivated by the very different scale and cost of large-scale supercomputers; when a job uses weeks of time on a multi-million dollar machine [15], it is worth demanding more programmer effort to tune and optimize performance.

HPC systems stretch from very low-level interfaces, such as MPI [113], 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 [68]. Extensive developer effort is required for a correct and scalable MPI

implementation, e.g. synchronizing between processes by exchanging messages, and mapping computation workload to processes evenly.

Charm++ [76] resembles MPI, but decomposes processes into smaller object-oriented units called *chares*. In Charm++, there is a clear distinction between sequential and parallel objects. This helps programmers have a clear knowledge of the cost of each operation and to account for expensive remote data exchanges needed for parallel computations. Also, it enables modular code development. For example, a parallel object interface could invoke various sequential implementations of a given algorithm with a same interface.

Parallel programming systems such as Legion [32], X10 [43], Chapel [41], and Uintah [55] allow developers to describe computations as a sequence of tasks, similar to cloud computing frameworks. This decouples the control flow, computations, and communications, such that the runtime could automatically infer parallelism, and opportunistically mask the communication overhead with computations. Legion [32] uses an asynchronous task model, with explicit dependency and coherency models over *logical regions*. Logical regions are encoded as tables over index spaces for efficient referencing in each task's metadata. Legion allows dynamic task spawning and provides a *mapping* interface for each task such that the application could control where each task executes in the cluster.

2.3.1 Mechanisms vs. Policies

HPC systems provide powerful abstractions to decouple control flow, computation and communication, similar to cloud computing frameworks. Their fundamental difference, however, is that HPC systems only provide mechanisms; applications are expected to provide their own policies. Specifically, in HPC systems, application developers can tradeoff programmer productivity with application specific customizations in favor of performance. On the other hand, cloud frameworks provide a generalized execution model for easier code development that works well in many cases. However, they leave almost no flexibility for application specific tunings. We elaborate on these different approaches in the context of two major functionalities: task scheduling and

task spawning.

Advanced HPC systems provide *mapping* interfaces for task scheduling in the cluster. For example, Chapel [41] expresses the computations over domains and sub-domains and allows fine-grained task placement over the hardware through the abstraction of *locales*. Locales are flat arrays of abstract locations used to map domains to hardware. In Chapel, application correctness depends on proper implementation of mappings by the developers. In contrast, in cloud frameworks the control plane takes care of task scheduling and also enforces correctness. Legion [32], decouples correctness from scheduling, and provides powerful task stealing mechanisms. However, it requires application developers to deal with subtle corner cases of load imbalance or machine failure. In contrast, cloud frameworks provide fault recovery and straggler mitigation, automatically.

Task spawning interfaces in HPC systems expose fine-grained dependency among tasks. This helps infer maximum parallel execution opportunities at the run time. For example, in Legion [32], application developers specify task dependencies at the granularity of the logical regions each task accesses. In addition, they explicitly express privilege (e.g. read or write access) and coherence (e.g. atomic or concurrent access) properties. This could be tedious for application developers. This has led to the development of various domain specific languages (DSL) [112, 34], which leaves the burden of learning new DSL interfaces on the shoulders of application developers. In contrast, cloud frameworks introduce a few general task dependencies (e.g. narrow or wide) through simple high level data-parallel operations (e.g. map and groupBy), but lose many application specific optimizations. For example, the opaque key-value interface of cloud frameworks does not suite applications with geometric localities (e.g stencil operations).

Chapter 3

Control Plane: The Emerging Bottleneck

This chapter starts by describing the common characteristics and requirements of data analytics workloads. It then evaluates the scope and limits of CPU performance improvements through a concrete case study. It shows that, with speedups in tasks, once CPU-bound workloads are now control plane bound and the bottleneck exacerbates with further CPU performance improvements. The chapter further analyzes the performance characteristics of in-memory computations, concluding that the results are not just limited to the specific case study. Last, to further motivate the problem, it introduces a new workload, graphical simulations, that would require orders of magnitude higher task throughput compared to traditional analytics workloads, even at moderate scales. Chapter 7 shows how these applications run in Nimbus for further evaluations.

Since cloud computing frameworks gained traction among engineers and data scientists [48, 6], there have been significant efforts in the academia and industry to improve performance of these systems on multiple fronts: networking platforms [25, 44, 75, 105], in-memory computing and disk I/O efficiency [28, 128, 107], and scheduling and straggler mitigation techniques [130, 29, 27, 99]. These efforts culminated with an accepted fact in the community that the advanced analytics workloads had become CPU-bound [98], meaning that job completion time is mostly

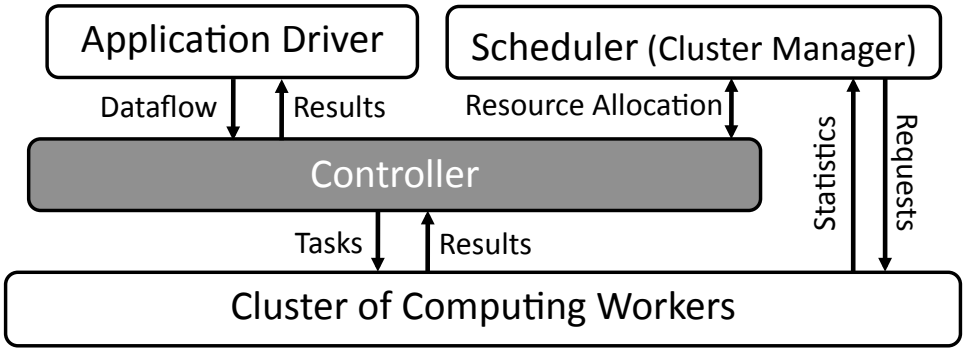


Figure 3.1: Control plane in a cloud computing system: it interacts with the application driver to execute application tasks on a cluster of workers. Control plane could also be connected to the cluster manager for resource allocation purposes.

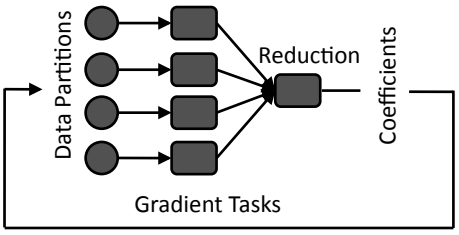
bound by the time worker nodes spend on computations. Specifically, the measurements showed that improving the network and disk performance could reduce the job completion time of modern analytics workloads with a median of at most 19% [98].

Recently, there has been a wave of research on improving the CPU performance of analytics workloads such as code generation optimizations [1, 127, 101, 23], and cross function data access optimizations [102]. The results are promising, showing orders of magnitude speedups for common benchmarks [121]. The performance improvements are not surprising [91], as widespread frameworks had focused on programmer productivity by providing APIs in higher level and managed programming languages, trading off performance for simplicity [128, 129, 31, 9]. As it is shown in this Chapter, there is, still, room left for improvements.

This dissertation argues that control plane is an emerging bottleneck in modern analytics workloads. Control plane translates the driver program of a job into smaller tasks and assigns them to the worker nodes, as depicted in Figure 3.1. Speedup in computations pushes control plane to process tasks at much higher rates to keep the workers busy. Specifically, with tasks as long as a few milliseconds, only a handful of workers could execute thousands of tasks per second. This is well beyond the task throughput of available frameworks with dynamic scheduling features [99]. As a result, the application completion time becomes control bound, meaning that the progress is limited by the speed of control plane processing tasks.

```
// Data dependent branch
while (gradient > threshold) {
   // Gradient decent algorithm
   gradient = Gradient(data, coeff)
   coeff += gradient
}
```

(a) Driver program pseudocode.



(b) Iterative execution graph.

Figure 3.2: Task graph and driver program pseudocode of logistic regression algorithm. It uses the gradient descent algorithm for optimizing the feature coefficients. The optimizer loop continues until the gradient drops below a certain threshold. `Gradient` is a parallel operation that executes parallel tasks on data partitions.

3.1 Data Analytics Applications

Cloud computing workloads are increasingly advanced data analytics workloads [119], including machine learning [45, 108, 87], graph processing [100, 30], natural language processing [37, 116], speech/image recognition [69, 47], and deep learning and neural networks [90, 66]. These applications are usually long running and iterative in nature. For example, they run a loop until computations converge. To capture these features effectively, most of the recent cloud frameworks support nested loops and data dependent branches in their API [79, 128, 22].

As a running example in this Chapter, we consider logistic regression, a common data analytics benchmark [53]. Figure 3.2 shows the pseudocode and task graph for this algorithm. It uses the iterative gradient descent optimizer for calculating the feature coefficients until the gradient drops below a certain threshold. Given the driver program in Figure 3.2(a), the cloud framework generates the task graph as depicted in Figure 3.2(b). The `Gradient` operation is partitioned into many parallel tasks, each computing the gradient on a partition of the data. It is followed by a global reduction to sum up the partial gradients from each partition. The goal of a cloud framework is to not only compute individual tasks faster, but also to minimize the completion time of the entire job.

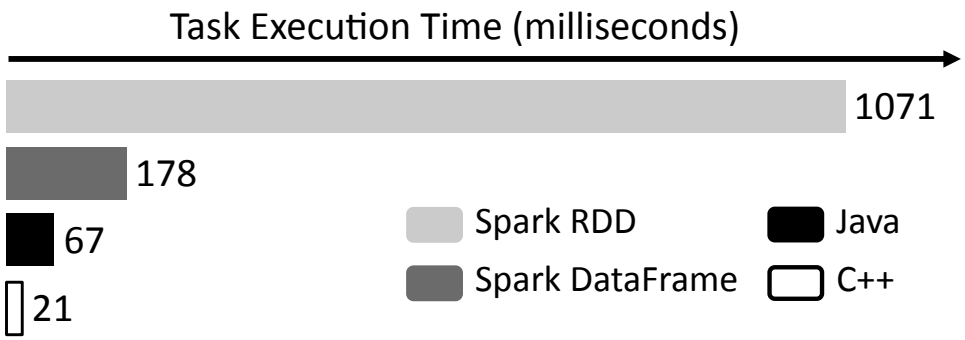


Figure 3.3: Execution time of a single gradient task of logistic regression implemented in Spark's Scala RDD, Spark's Scala DataFrame, Java and C++. While DataFrame optimizations improve the original RDD performance, it is still more than 8 times slower than C++. The data size is 64MB, and the results are averaged over 30 iterations excluding the first iteration to allow JVM to warm up and JIT compile.

3.2 Optimizing Data Analytics

Data analytics that are implemented in high-level and managed programming languages used by many frameworks leave a great deal of room for performance optimizations. As a concrete example, Figure 3.3 shows the execution time of a single gradient task in logistic regression algorithm depicted in Figure 3.2, implemented using 1) Spark's classic RDD interface in Scala [128], 2) Spark's newer DataFrame interface in Scala [121], 3) Java, and 4) C++. The execution time shows a single iteration of the algorithms over the input set of size 64MB. The numbers are averaged over 30 iterations and the initial iteration is excluded to allow Java Virtual Machine (JVM) to warm up and just-in-time (JIT) compile. As you can see, the C++ implementation runs *51 times faster* than the classic Spark implementation in Scala. The Spark's DataFrame interface benefits from the latest tungsten engine [1] and whole-stage code generation techniques [23], but it is still slower than direct Java, and even further away from C++ implementation.

A push for greater programmer productivity has led many cloud computing frameworks to support higher-level languages [128, 31, 9, 129]. For example, Spark [128] provides program interfaces in Java, Scala, and Python. However, Scala's concise and handy interface is favored by users: 70% of the Spark applications are written in

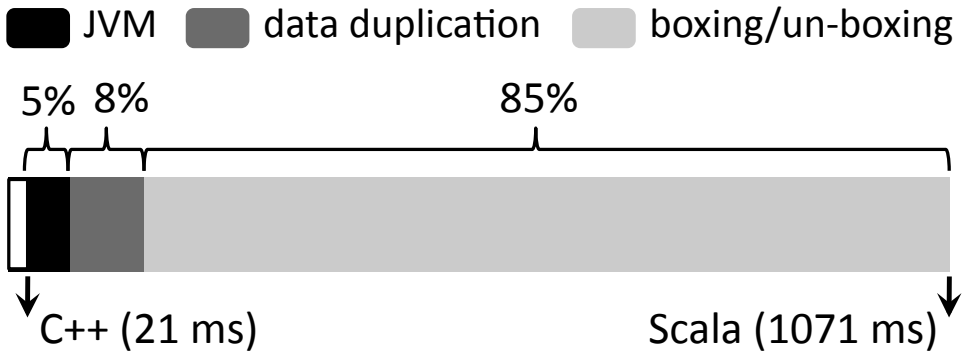


Figure 3.4: Breakdown of the contributing factors in the performance difference between Scala and C++ implementation of logistic regression. These results are obtained by decompiling the JVM bytecode, inspecting it, and removing the sources of overhead, step-by-step, until the code matches the direct Java implementation. These results are averaged over 30 iterations and discard the first iteration to allow JVM to warm up and JIT compile.

Scala [127]. According to a survey, conducted in 2015, 88% of the Spark developers picked Scala as their favorite development language [2]. This simplicity is one of the main reasons Spark has gained significant traction among data scientists, and has become the most active open source project in big data with more than 280 individual contributors [126]. Despite simplicity, the inefficiency of higher-level languages becomes significant for in-memory computations.

3.2.1 Where Do the Cycles Go?

Figure 3.4 shows the breakdown of the sources of inefficiency in Scala implementation of RDDs compared to C++ and Java. Note that the Scala code is translated into intermediate Java bytecodes, which is then JIT compiled and run by a JVM. To determine the sources of inefficiency, we configured the JVM to output the JIT assembly and inspected it. We inserted performance counters in the Scala code, and reinspected the assembly to verify they captured the correct operations. To separate different sources of overhead, we decompiled the JVM bytecodes that Scala generated back into Java, rewrote this code to remove its overheads step-by-step, recompiled it, and verified that the computational operations remained unchanged.

The poor performance of the Spark's Scala RDD implementation has three major causes. First, since Scala's generic methods cannot use primitive types (e.g., they must use the `Double` class rather than a `double`), every generic method call allocates a new object for the value, boxes the value in it, un-boxes for the operation, and deallocates the object. In addition to cost of a `malloc` and `free`, this results in millions of tiny objects for the garbage collector to process. 85% of logistic regression's CPU cycles are spent boxing/un-boxing. Second, the immutable characteristic of the Spark's RDD forces methods to allocate new arrays, write into them, and discard the source array. For example, a `map` method that increments a field in a dataset cannot perform the increment in-place and must instead create a whole new dataset. This data duplication adds an additional factor of $\approx$ 2x slowdown. Third, using the Java Virtual Machine has an additional factor of $\approx$ 3x slowdown over C++. This result is in line with prior studies, which have reported 1.9x-3.7x for computationally dense codes [70, 57]. In total, this results in RDD code running 51 times slower than C++.

The DataFrame interface bypasses the intermediate Java objects and directly operates on binary blobs of data [31]. This removes the boxing/un-boxing overhead; however, it still has the data copy and JVM overhead compared to a direct C++ implementation.

3.3 Control-Bound Data Analytics

Cloud computing frameworks decompose data parallel jobs in to many smaller computation units, called tasks. Input dataset is partitioned into pieces, one for each task. This allows the frameworks to speedup the job completion time through parallelism. This is the common notion usually referred to as *strong scaling*: increasing the compute power for solving a fixed size problem. The degree of parallelism depends on the number of available computing slots on the workers, for example, the number of CPU cores or GPU units allocated to the job by the cluster manager [67, 117]. Also, straggler mitigation techniques usually recommend partitioning a job into more tasks compared to the available computing slots [97]. For example, 10 partitions per core is a rule of thumb for task granularity in favor of straggler mitigation [46].

Figure 3.5: 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. This implementation uses Spark's DataFrame interface.

Speedup in individual tasks, does not necessarily result in reduction in job completion time. For shorter tasks, the control plane overhead in processing and assigning tasks becomes comparable to the task execution itself, and could dominate the job completion time. As a concrete example, we consider running logistic regression in a cluster of workers in Amazon EC2 [3] over a data set of size 100GB. Worker nodes use `c3.2xlarge` instances with 8 virtual cores and 15GB of RAM. Further information on the details of the experimental methodology is laid out in Section 6.1.

Figure 3.5 shows the performance of Spark 2.0's MLlib logistic regression running on 30–100 workers. The implementation uses the DataFrame interface. 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. In other words, as tasks are partitioned and spread over more cores for strong scaling, controller cannot keep up with the execution speed, and workers fall idle waiting to receive tasks from the controller. We measured Spark's controller to be able to issue ≈6,000 tasks per second. Execution over 100 workers in Figure 3.5 requires more than 23,500 tasks per second.

These results show that with the task performance improvements, once CPU-bound analytics workloads have now become control-bound. For example, Figure 3.6

Figure 3.6: Effect of running logistic regression with optimized tasks in Spark. Spark-opt shows the case where computations are replaced with spin-wait as fast as C++ tasks. Although Spark-opt tasks can run 51 times faster than RDD implementation, a job using C++ tasks on 100 nodes only runs 2x faster: much slower than the expected speedups. This is because the control plane becomes a bottleneck, wasting 97% of the iteration time.

shows the results of implementing logistic regression with the classic RDD interface that does not have the DataFrame optimizations. In this case, the tasks are long enough that controller can keep up with the task execution rate, and the majority of time is spent at the workers. Increasing the number of workers from 50 to 100 increases the compute power and helps reduce the job completion time. This aligns with the previous work that found analytics workloads had become CPU-bound [98].

However, for the DataFrame implementation, control plane is the clear bottleneck: 80% of the run time is wasted at the controller when running on 100 workers. In this case, increasing the number of workers hurts the job completion time due to the control plane being unable to schedule tasks fast enough.

If the tasks where to run faster, the control plane overhead would take even a bigger portion of the run time. To this end, we consider a hypothetical scenario where Spark tasks run as fast as a C++ implementation. To emulate the C++ tasks, we replaced the computation blocks of the Spark driver program with a spin-wait as long as the C++ task. This represents the best-case performance of Spark calling into a native code. The results are marked as *Spark-opt* in Figure 3.6. As you can see,

97% of the iteration time is wasted at the controller when running on 100 workers.

Task optimization in Spark does not necessarily translate to a reduction in job completion time. For example, while the computational tasks run 51 times faster than RDD implementation, on 100 nodes the overall job runs only 2 times faster. Worker nodes spend most of the time idle because the central Spark controller cannot schedule tasks fast enough. Each core can execute 250 tasks per second (each task is 4ms), and 100 nodes (800 cores) can execute 200,000 tasks per second, well beyond what Spark controller can handle, ≈6,000 tasks per second.

3.3.1 Why Not Distribute the Control Plane?

As explained in Section 2.1.2, frameworks with distributed control planes [22, 94, 115, 24] remove the central decision maker and can execute hundreds of thousands of tasks per second. However, the lack of a single coordinator makes dynamic scheduling at runtime very expensive or even impossible. A single tasks migration between two nodes could take as high as few hundred milliseconds, orders of magnitude longer than the task itself. The specific details of why that is the case is framework dependent, but fundamentally it stems from the fact that any changes in the scheduling without an omniscient node requires expensive coordination procedures among all the nodes. As a specific example, we explain the details of scheduling in Naiad [94].

In Naiad, nodes send and receive date messages with an associated key. Since there is no centralization, there is a key-node mapping, such that every node could route messages independently. At the beginning of a job, driver program is first executed completely in the "logical" mode; it is called logical since it does not trigger any computations. During the logical execution, a dataflow is generated based on the key-node mapping on each node. The dataflow specifies the computations and communication channels among nodes.

A scheduling change in Naiad would essentially mean changing the key-node mapping. Note that even a small change could potentially affect the dataflow on every node. For example, if a reduction operation is migrated, then every node needs to

transmit partially reduced data to a new destination node. Without a central controller, there is no way for isolated nodes to comprehend and account for the effects of scheduling changes on the dataflow of the other nodes. As a result, any scheduling change requires rerunning the driver program in the logical mode and regenerating the dataflow on all the nodes. We have measured that it takes ≈230ms to generate a simple, one-stage dataflow on Naiad nodes. This means even a single task migration would induce ≈230ms latency.

3.4 Other Control-Bound Applications

This section shows that for many applications, the required task throughput is well beyond what a centralized controller can handle. First, it considers a general scenario based on a common application setup in data analytics. Then, it evaluates the performance characteristics of an existing graphical simulation library, as a concrete example of high performance applications.

First, we provide a simple back-of-the-envelope calculation of why efficient implementation of algorithms that operate on reasonably sized inputs and outputs have tasks in the order of a few milliseconds. We consider a conservative case, where there are only two variables, an input and an output. We use Amazon C3 instances as a representative memory to CPU core ratio of 1.9GB of RAM per core.[1] A `c3.2xlarge` instance, for example, has 8 virtual cores and 15GB of RAM. A well-designed workload typically breaks each core's work into ≈10 parallelizable partitions [46, 97]. For example, an 8-core node runs 80 instances of a task. We consider a simple task with a single input and a single output – a filter that reads integers and outputs the values above a threshold. Assuming that the OS and program use absolutely no memory and the system has no other data, the maximum size the input and output can be is 93.8MB (15GB/80/2). Depending on its selectivity, a C++ filter over 93.8MB takes 16–30ms on a modern processor.

[1] At time of writing, Amazon R3 instances provide a higher memory-to-core ratio, but are more expensive per core ($0.0875/hour rather than $0.0525/hour for each core), so a given CPU-bound task will cost more to compute.

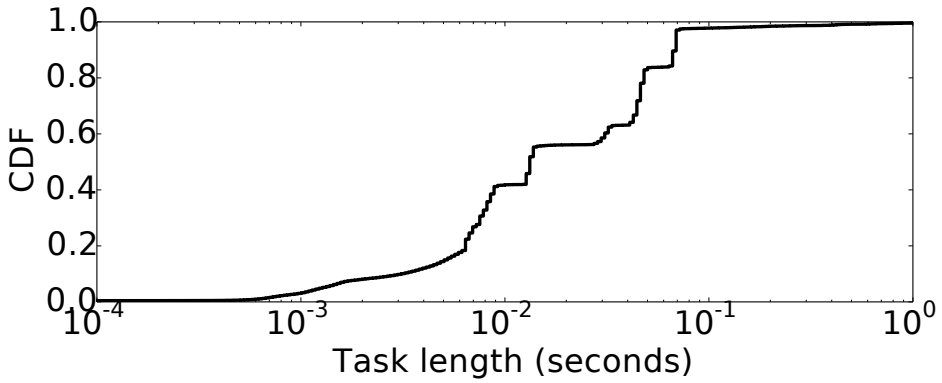


Figure 3.7: Task length distribution for a canonical water simulation. An example application developed to run fast rather than fit the constraints of cloud frameworks.

To give a sense of how long tasks in optimized, computationally bound workloads can look like, we consider PhysBAM, an open-source computational fluidics library, used by movie and special effects studios [51]. We use PhysBAM because it is complex, open source, the result of several person-decades of development effort, used significantly in production, and an example of a system that has been designed to run fast, rather than fit the performance tradeoffs of cloud systems. A single time-step of a PhysBAM simulation involves 26 computational stages operating over 40 different variables. Currently, graphical simulation software packages are designed to run on a single powerful server or small, 3–4 node high performance computing clusters. Being able to run them in the cloud allows researchers to benefit from the elastic scalability of the cloud providers to speedup computations. These applications are long running, and hence require dynamic scheduling during the runtime for reactive load balancing due to stragglers or failures in the cloud.

Figure 3.7 shows the task duration of a PhysBAM water simulation with a size of 512^3-cells running over 8 workers (64 cores), with one partition per core. While most CPU time is spent in 65–70ms tasks, the median task length is 13ms. Some tasks, such as computing whether to execute the next iteration, have a maximum execution time of hundreds of microseconds.

Applications with millisecond tasks easily saturate the controller even at moderate scales. For example, with 10ms tasks, even a cluster with 10 workers (8 cores each) can

Simulation Size	#Worker (#Core)	Task Rate	
		Main Loop	Implicit Solver
512^3-cell	8 (64)	63Kps	92Kps
1024^3-cell	64 (512)	394Kps	463Kps

Table 3.1: Task rate of a canonical water simulation for two different experimental setups. Simulation has a main loop including explicit solvers and an inner loop for an implicit solver. Task rates are reported separately for the implicit solver and the main loop. Note that solver iterations have higher rate as the tasks are shorter.

execute 8,000 tasks per second, already beyond Spark's task throughput (measured at 6,000 task per second in Section 6.3.2). As another example, Table 3.1 shows the task rate for a canonical water simulation in two different settings. As you can see, a 512^3-cell water simulation distributed over 8 workers (64 cores) executes up to 92,000 tasks per seconds. If we were to scale out a bigger, 1024^3-cell simulation over 64 workers (512 cores) the task rate could go as high as 463,000 task per second.

3.5 Conclusion

As cloud systems optimize their CPU performance, they will have to deal with tasks that are in the range of milliseconds or even microseconds. Specialized software and hardware for computations on GPU [12, 22] will speedup the tasks even further. The general trend shows a tremendous potential speedup for individual tasks. These short tasks lead to a much higher task rate than current centralized controllers can support. To keep the low cost, and dynamic scheduling provided by centralized frameworks, there is a need for new control plane architecture. In the next chapter, we introduce execution templates that increase the task throughput of centralized controllers by orders of magnitude. This removes the control plane bottleneck such that individual task speedups result in significant reduction of overall job completion time.

Chapter 4

Execution Templates

This chapter introduces execution templates as a novel abstraction for the control plane. Execution templates are based on the observation that an increasing portion of advanced analytics workloads are iterative in nature [119]. For example, machine learning and graph processing algorithms usually deploy iterative optimization techniques [87, 108, 111, 100]. Different iterations execute similar tasks with minor differences. This results in repetitive patterns in the control plane that are saved in templates. Instead of generating and assigning the tasks from scratch for each iteration, templates are instantiated by loading few changing parameters. As we will evaluate in Chapter 6, this improves the control plane performance by orders of magnitude for optimized analytics.

An execution template is a parameterizable block of tasks that is installed on the nodes during the runtime, and instantiated by filling in new parameters. Despite the fixed structure of templates, they support dynamic scheduling and dynamic program control flow. In addition to installation and instantiation, execution templates provide two special mechanisms: edits and patching. Edits allow fine-grained updates in the template to react to the scheduling changes. Patches enable controller to match the state of workers to the fixed structure of the templates despite dynamic control flow of the program. As we will see in Chapter 7, execution templates are general enough to not only support traditional data analytics, but also complex applications with nested loops and data dependent branches.

The chapter starts by describing the execution model of the execution templates and their requirements. It continues by elaborating on the repetitive patterns in the control plane as a basis for the design of execution templates. It then introduces the execution templates abstraction and the mechanisms they provide to increase task throughput without sacrificing dynamic scheduling or dynamic program control flow.

4.1 Execution Model

Execution templates require a centralized control plane architecture. As depicted in Figure 4.1, a centralized controller harnesses a cluster of workers allocated by the cluster manager to perform the computations defined by a driver program. Execution templates operate on the controller and its interfaces.

A driver program specifies the application logic for the controller. The details of the driver program interface is framework dependent, however, at a high level, it provides a lineage of computations, the input and output for each computation, and their ordering. For example, Figure 4.1 shows an example driver program with two stages: the `map` stage operates on the input data and produces intermediate results that are fed in to the `reduce` stage.

Controller transforms the driver program into individual tasks for execution on the workers. For data parallel computations the data set is partitioned in to smaller pieces for parallel execution of the tasks. Not all tasks can run in parallel due to data dependencies enforced by the driver program. To this end, the controller translates the driver program into a *task graph*: a directed acyclic graph (DAG) with tasks as vertices and task dependencies as edges. For example, Figure 4.1 shows a sample task graph where data is partitioned into 4 pieces to perform the `map` stage in 4 parallel tasks, followed by a single global reduction.

The controller partitions the task graph among the workers. For example, in Figure 4.1 data partitions are split between the workers, hence the controller splits the `map` tasks among them. The controller generates a per worker *execution plan* for each worker. The execution plan includes the regular tasks in the task graph, as well as additional *copy tasks* for data exchanges among the workers. The copy

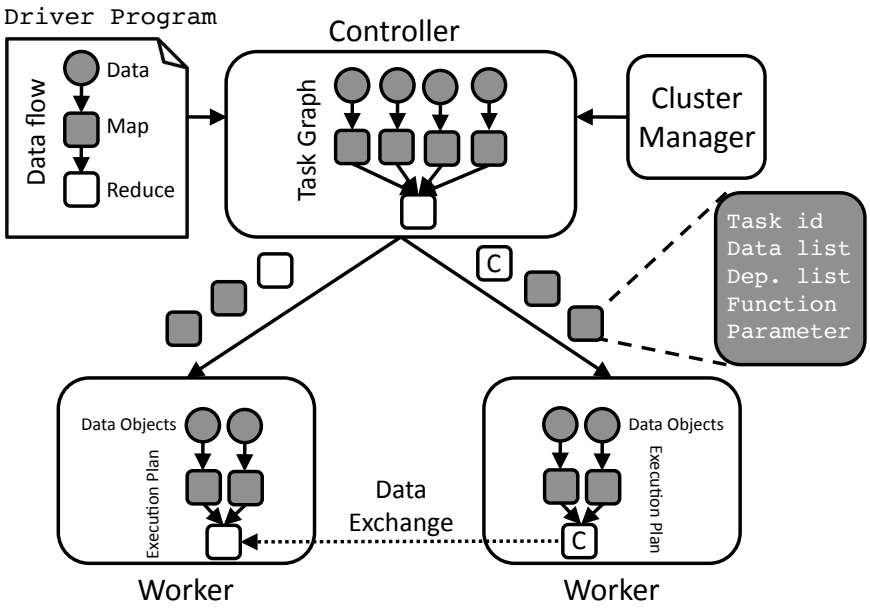


Figure 4.1: Generalized execution model assumed by execution templates. A driver program specifies the application logic to a centralized controller. The controller turns the driver program into a data parallel task graph and partitions the tasks among a cluster of workers for execution. The controller generates a per worker execution plan that includes the tasks and data exchanges among the workers for global operations such as reductions. A cluster manager controls available resources and can add/remove workers to/from the cluster, dynamically.

tasks explicitly specify the data object, the sender, and the receiver worker such that workers can directly exchange data without looking up the controller. For example, Figure 4.1 shows that controller assigns a copy task to the right worker to transmit the intermediate `map` results to the left worker, which then performs the global reduction.

The controller assigns the tasks to the workers for execution, along with a metadata. The metadata, among other things, includes the tasks dependencies. Workers use this metadata to queue the tasks and execute them in the correct order. For example, in Figure 4.1, the controller sends a batch of task metadata to the workers, and workers create the execution plan locally for correct execution.

4.1.1 Framework Requirements

Conceptually, execution templates can be incorporated into many existing cloud frameworks. Incorporating them, however, assumes certain properties in the framework's control plane to satisfy their execution model as drafted above. There are three major requirements:

1. Controller partitions and schedules the task graph at the granularity of individual tasks. Also, workers can receive and execute individual tasks. Fine-grained tasks are a prerequisite to support fine-grained, dynamic scheduling; they define the minimum scheduling change that a system can support.

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. Controller conducts global operations that require data exchanges among workers proactively by sending copy tasks, instead of reactively responding to data lookups by the workers.

3. Workers maintain a queue of tasks and locally determine when tasks are runnable. Workers can receive a batch of tasks from the controller, 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. This requires the workers to hold and manage local state for proper execution.

Depending on the framework, satisfying all these requirements might need few simple or drastic changes in the control plane design of available frameworks. The implementation details depend on the specifics of the underlying framework. For example, since Spark [128] workers are stateless, deploying execution templates requires implementing the task queue and local task graph for the workers. Also, Spark workers resolve data exchange dependencies through reactive lookups at a centralized block manager at the controller, while execution templates require local metadata for direct data exchanges among the workers.

4.2 Repetitive Patterns

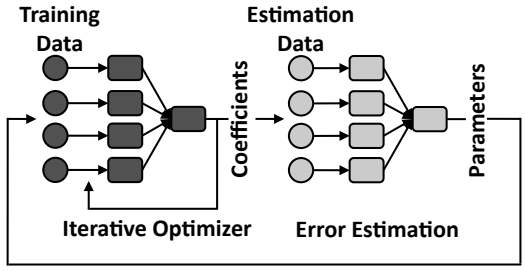
A great portion of cloud workloads is long running and iterative. Cloud computing applications are increasingly advanced data analytics including machine learning [45, 108, 87], graph processing [100, 30], natural language processing [37, 116], speech/image recognition [69, 47], and deep learning and neural networks [90, 66]. These applications are usually implemented on top of frameworks such as Spark [128] or Naiad [94], for seamless parallelization and elastic scalability. A recent survey [10] of Spark users, for example, shows 59% of them use the Spark machine learning library [9]. Efforts such as Apache Mahout [8] and Oryx [19] provide machine learning libraries on top of Spark. Cloud providers, in response to this need, now offer special services for machine learning models [18, 4].

One important property of advanced analytics is that they are iterative in nature: they execute a loop (or set of nested loops) until a convergence condition. For example, Figure 4.2 shows the pseudocode and task graph for a training regression algorithm called cross-validation technique [53]. The data set is split into training and estimation sets, hence the name cross-validation. The algorithm consists of a nested loop. In the inner loop, the training data is fed to an iterative optimizer, e.g. gradient descent, to optimize the feature coefficients. These coefficients are then used in the outer loop for prediction over the estimation data. Depending on the estimation error, the model parameters are updated for another iteration of the algorithm. The loop continues until the estimation error drops below a certain threshold. In addition, advanced analytics are long-running, meaning that the loops run for many iterations. For example, deep neural networks might run for several weeks over the production data sets to generate the optimized link weights.

The execution template abstraction is motivated by the fact that the iterative nature of advanced analytics results in repetitive patterns in the control plane. Different iterations generate and run the same DAG of tasks with few minor changes. For example, the `Gradient` and `Estimate` operations in Figure 4.2 can each generate many thousands of tasks per iterations in the task graph. However, the task graph structure is identical for each iteration, except that the same vertex in two iterations can have

```
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 4.2: 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.

different values across iterations, such as the `coeff` and `param` parameters. Furthermore, task identifiers change across iterations. Instead of processing the tasks from scratch for each iteration, control plane can leverage the fixed structure to improve the performance.

Many systems have observed and leveraged this iterative nature for various purposes. For example, Spark [128] caches the data sets in-memory to improve the performance when data set is referenced frequently, as in iterative applications. The Ernest system [119] leverages this observation for predicting the performance of the iterative analytics by sampling the performance of a few iterations and extrapolating the results to the entire execution lifetime. Execution templates take advantage of the repetition for caching the control plane decisions. Next, we introduce execution templates and how they capture these repetitive patterns to improve the performance of the control plane.

4.3 Abstraction

An execution template is a parameterizable list of tasks. It caches the repetitive segments of the task metadata and leaves the changing parts as parameters. The fixed structure of the template includes executable functions, task dependencies, and data

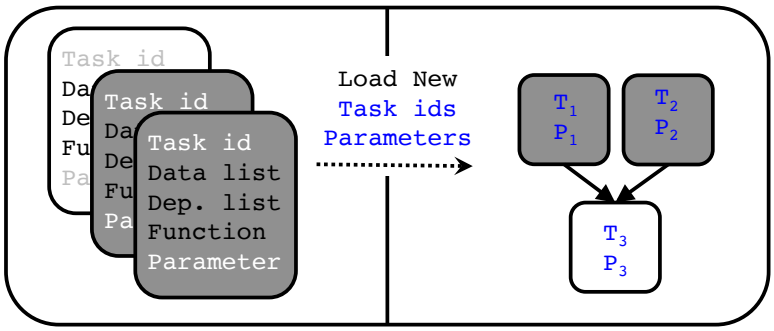


Figure 4.3: An execution template is a parameterizable list of tasks. The fixed part of the template includes the executable functions, task dependencies, and data access lists. A new batch of tasks are spawned by loading the parameters of a template.

access list. The parameter list includes the task identifiers and runtime parameters passed to each task. For example, Figure 4.3 shows how only by loading new task identifier and parameters a new batch of tasks are spawned from the template.

Execution templates are installed during the run time for the repetitive sections of the task graph. After installation, instead of generating and assigning the task from scratch, templates are instantiated with a single message that fills in the parameter list of the template. This way, a new iteration of all the tasks cached in the template execute with a single light message with the payload of only changing parameters.

Upon instantiation, not all the tasks in a template are immediately runnable. The dependencies in the task graph enforce an ordering among the tasks. Worker nodes need to properly order and execute the tasks within a template. To this end, the fixed structure of the template also includes the task dependencies in the form of a task graph DAG, as depicted in Figure 4.3. With the local task graph, workers could run a batch of tasks without receiving explicit synchronization signals from the controller, which could become a bottleneck in a centralized control plane.

4.3.1 Template Granularity

Larger templates are more efficient, since the template instantiation cost is amortized over a greater number of tasks. At one extreme, each individual task could be a template, which results in the lowest efficiency. At the other, the entire task graph of

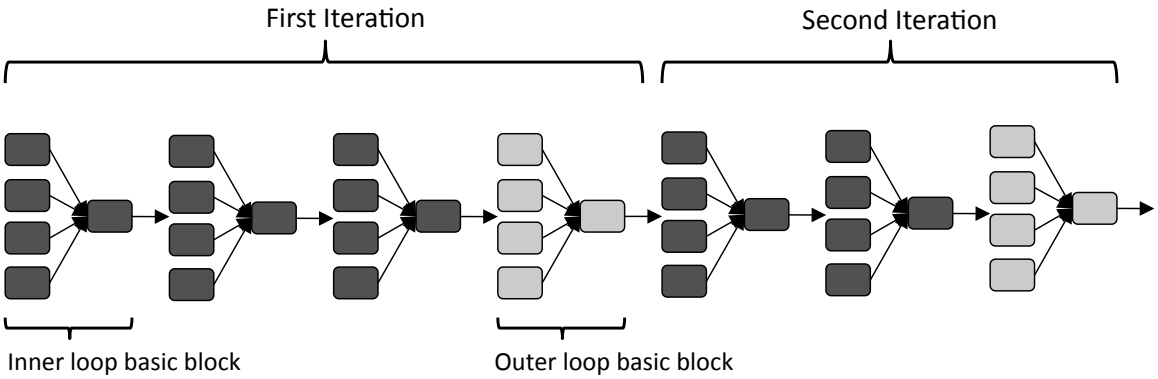


Figure 4.4: An actual run of the task graph in Figure 4.2. Depending on the computed data, the inner loop has different number of iterations in different instances of the outer loop. To keep the dynamic control flow, templates are installed and instantiated at the granularity of single basic blocks. This way, the number of times a basic block is executed could vary based on the computed data at runtime.

the driver program could potentially be one giant template. This case has the highest efficiency, as the entire task graph is executed with a single instantiation message.

However, templates cannot go beyond a branch in the driver program without constraining the control flow. To see why that is the case, note that once a template is instantiated all the cached tasks within the template are triggered for execution. This implies a *deterministic* control flow within a template. A driver programs with data dependent branches (e.g. conditional while loop or data dependent if/else statement) cannot be captured with a single template. In other words, loop unrolling and other batching techniques [118] cannot capture nested loops and data dependent branches.

For example, Figure 4.4, shows an actual run of the task graph in Figure 4.2. For the first iteration of the outer loop, the inner loop converges after three iterations. In the second iteration of the outer loop, the inner loop converges only after two iterations. The convergence rate depends on the computations and is not known in advance. Caching the entire iteration in a template only allows fixed number of inner loop iterations, and cannot differentiate between these two cases.

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 [103].

A basic block is the largest unit of execution in the driver program with deterministic control flow. Hence, generating templates at the granularity of the basic blocks maximizes the size of the templates without constraining the control flow.

For example, the task graph in Figure 4.2 has two basic blocks, one for the inner loop and one for the outer loop operations. Figure 4.4 shows the boundaries of these basic blocks. For each instance of the inner loop its according template is instantiated until convergence. Similarly, for every outer loop iteration the template for the outer loop basic block is instantiated.

4.3.2 Template Preconditions

Instantiating the execution template has an implicit assumption that the data objects in the data access lists are already in the local memory on the workers. Since a basic block can have multiple entry points, the data objects on the workers may not always match the data access lists. As a concrete example, the execution template for the inner loop basic block in Figure 4.2 needs to have the updated model parameters, `param`, on every worker. As depicted Figure 4.5, there are two cases in which an inner loop basic block is entered: 1) after itself, and 2) after the outer loop. In the fist case, `param` is inductively already on every worker. However, in the second case, `param` exists only on the node that reduced it.

We define *template preconditions* to be the list of data objects that are required for the template to start execution properly. The worker state needs to match the precondition list before template instantiation. The details of a precondition list is framework dependent. For example in case of a framework with immutable data model (e.g. Spark), it simply lists the required data objects. But in a framework with mutable data model (e.g. Nimbus), the preconditions further specify whether the latest update of the data is required or any copy of the data would suffice. For example, for the write-only operations (e.g. intermediate reductions) the initial state of the data is not important.

In frameworks with mutable data model, enforcing the preconditions of the templates are more complicated, since it requires tracking the updates to the data objects

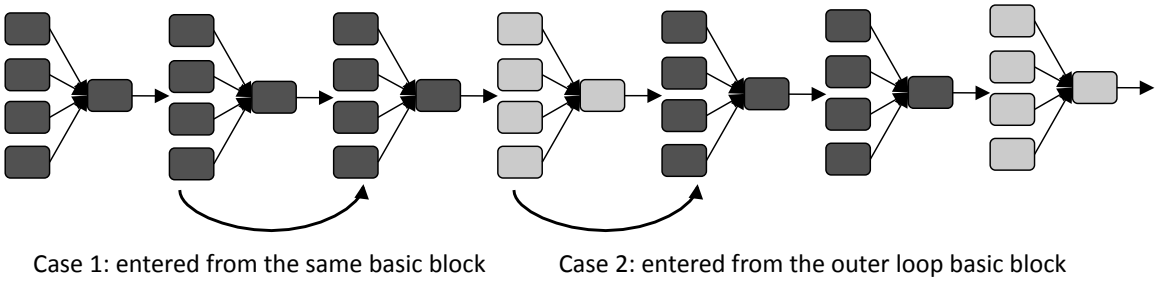
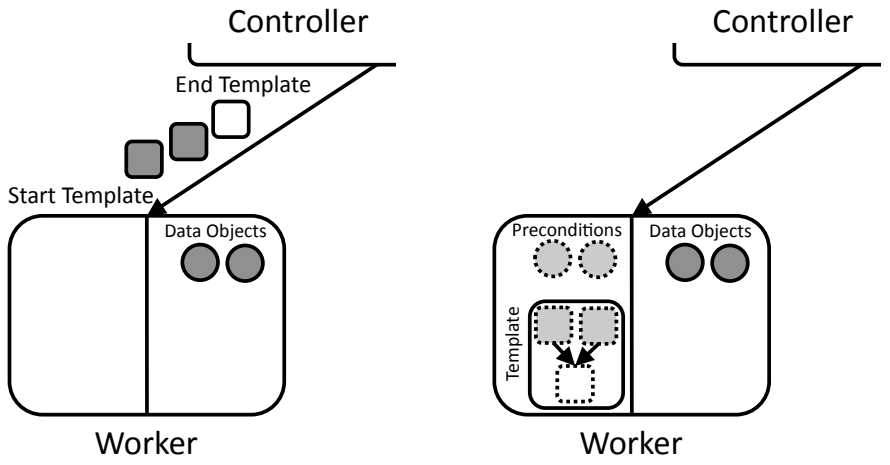


Figure 4.5: A basic block can be entered from different parts of a program. As a result the template preconditions may not hold in all circumstances. Here, the inner loop basic block can either follow itself (case 1), or the outer loop (case 2). The inner loop template requires the reduced `params` value to be on all the workers, but in the second case the updated value only exists on the reducer worker.

as well. However, they have two major benefits. First, for data intense applications they allow in-place operations. This is for example crucial for graphical simulations. Second, since the data ids are persistent beyond a single write operation, the data access reference could be cached as the fixed part of the execution template to reduce the payload size of the instantiation messages. The Nimbus framework introduced in Chapter 5 has a mutable data model.

4.4 Mechanisms

Execution templates are *installed* and *instantiated* at run time. These two mechanisms result in performance improvements in the control plane by caching and reusing repetitive control flow. At first glance, the fixed structure of the templates might seem inconsistent with the design goal of having fine-grained flexible scheduling. However, templates have an *edit* mechanism to deal with dynamic scheduling. Also, the fixed precondition list of a template may not match the worker state in all different entry points into a basic block. To this end, execution templates support a *patching* mechanism to handle dynamic program control flow. Each mechanism, and their semantics are discussed in the following.



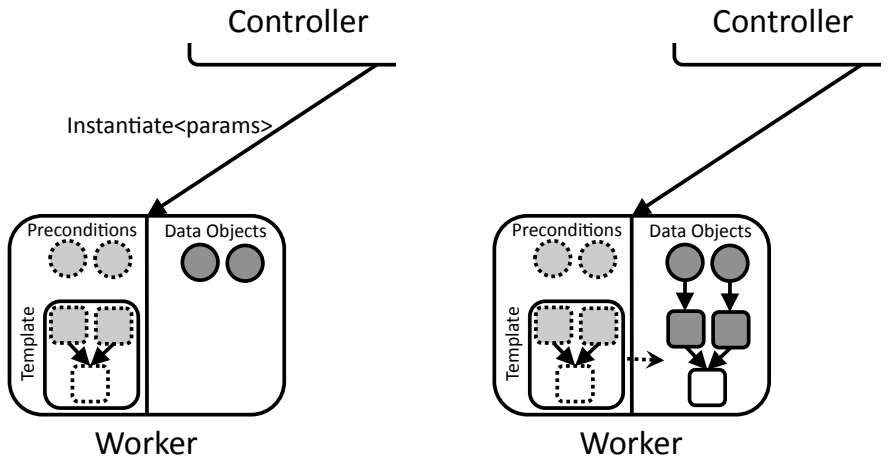
(a) Template start and end messages. (b) Installed templates.

Figure 4.6: Controller installs templates on the workers. The first time a basic block in the task graph is scheduled for execution, controller sends the tasks to the workers one-by-one, and also marks the beginning and end of the basic block with explicit messages. In addition to executing the tasks normally, workers install a copy of the tasks sent within the basic block window as template for later instantiation.

4.4.1 Installation and Instantiation

As explained above, execution templates are installed at the granularity of the basic blocks in the driver program. Basic blocks can be detected either by static analysis, or through explicit annotations by the programmer. Either way, once controller detects a new basic block launched by the driver program, it initiates the template installation procedures. In addition to normally assigning the tasks in the basic block to each worker, it also marks the beginning and end of the tasks belonging to each basic block with explicit messages to the workers. This concept in depicted in Figure 4.6. Once a worker receives a `StartTemplate` message, it starts installing a new template. In addition to normal execution of the task, it adds a copy of the tasks to the template until the `EndTemplate` message is received.

For later iterations, controller instantiates the templates by loading the new parameters in the template, as depicted in Figure 4.7. Once worker receives the instantiation message, it clones the tasks in the template and executes them with the new



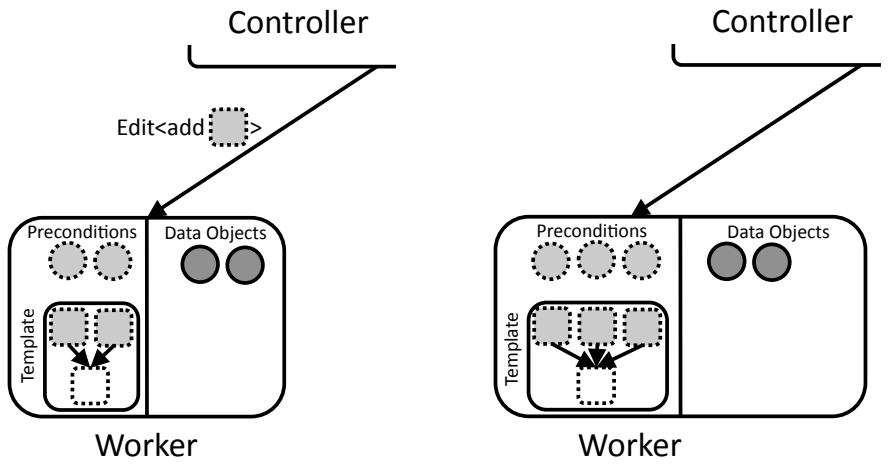
(a) Template instantiation message. (b) Instantiated template.

Figure 4.7: The template is instantiated with a single message that updates the template parameters including task identifiers and parameters passed to each function. Instantiation spawns all the cached tasks in the template without generating and sending them one-by-one. Each template has a list of preconditions that specifies the data objects needed to execute the tasks cached in the template, properly.

parameters provided by the controller. The details of the parameters list is framework dependent.

Depending on the partitioning strategy and available resources, controller could install multiple versions of a template corresponding to a single basic block. This enables a low cost switch between drastic scheduling changes, for example due to changes in allocated resource. Specifically, this is helpful as controller can install different execution plans on the workers depending on the cluster manager resource allocation decisions. Workers cache multiple execution templates, so a controller can move between several different schedules by invoking different sets of execution templates. Accordingly, the instantiation message could pick among different versions of the templates installed at the workers.

For example, Figure 4.7 shows the templates where there are two tasks assigned to the worker. If controller were to assign more or less tasks in the parallel stage to the worker, or migrate the reduction task to another worker, then a new set of templates could be installed at the worker for the same basic block.



(a) Template edit message. (b) In-place template update.

Figure 4.8: Controller can modify the content of the templates at the task granularity with the edit mechanism. Edits change the content of already installed templates in place in response to minor scheduling changes, instead of paying the cost of installing a complete new template. Edits keep the control plane overhead proportional to the extent of scheduling changes.

4.4.2 Edits

Scheduling decisions change over time: the decisions made when a template is installed might not hold for later iterations. This is specially the case for long running applications. Installing the complete template for every single change in the scheduling is expensive. First, for minor changes such as migrating a task between two workers the installation cost might be overwhelming and even outstrip the gains from the new scheduling plan. Second, the speedup gains from templates are only realized after the installation phase for the instantiation of later iterations. For transient changes (e.g. temporary stragglers), the installed templates might not even be relevant for later iterations for fast instantiation. Analogous to a memory cache where invalidating an object does not invalidate the entire cache, small changes in the template should not require complete template installation.

To this end, templates support edits to change an existing execution template. Figure 4.8 shows how edits manifest in the control plane: they modify already installed

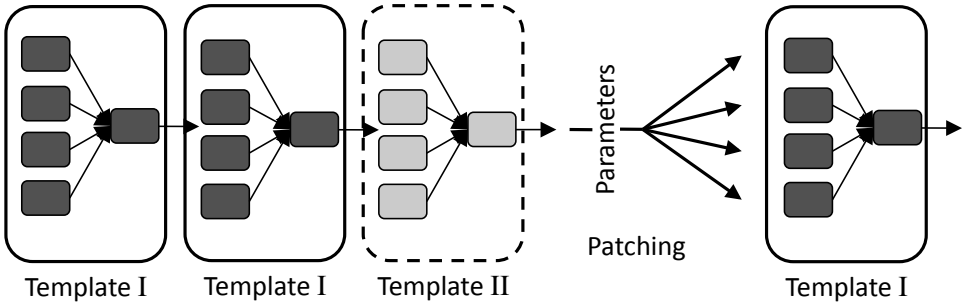


Figure 4.9: Due to dynamic program control flow, template preconditions may not always match the data objects on the workers. Before instantiation, controller checks, and if needed, patches the worker states to match the preconditions of the templates. Here, for the first iteration of the inner loop after the outer loop, the reduced `params` value is only on the reducer worker. Before instantiating the template, controller patches the worker states by updating the reduced value on all the workers. The following instantiations do not need any patching.

templates in place. Edits are issued by the controller before the instantiation message and modify the template's data structures. Depending on the extent of changes, controller can only edit a few tasks in a template, or install a new version of the template for extensive changes. Edits make the control plane overhead scale gracefully with the size of scheduling changes. If large changes are needed, the controller can install new templates.

4.4.3 Patching

Patches allow 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, or data dependent if/else blocks. With dynamic program control flow, there could be multiple nondeterministic entry points to a basic block. As a result, worker states may not meet the preconditions of the associated execution templates in all instantiation cases. In the case of precondition mismatch, controller patches the worker state. Patches are sent along with the instantiation message as a dependency for the template: no task from the template can execute until the patch is completely applied.

For example, Figure 4.9 shows instantiating a series of templates for the task graph in Figure 4.2. When the inner loop runs after the outer loop, the `param` value resides only on the reducer worker. So, controller patches the worker states by updating the `param` on all the workers before instantiating the inner loop template. The patch could be as simple as loading a new data from a networked file system, or data copy among workers; the details are framework dependent. When the inner loop runs after itself, the preconditions are already satisfied and there is no need for patches.

The driver program controls job execution and decides which basic block to execute next. Since the controller, and not the workers, tracks the entire task graph specified by the driver program, controller has the sole knowledge to make sure that the template preconditions satisfy before instantiation. Controller reacts to the driver's execution, and patches templates if needed, on the fly. Patching is analogous to register allocation for basic blocks in compilers: to enforce the correct execution of the internal block, compiler makes sure that registers hold correct variables for the basic block at each entry point.

4.4.4 Template API

Execution templates are installed and instantiated transparent to the application and driver program. From the application perspective, whether controller installs or instantiates the templates has no effects on the program functionality. Except for explicit annotation by the programmer that marks the beginning and end of each basic block, applications do not need to change at all to support execution templates. Control plane guarantees correct execution, and provides same semantics as running without templates. Available frameworks could implement execution templates support with complete backward compatibility with the programs already written.

Chapter 5

Nimbus

This chapter describes Nimbus, an analytics framework designed to support fast, optimized tasks written in lower level languages with similar performance. Nimbus meets all three requirements that are needed for execution templates as drafted in Section 4.1. After describing the architecture, execution model, and data model of Nimbus, the chapter provides the details on how execution templates are implemented in Nimbus along with the optimizations and program analysis to make them efficient.

Like Spark [128], Naiad [94], and TensorFlow [22], Nimbus is designed to run computationally intensive jobs that operate on in-memory data across many nodes. Nimbus has a mutable data model, which is crucial for in-place operations for memory intensive applications. The mutable data model lends itself to further optimization in execution template, as data objects could become the static segment of the templates. We have developed Nimbus in C++, and the task speedups discussed in Chapter 3 are readily realized in application implemented for Nimbus.

Nimbus has a task-based execution model. A driver program specifies the individual tasks along with a metadata and submits them for execution to the centralized controller. The task's metadata includes explicit data access patterns and task dependencies such that the controller can generate a *task graph* with tasks as vertices and dependencies as edges. Controller then transforms the tasks graph into a per worker *execution plan*. The execution plan includes explicit data copy tasks to realize maximum parallelism on the workers and handle inter-worker synchronizations.

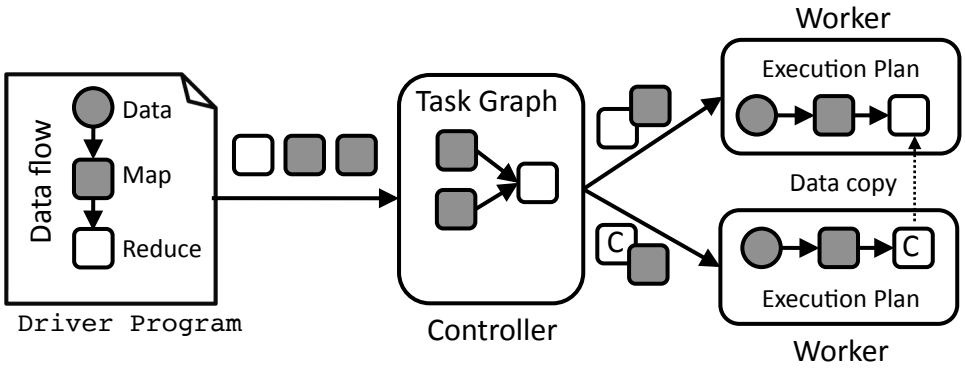


Figure 5.1: Overall design of Nimbus. The driver program submits the tasks along with a metadata of data access patterns and dependencies to the centralized controller. Controller generates a task graph with tasks as vertices and dependencies as edges, and then transforms it to per worker execution plans. The execution plan includes copy tasks for data exchange among the workers.

Figure 5.1 depicts Nimbus's overall design and execution model.

Nimbus supports traditional data analytics, including a machine learning library with supervised and unsupervised learning algorithms, and graph algorithms. Additionally, Nimbus distributes hybrid graphical simulations. This chapter describes the API with data analytics examples. Chapter 7 covers the details of porting graphical simulations into Nimbus. The core Nimbus library, excluding the applications, is about 35,000 semicolons of C++ code. The code repository is open source and can be accessed at github.com/omidm/nimbus. It includes Nimbus runtime, applications, documentations, and a glossary of scripts for deployment of Nimbus in Amazon EC2 [3]. The rest of this chapter describes Nimbus design and how it implements execution templates, in details.

5.1 Architecture

Nimbus has a centralized control plane architecture similar to MapReduce [48], and Spark [128]. Figure 5.2 shows the overall architecture and components of Nimbus.

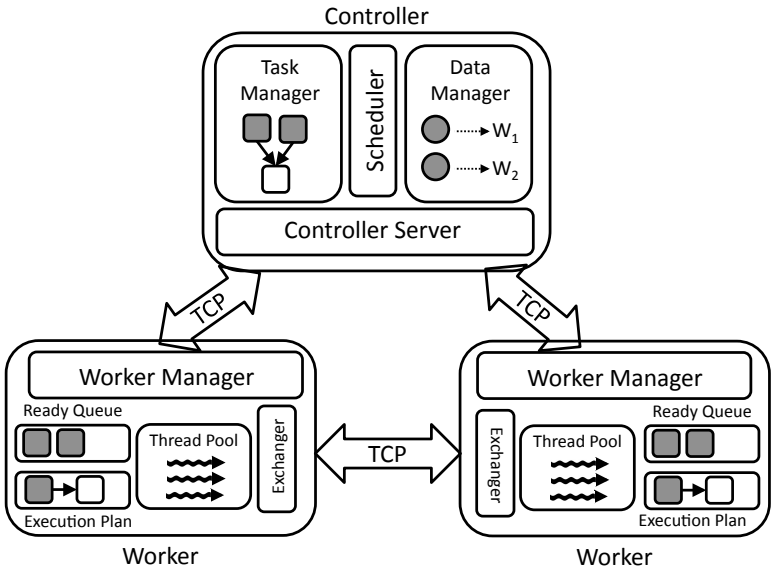


Figure 5.2: Nimbus architecture: central controller keeps the global task graph and data mapping, and schedules tasks for execution at workers. Workers keep local queues of tasks and can directly exchange data if needed. Controller-worker and worker-worker connections are TCP/IP pipes.

5.1.1 Controller

Controller has four major modules: data manager, task manager, server, and scheduler. Data manger keeps track of data partitions and their locations in the cluster. A data partition could have multiple instances either in the main memory or on the persistent disk of different workers. Task manager generates and tracks the distributed execution plan of the application over the workers. Controller server is responsible for communications between controller and the workers, and handles command exchanges among them. Controller scheduler enables dynamic load balancing and task partitioning among the workers.

In Nimbus, workers can dynamically join or leave a cluster managed by a controller. Controller server listens on a publicly announced port number for new TCP connections from workers. For each new worker that connects to the controller, server opens a persistent TCP connection between controller and the worker. Controller uses asynchronous remote procedure calls (RPC) over the TCP connection to assign tasks to the workers, and receive heartbeat messages and execution results from the

workers. The TCP connection is kept live to avoid the TCP slow start for each RPC.

Controller has a scheduling interface that allows dynamic resource allocation and task partitioning among the workers. Scheduler supports both batch and per task assignments interfaces. Specifically, scheduler can assign a batch of tasks to the workers at once. As we will see, this is important to enable execution templates under Nimbus, as batch assignments are required to instantiate a batch of tasks at the workers in the form of a template.

5.1.2 Workers

Each computing machine in the cluster runs an instance of the Nimbus worker. Worker manages the resources at the machine such as shared memory, CPU cores, Network I/O, and disk accesses. Worker allocates memory for the data objects in the main memory for task executions. Also, it can save persistent copies of the data objects on the disk upon controller requests, for example for checkpointing purposes.

Each worker has a task queue that holds the tasks received from the controller. Controller assigns tasks along with explicit dependency metadata such as prior tasks that needs to complete, or a data exchange. The details of this metadata and explicit dependencies are discussed below (Section 5.2). Once all the dependencies of a task are satisfied, worker pushes the task to a ready queue. Worker has a thread pool that executes the tasks in the ready queue on the CPU cores. To avoid unnecessary context switches, there are as many threads as the number of cores at the worker.

Workers can directly exchange data through TCP connections among them. Each worker has a data exchanger module that listens on a specific port number for TCP connections. Worker announces the port number to the controller once it connects for the first time. If data exchange is necessary among the workers, controller explicitly sends copy tasks to the workers. Each copy task has a metadata that includes receivers listening port and IP address, such that the sender can initiate the communication with the receiver. Nimbus worker has a push model for data exchanges, meaning that transmission starts by the sender once the data is available. This helps mask the communication latencies with the task execution at the workers.

5.2 Execution Model

Each instance of an application that runs in Nimbus is called a *job*. Similar to other data analytics frameworks [48, 128, 94], Nimbus has data parallel operations to increase memory bandwidth and aggregate computation available to the job. Data sets are partitioned into smaller pieces and spread among the workers. There are as many computation units, called *tasks*, as the data partitions.

Nimbus has a task-based driver program abstraction similar to Legion [32]. The Nimbus driver program specifies the application logic at the granularity of individual tasks. For example, Figure 5.1 shows that driver program generates and submits individual `map` and `reduce` tasks to the controller. Each tasks has a metadata that specifies its data access patterns and dependencies, explicitly. Nimbus allows arbitrary tasks dependencies patterns among tasks. Note that this is more general than the stage-by-stage data flow model in counterpart frameworks such as Spark [128] and MapReduce [48]. The stage-by-stage model is restricted by the narrow or wide dependencies. However, there are many applications that does not fit this model. For example, as we will see in Chapter 7, graphical simulations have geometric dependencies that require more complex task dependencies.

5.2.1 Program Control Flow

Nimbus allows dynamic control flow based on the computed data at runtime. A task, in addition to operating on the data objects, can also spawn other tasks for execution. Specifically, tasks execute on the workers and might submit other tasks to the controller for scheduling. Controller then partitions and assigns the submitted tasks back to the workers for execution. Figure 5.3 shows the task spawning concept with an example. In this scenario, the black task spawns three other tasks and submits them to the controller for scheduling, which are then assigned back to the workers for execution.

The task spawning model for the program control flow is similar to continuation passing style [114], where tasks are continuations. Driver program is a lineage of tasks that execute on the workers. Application developers provide an execution function

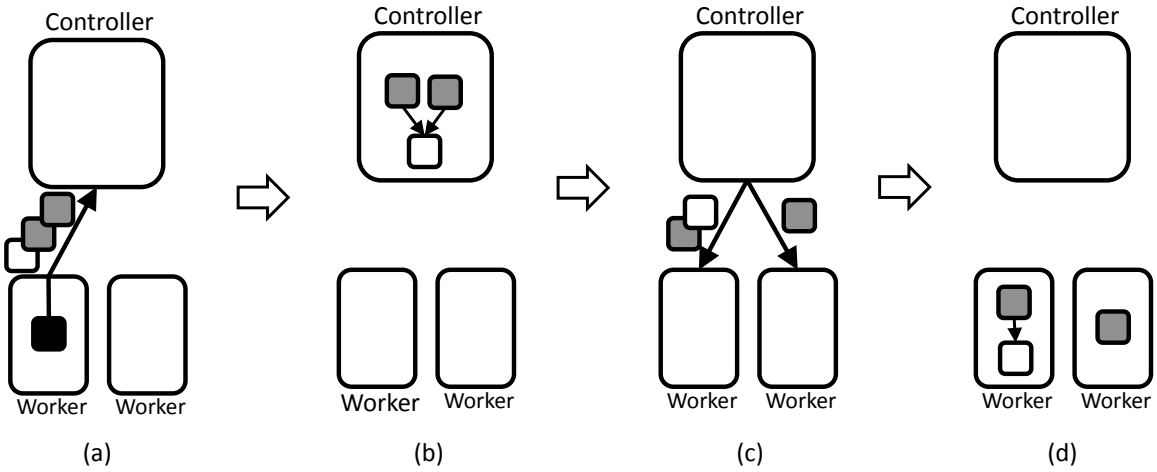


Figure 5.3: A task spawning example scenario in Nimbus: a) the black task that runs on the left worker spawns three tasks and submits them to the controller, b) controller receives the tasks from the worker, c) controller partitions and assigns the tasks back to the workers for execution, d) workers execute the tasks.

for each of the tasks, which might include spawning other tasks. Every application needs to provide the execution function for a special task, called `main`. Controller starts the job by executing the `main` task on one of the workers, and the rest of the tasks are generated and scheduled for execution from there.

Nimbus task spawning model allows dynamic control flow and data dependent branches in the driver program. Tasks could change the computation course of a job during runtime by spawning different set of tasks either through deterministic decisions or nondeterministic and dependent on the values held in the data objects that the task accesses. For example, depending on an error value reduced in a data object, the task can either spawn more tasks for another iteration of the optimization or terminate the loop and spawn the tasks for the operations after optimization. Spawning and scheduling at task granularity satisfies the first requirement for execution templates, as drafted in Section 4.1.1.

The task spawning model, while flexible, induces extra overhead on the control plane as the tasks are submitted to the controller one-by-one, and then assigned back to the workers for execution. As we will see in Section 5.5, execution templates help optimizing this pin-pong effect between controller and workers.

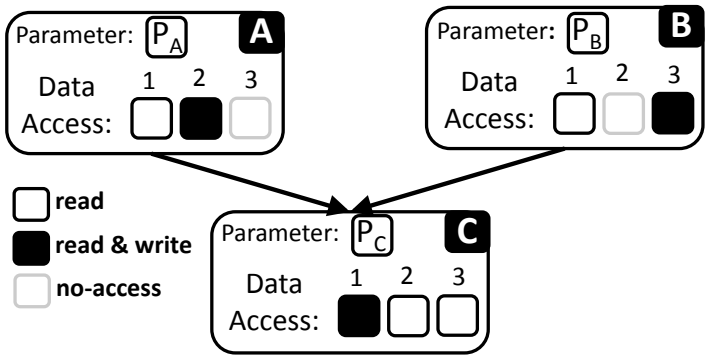


Figure 5.4: An example task graph in Nimbus. The task graph has task metadata as vertices and before set dependencies as edges. Also, the tasks metadata specifies the read/write access patterns, execution function, and a binary blob of parameters passed to the function.

5.2.2 Task Graph

In addition to a globally unique identifier, each task when spawned has a metadata of 5 pieces:

1. **Read set** of data objects to read.
2. **Write set** of data objects to write.
3. **Before set** of tasks that need to finish executing before the task starts.
4. **Function name** that gets called for task execution over the data objects.
5. **Parameter** as binary blob that are passed to the task function.

The task metadata specifies the explicit data access patterns in terms of read/write sets, and explicit task dependencies in terms of before sets. The before set enforces the ordering and data flow among the tasks. Nimbus driver program is compiled as a dynamically linked library, and each worker has a copy of it. Workers load and run the execution function for each task based on the function name in the metadata.

Once controller receives the spawned tasks, it generates a directed acyclic graph (DAG), called *task graph*, with task as vertices and task before set information as edges. Each vertex holds the metadata of the task as well. For example, Figure 5.4

shows a simple task graph with three tasks. The arrows between the tasks show the ordering based on the before set relations. Here, for example, task C has tasks A and B in its before set. This enforces C to run only after A and B finish, such that C reads in the updated second and third objects that are written by A and B. We will use this task graph as a running example in the rest of this Chapter.

The read/write/before set abstraction is general enough to cover any data access pattern or dependencies. However, filling in and handling the task metadata information could be tedious for the application developer. Nimbus provides simple interfaces to fill in the task metadata for classic stage-by-stage data analytics, or geometric based operations for applications such as graphical simulations. The interface is covered in Sections 5.6 and 7.4.

5.2.3 Execution Plan

Depending on the available resources, controller partitions and transforms the task graph to per worker *execution plans*. In addition to the tasks in the task graph, controller inserts special data copy tasks in the execution plan for data replication on a worker locally, called local copy, or synchronization among workers, called *remote copy*. Additionally, controller updates the original before set of the tasks in the task graph by removing cross worker dependencies and adding necessary copy tasks to enforce communication and computation orders for correct execution.

Local copy tasks enable controller to exploit maximum parallelism among the tasks in the task graph. It helps remove the read-write conflicts between the tasks that could potentially run in parallel on the same worker. Specifically, if one of the tasks has read and write access over a data object, and another tasks has a read-only access, and there are no other explicit before set dependencies, they could potentially run in parallel. However, if both tasks share a same instance of the data object, they are forced to run serially: the task with the write access could run only after the task with the read-only permission finishes. Controller detects such conflict in the tasks graph and resolves them by deciding whether to duplicate the data objects

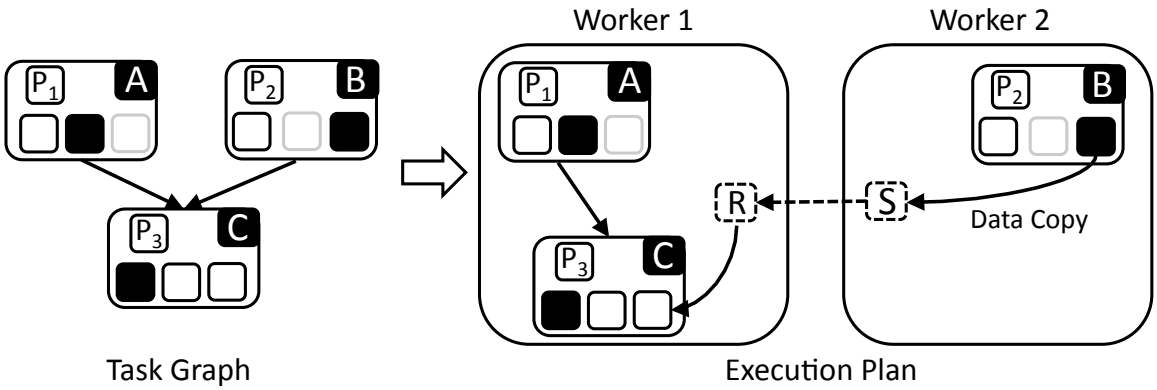


Figure 5.5: An example execution plan corresponding to the task graph in Figure 5.4. In addition to the tasks in the task graph, the execution plan includes explicit data copy tasks for sending and receiving data. Controller replaces inter-worker dependencies with copy tasks, such that workers can synchronize execution, independently.

through local copy commands and by giving each tasks its own data copy. The read-write conflict happens often in scientific applications (e.g. graphical simulations) with overlapping ghost regions.

Remote copy tasks realize data exchanges among workers. To avoid the cost of data migration among workers, controller normally assigns tasks to where data exists. However, there are cases where there is no single worker that holds all the data objects needed by a task, and data exchange among workers is inevitable. For example, in a global reduction, partial results computed on each worker are gathered on one worker for final reduction. Since Nimbus controller has the global view of the data objects, it coordinates the required data exchanges among the workers *proactively*. For each data exchange controller assigns a *send* copy task to the sender worker, and a *receive* copy task to the receiving worker.

For example, Figure 5.5 shows the execution plan over two workers corresponding to the tasks graph in Figure 5.4. In this scenario controller has decided to run tasks A and C on one worker, and task B on the other. Task C that runs on the left worker requires the updated third object on the right worker. So, controller inserts send and receive copy tasks on the workers to exchange data. Copy tasks explicitly name the workers and data object involved in the transfer, such that workers can

directly exchange data. This satisfies the second requirement for execution template, as drafted in Section 4.1.1.

Controller assigns tasks to the workers along with their before set. This enables the workers to order a batch of tasks, locally. However, the before set might differ from the initial before set in the task graph. Note that, after partitioning a task graph among workers, some tasks on one worker might appear in the before set of tasks in the other worker. To avoid flooding explicit synchronization messages among workers, controller replaces the inter-worker dependencies with copy tasks: a data exchange acts as an implicit synchronization. For example, in Figure 5.5, controller updates the before set of task C by replacing task B with the receive copy task.

By adding explicit copy tasks, workers execute a batch of tasks locally based on the before set ordering, without any further explicit notification by the controller. In this design, controller can assign tasks beyond a synchronization point since inter-worker dependencies are resolved, proactively. This is similar to data flow graph model in TensorFlow [22]. Once controller assigns the tasks, workers can independently order and execute them. This satisfies the third requirement for execution templates, as drafted in Section 4.1.1.

5.3 Data Model

In Nimbus, each data set is defined as an instance of a data type along with the partitioning information. The data set is partitioned into many *data objects* with the same type, so that tasks can operate on them in parallel. Each data type defines four methods: serialization, deserialization, create and destroy. The first two methods are used for transmitting data objects among workers or saving and loading them off the persistent disk. The second two methods are used to allocate and deallocate the data objects on the worker nodes.

In addition to a unique identifier, data objects have an associated geometric bounding box. The data set is defined over a global bounding box and based on the partitioning information, each data objects covers a sub-domain. This is quite handy for operations with geometric locality, such as graphical simulations. For data

analytics you can think of data objects partitioned along one dimensional space. Chapter 7 elaborates on the benefits of explicit geometry for graphical simulations.

5.3.1 Mutable Data

Data objects in Nimbus are mutable. Tasks with write access can mutate the data objects. Supporting in-place modification of data avoids data copies and is crucial for computational efficiency because writes and reads operate on the same cache line. Also, for memory intensive computations, it saves memory space since there is no need for memory allocation for intermediate placeholders.

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 being a runtime parameter. This makes templates more efficient to parameterize, as the object identifiers can be cached 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 at the controller.

Tasks with write access privilege mutate the data. To avoid inconsistencies, Nimbus does not allow parallel writes to a data object. All tasks with a write access to a data object have to be ordered in the tasks graph with either direct or indirect before set relations. In other words, each data object has a well-defined *lineage* of tasks that mutate the object. If parallel writes are detected, Nimbus raises a runtime error.

5.3.2 Data Versioning

Mutable object means that there can be multiple copies and versions of an object in the system. For example, for the scenario in Figure 5.5, before executing the tasks both workers have a copy of the first data object. However, after task C executes, the first data object is updated on the left worker, while the right worker copy of the first object becomes obsolete. Each data object in the system therefore combines an object identifier with a version number. The Nimbus controller ensures, through data copy tasks, that tasks on a worker always read the latest value according to the

program's control flow.

The data version that a task receives depends on the data flow in the task graph. To be concrete, we define the version context to be the mapping between a data object id and a version number. Each task has an input version context based on which it accesses the data objects, and an output version context that it produces. The version of a data object, d, in the version contexts of a task, t, is determined with the following two simple rules:

1. If t has write access on d, then the version of d in the input version context of t is incremented by one in the output version context of t. Otherwise, the input and output versions are equal for d.
2. The version of d in the input version context of t is the maximum version among all the versions in the output contexts of the tasks in the before set of t.

These two rules concretely defines the data flow in the task graph based on the before set information and data access patterns.

In theory, to determine the data version for each task, controller can calculate the input and output context for every task according to the rules above. The computation complexity of such naive approached would be $O(TD)$, where T is the number of tasks, and D is the number of data objects. Since, the number of data objects and the number of tasks are within a constant factor of each other, the complexity is in fact quadratic in T, $O(T^2)$. This is not tractable for controller, especially at scale.

Instead, Nimbus leverages the lineage of the data objects to expedite the versioning process. Specifically, controller keeps track of the lineage of the tasks with write accesses over each data object. This requires a doubly linked list data structure with space complexity of $O(dD)$, where d is the depth of the task graph. Then, for every version lookup, controller traverses the lineage backward and finds the first ancestor of the tasks in the linage. The ancestor relation is defined according to the before set metadata in the task graph. That version is the version that the task receives in the task graph. With this approach, only the data objects in the read/write set of the task are resolved, and hence the computation complexity is $O(|A|T)$, where $|A|$ is the cardinality of the read/write set. Note that $|A|$ is usually a constant factor,

and greatly smaller than T. Also, as we will see, controller can further optimize the versioning process by memoizing the versions within execution templates.

5.3.3 Garbage Collection

Nimbus provides automatic garbage collection at the run time. As a part of generating the execution plan, controller recycles data objects with obsolete versions by letting the tasks with write access to overwrite them with newer versions. A data version is obsolete if there are no other tasks currently in the task graph or potentially spawned in future by other tasks that might need to read a data object with the same version.

Note that tasks are spawned dynamically in Nimbus, and child tasks could reference any of the data version in the version context of the parent task. Nimbus does not impose any context restriction for the sake of driver program simplicity. As a result, controller has to hold on to at least one copy of every single data version in the version context of every potential parent task in the task graph. Note that not all tasks spawn other tasks in the driver program. For example, usually only the last tasks in the task graph spawns the tasks in the later iterations. Nimbus leverages this fact by requiring the driver program to explicitly mark the parent tasks.

All the tasks are *sterile* by default unless marked explicitly as *parent* in the driver program. If a sterile task attempts to spawn other tasks, Nimbus raises a runtime error. This way, controller needs to only hold on to the data versions in the context of parent task, and efficiently allow data mutation for sterile tasks. This approach effectively resemble memory footprint of a program in managed languages. Without the notion of parent tasks, the garbage collection is not feasible in applications with data mutations, as it means replicating the data object before each write access.

5.4 Control Plane

Controller plane in Nimbus includes the interactions between controller and workers. Tasks graph is generated dynamically during runtime. The spawned tasks at the workers are collected and submitted to the controller for processing. Controller

transforms the task graph into an execution plan by inserting copy tasks in the tasks graph, partitions and schedules tasks among workers, and recovers the execution from worker failures.

5.4.1 Commands

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 are either for spawning tasks from workers to controller or for submitting tasks from controller to workers for execution.

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. Controller updates the before set of spawned tasks, such that it includes only other tasks on the same worker. If there is a dependency on a remote worker, this is encoded through a copy command.

Copy commands, similar to tasks, have a before set. Copy tasks 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 receive copy task is satisfied (the new object is safely visible to the worker), it changes a pointer in the data object to point to the new buffer.

5.4.2 Load Balancing

Production data analytics over big data require a large amount of memory and node hours. Elastic cloud services such as Amazon EC2 [3], Google Cloud [13], and Microsoft Azure [17] have a pay for resources as needed model, making it possible to run large scale applications in the cloud. However, one is left to deal with straggler nodes, arising from oversubscribed and shared resources such as compute nodes and

network, and failures such as I/O failures. Even private clusters exhibit these problems, when using a large number of nodes for long periods of time. Disruptions due to failures and slow-down due to straggler nodes can be very expensive in terms of time and effort. Nimbus automatically load-balances applications, and provides fault tolerance.

Nimbus worker nodes periodically send total time spent in computation tasks to the controller. A high compute time to total time ratio indicates that a node may be a straggler – the node takes more time to complete compute tasks, while other nodes are blocked on it on synchronization points. Such imbalance can come from oversubscription of shared resources or interference from other applications, difference in CPU speeds, or even from within the application, for instance, skew in the data size [29]. A low compute time to total time ratio triggers migration – Nimbus moves some partitions from the straggling worker to neighboring workers. The controller sends commands to migrate tasks and data to worker nodes, and worker nodes exchange data accordingly. This is repeated till the ratio of compute time to total time falls within the threshold. If a node is particularly slow, then all the tasks from that node may be moved to neighboring nodes.

5.4.3 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 [128] 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 snapshot.

Nimbus automatically inserts checkpoints into the task stream from a driver program. It chooses the checkpoint interval based on an estimation of the node failure rate and the time to write to stable storage. When a checkpoint triggers, the controller waits until all worker task queues drain, stores a snapshot of the current task 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 and/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 that they are ready to restart. The controller sends commands to load the latest snapshot into memory, reverts to the stored task graph, and restarts execution.

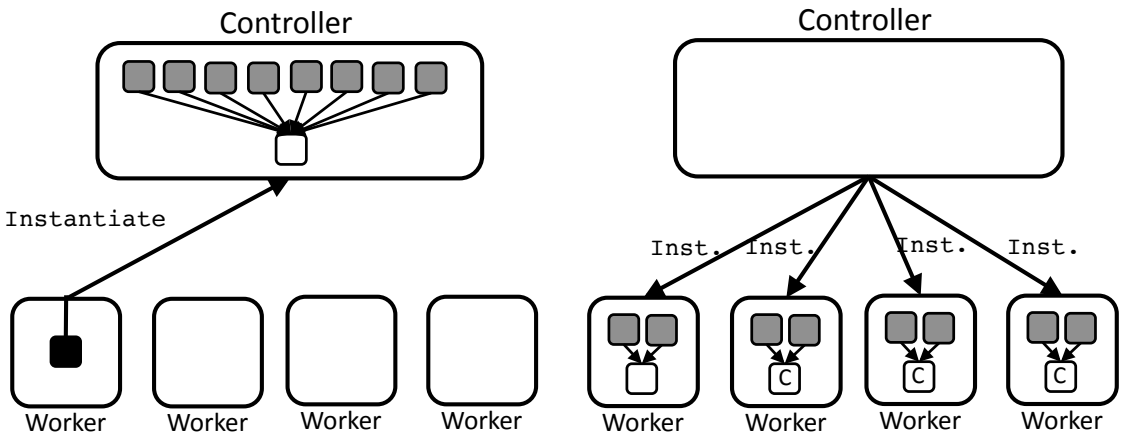
Current implementation of Nimbus does not survive controller failures. However, one can imagine a replicated mechanism for snapshots, for example through a Paxos ring [81]. Upon controller failures, workers could connect to a backup controller and resume execution from the latest snapshot.

5.5 Execution Templates in Nimbus

This section covers the details on how execution templates and their mechanisms are implemented in Nimbus. Execution templates optimize the functionalities of control plane between controller and workers. Since in Nimbus control flow workers generate and spawn the tasks and then controller schedules them for execution on the workers, 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 caches the entire task graph, while worker template caches the per worker execution plans.

5.5.1 Controller and Worker Templates

Controller templates contain the complete list of tasks in the task graph of a basic block. They cache the results of creating tasks, dependency analysis, data lineage, bookkeeping for fault recovery, and assigning data partitions as task arguments in the task graph. 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. As depicted in Figure 5.6(a), controller templates allow driver program to instantiate the entire tasks graph on the



(a) Controller template. (b) Worker template.

Figure 5.6: Controller and worker template in Nimbus. Controller template enables driver program to spawn an entire task graph on the controller with a single message. Worker templates let controller instantiate the execution plan on the workers with a single message.

controller with a single instantiation message, instead of sending individual tasks and their metadata, one-by-one from scratch.

Where controller templates describes a basic block over the whole task graph, each worker template describes the portion of the execution plan that runs on a particular worker for each basic block. Workers cache the dependency information needed for a worker to execute the tasks and schedule them in the right order. Like TensorFlow [22], 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. As depicted in Figure 5.6(b), worker templates allow controller to instantiate an execution plan on the workers with a single message.

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

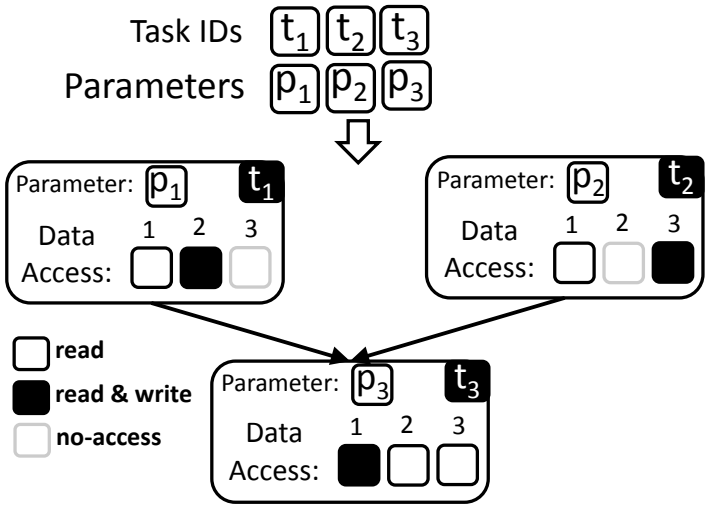


Figure 5.7: Controller template content for the task graph depicted in Figure 5.4. A controller template represents the common structure of a task graph metadata. It stores task dependencies and data access patterns. It is invoked by filling in task identifiers and parameters to each task.

assignment has changed, 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. This is shown in Figure 5.6.

5.5.2 Installation and Instantiation

Driver programs explicitly install controller templates. This is necessary because only a driver program has the complete program structure so it knows where basic blocks begin and end. Template installation begins with the driver sending a start template message to the controller at the beginning of a basic block. In current implementation of Nimbus, programmer explicitly marks the basic block in the driver program; one can imagine 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

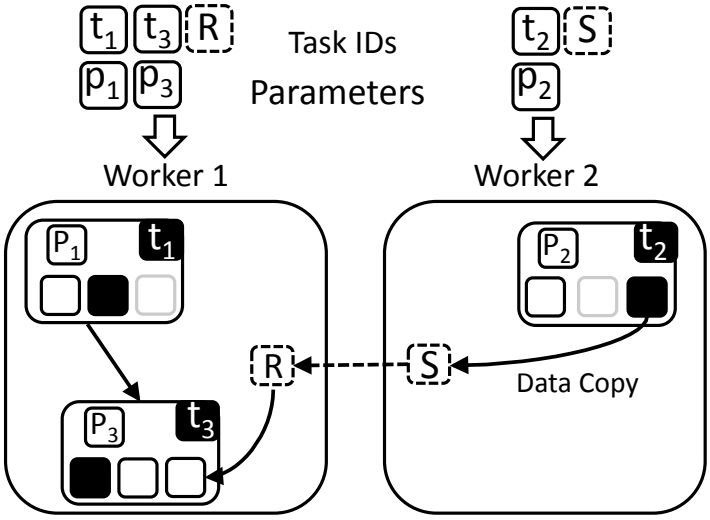


Figure 5.8: Worker template content for the execution plan depicted in Figure 5.5. A worker template represents the common structure of a task graph metadata. It stores task dependencies and data access patterns. It is invoked by filling in task identifiers and parameters to each task.

the task graph and post-processes it into 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 along with before set metadata as indices into an array as depicted in Figure 5.7. A template instantiation message includes an array of task 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 5.7 shows the instantiation of a controller template with a new set of task identifiers and parameters.

Once it has generated the controller template, the controller generates the associated worker templates. Worker templates cache the per worker execution plan on each worker, and their content depends on how tasks are partitioned among workers. It caches the tasks (including copy tasks), each data object that they access, and the dependency list, as depicted in Figure 5.8.

Worker template has a preconditions list of which data objects at each worker

must hold the latest version to that object. An important detail is that 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 5.8, 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. Controller keeps a copy of the worker template preconditions centrally. Before instantiation, controller validates whether data object states match the preconditions, and generates and applies patches, if needed.

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 5.8 shows a set of worker templates and how they are instantiated for the execution plan in Figure 5.5.

5.5.3 Patching

Validation and patching allow templates to efficiently handle dynamic program control flow. Before instantiating a worker template, controller must *validate* whether the template's preconditions hold and patch the worker's state if not. For example, suppose that the driver invokes the controller template in Figure 5.7 twice, back-to-back as depicted in Figure 5.9. Both worker templates have a precondition that their copy of object 1 contains the latest update. This is true for worker 1, since it wrote to object 1, but it is not true for worker 2. The controller therefore needs to issue patches: a data copy from worker 1 to worker 2.

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.

Validation and Patching Optimizations

The first optimization relates to template generation. When generating a worker template, Nimbus ensures that the precondition of the template holds when it finishes.

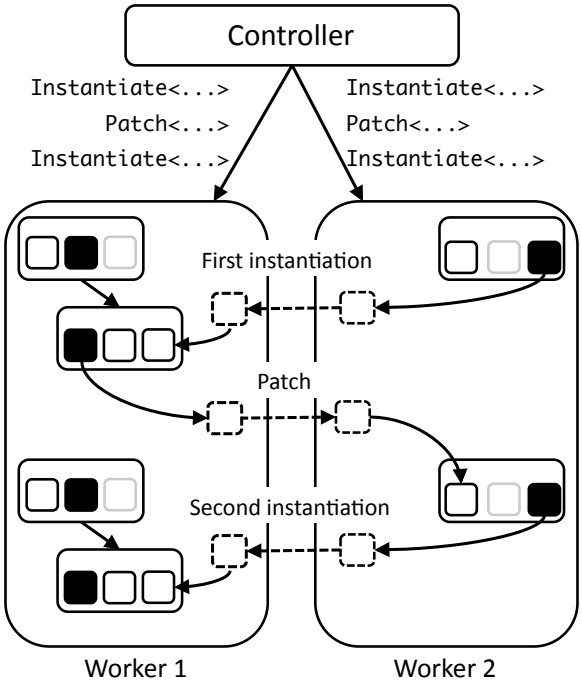
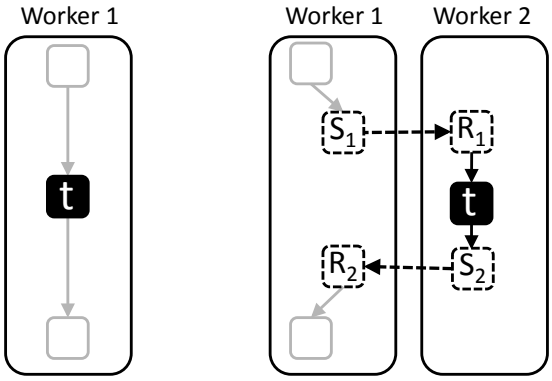


Figure 5.9: An example patching scenario in Nimbus. After first instantiation of the worker templates, the first data object is outdated on the right worker. Controller patches the right worker's state to match the preconditions of the worker template, before instantiating a second worker template.

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 5.8, 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. Otherwise, it calculates a new patch and sends all of the resulting commands. We have found that the patch cache has a very high hit rate in practice because control flow, while dynamic, is typically quite narrow (analytics applications do not have switch statements or many nested conditionals).



(a) Before. (b) After.

Figure 5.10: Edits to migrate 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.

5.5.4 Edits

Controllers need to make small changes to how a program is distributed across workers, either in response to stragglers, changes in the available nodes, or load imbalance. Templates can contain thousands of tasks and so installing one can be a significant control plane cost, comparable to the cost of not using a template at all for that iteration. Instead of installing new templates for every change, controller edits the content of templates to match the scheduling changes.

Whenever a controller instantiates a worker template, it can attach a list of edits for that template to apply before instantiation. Each edit specifies either a new task to include or a task to remove. Content changes due to 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 5.10 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 and the cost scales with the size of the change.

```
1  class Sample : public Data {
2    public:
3      void Create() {
4        // allocate memory and load data.
5        LoadDataFromDisk(data_id_);
6      }
7      void Destroy() { // free the allocated in-memory state.}
8      string Serialize() {// serialization logic.}
9      Data* Deserialize(string bytes) {// deserialization logic.}
10 };
```

Listing 5.1: Defining the `Sample` data type for logistic regression algorithm.

5.6 Application API

This section presents Nimbus API and describes how applications are written in Nimbus with an example. Specifically, it shows how the pseudo code of the logistic regression algorithm depicted in Figure 3.2 is implemented in Nimbus. The core library implements common functionalities, and there are simple interfaces to implements application specific details. There are two main classes in Nimbus: `Data` class, and `Task` class. Every data type inherits from the `Data` class and implements a few basic methods for memory allocation and serialization. The tasks inherit from the `Task` class which implements basic methods for control flow and data access functionalities, and developers override the execution methods to implements the application logics. Nimbus library also provides high-level helper methods for task spawning and explicitly marking the boundaries of templates.

5.6.1 Data Interface

Nimbus does not constrain the structure of the data types. However, every data type should implement four main methods: `Create`, `Destroy`, `Serialize` and `Deserialize`. The first two methods cover memory allocation and deallocation for the data objects. The second two allow Nimbus to exchange data object over the network or save/load the data objects on/from the persistent disk for checkpointing and recovery purposes. For example, Listing 5.1 shows how the input data samples of logistic regression are

```
1  class Main : public Task {
2    public:
3      void Execute() {
4        int iteration_num = 10;
5        int partition_num = 100;
6        CreateData("samples", Sample, partition_num);
7        CreateData("coeff", Coefficient, partition_num);
8        SpawnTask("loop", string(iteration_num));
9      }
10 };
```

Listing 5.2: The `main` task implementation for logistic regression.

ported into Nimbus. Each data object upon instantiation has a unique identifier. In line 5, for example, the data identifier is used to load a partition of the data samples off the disk into main memory for later computations.

5.6.2 Task Interface

Application specific tasks override the `Execution` method of the base class. This is the method that gets called once the task is executed. Each task can access its binary string of parameters and the list of data objects in its access list, perform arbitrary computation, create new data objects, and spawn other tasks. The application lifetime starts by executing a special task named `Main`. Other tasks are spawned from there.

For example, Listing 5.2 shows how the `Main` task for logistic regression example is implemented. In line 6 and 7, the data objects are defined and partitioned. In addition to the input data sample type defined in Listing 5.1, there is also the `Coefficient` type to hold feature coefficients. Here, for example, the input data samples are partitioned in to 100 objects, and there are same number of coefficient object, one for each local gradient operations on the samples partition. After defining the data objects, the `Main` task spawns the *Loop* task which will in turn spawn the tasks iteratively for the gradient and reduction operations. The loop task does not read/write any data objects, however, it has an iteration number parameter that specifies the loop counter.

```
1  class Loop : public Task {
2    public:
3      Execute() {
4        int partition_num = 100;
5        StartTemplate();
6        {
7          StartStage();
8          for (int i = 0; i < partition_num; ++i) {
9            set<uint64_t> read, write, before;
10           LoadAccessSet("samples", &read, i);
11           LoadAccessSet("coeff", &read, i);
12           LoadAccessSet("coeff", &write, i);
13           LoadBeforeSet(&before);
14           SpawnTask("gradient", read, write, before);
15         }
16         EndStage();
17       }
18       {
19         StartStage();
20         set<uint64_t> read, write, before;
21         for (int i = 0; i < partition_num; ++i) {
22           LoadAccessSet("coeff", &read, i);
23           LoadAccessSet("coeff", &write, i);
24         }
25         LoadBeforeSet(&before);
26         SpawnTask("reduce", read, write, before);
27         EndStage();
28       }
29       EndTemplate();
30
31       int iteration_num = int(GetParameter());
32       if (iteration_num > 0) {
33           SpawnTask("loop", string(iteration_num - 1));
34       } else {
35           TerminateApplication();
36       }
37 };
```

Listing 5.3: The `loop` task implementation for logistic regression.

5.6.3 Stage Operations

Filling in the identifiers in the read/write set of the tasks for spawning could be tedious. For example, each iteration of the logistic regression in Figure 3.2 requires many parallel tasks for gradient operation followed by a global reduction. Each gradient task reads a partition of the samples and coefficients, and writes the coefficients partially. The reduce task reads in all partially updated coefficients and reduces them. Listing 5.3 shows how these tasks are spawned along with their metadata within the loop task. Nimbus library provides helper task methods for loading the read/write set as shown in lines 10 to 12.

Also, instead of handling before set entries manually to shape the task graph, programmers could only mark the beginning and end of each parallel stage. For example, all the gradient tasks are spawned between the `StartStage` and `EndStage` directives in lines 7 and 16. Based on the parallel boundaries, the built in method fill in the before set entries, as in lines 13 and 25. Any read after write dependency beyond a single stage results in a before set relation. For example, the reduce tasks has all the gradient tasks in it before set, since it reads the coefficients written by the gradient tasks. The tasks within a stage boundary do not have any before set relation with each other and hence can run in parallel.

5.6.4 Template Boundaries

Current implementation of Nimbus requires the application developers to explicitly mark the beginning and end of each basic block to mark the part of the program with no branches. For example, Listing 5.3 shows how the code block between lines 5 and 29 is marked as a template. The gradient and reduce tasks spawning is captured within the template, while the conditional branch afterward is excluded. After the template basic block, depending on the loop counter, another instance of the loop tasks is spawned or application is terminated (line 35).

```
1  class Gradient : public Task {
2    public:
3      Execute() {
4        Sample& samples = GetData("samples");
5        Coefficients& coeff = GetData("coeff");
6        Coefficient gradient(0);
7        for (int i = 0; i < samples.size(); ++i) {
8          gradient += DotProduct(samples[i], coeff);
9        }
10       coeff = gradient;
11     }
12 };
```

Listing 5.4: The `gradient` task implementation for logistic regression.

5.6.5 Computation on Data

Finally, Listing 5.4 is an example of how tasks access and compute on the data objects in their access list. Here, the gradient task reads the samples in its data partition and updates it coefficient data object. The reduce task would similarly read in all the coefficient object and reduce them.

Chapter 6

Evaluation

This chapter evaluates execution templates in Nimbus, comparing them with state-of-the-art frameworks with distributed and centralized control planes. It considers micro benchmarks to evaluate the limits of execution templates in extreme cases. End-to-end performance evaluations of common machine learning and graph processing applications show execution templates in practice for traditional data analytics. In addition, it shows the behavior of execution template in dynamic scheduling and resource allocation scenarios. In summary, we find:

- Execution templates allow centrally scheduling **high task throughput** applications at hundreds of thousands of tasks per second, imposing a control plane overhead competitive with distributed control planes. For example, in an application with a *single-stage* task graph (worst case for templates in terms of cached block size), Nimbus can schedule more than 500,000 tasks per second over a cluster of 100 workers.
- Execution templates enable **dynamic scheduling** at task granularity, providing runtime flexibility and adaptivity equivalent to frameworks with a centralized controller. A single task migration imposes an overhead of 41μs, and the scheduling cost grows linearly with the number of changes.
- Execution templates allow low-cost, **dynamic resource allocation** in the cluster. Template metadata size is negligible compared to the in-memory data of

the applications, such that worker nodes can install many different versions of the templates, and controller validates, patches, and instantiates them quickly according to even drastic resource changes in the cluster.

The next chapter describes porting graphical simulations in Nimbus to evaluate execution templates in practice for applications with complex **dynamic control flow**. The nested loops and data dependent branches in graphical simulations trigger subtle validation and patching cases. Execution templates allow Nimbus to scale to the task throughput requirements of graphical simulations while imposing negligible patching cost, showing comparable performance to application-level MPI [113] implementation, where control flow is statically compiled.

6.1 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, with twice as many resources, to show how jobs bottleneck on the controller even when it has more resources. All nodes are allocated in a single placement group and have full bisection bandwidth at 1Gbps. We have measured the round trip time (RTT) to be $\approx$700μs in average among instances. In the experiments, we measure iteration time and control plane overhead on clusters with up to 100 worker nodes.

We compare the performance of Nimbus with Spark 2.0 and Naiad 0.4.2. Spark [128] is the state-of-the-art and widely used framework [10] with a centralized control plane model. It is an accepted fact that Spark is superior to MapReduce [48] and Hadoop [6] in terms of performance for in-memory computations [128]. The latest distribution, Spark 2.0, implements tungsten code generation engine for CPU performance optimizations [1, 121]. We use MLlib [9] and GraphX [61] libraries from Spark 2.0 to implement machine learning and graph processing benchmarks.

We consider Naiad [94] as a representative for frameworks with distributed control plane model. TensorFlow's control plane design [22] is very similar to Naiad's, which

results in close performance and behavior. Note that TensorFlow uses a centralized Master for data flow instantiations, which leads to slightly lower performance numbers compared to completely decentralized data flow model in Naiad. We measured TensorFlow's task throughput at about ≈200,000 tasks per second for a cluster of 100 workers, while Naiad reports task throughputs as high as ≈680,000 tasks per second for a cluster of 64 workers [94]. We do not believe that these performance differences are fundamental flaws in the design of TensorFlow, but rather implementation artifacts. To be safe, we consider Naiad as the bottomline performance target.

Because our goal is to measure the task throughput and scheduling granularity of the control plane, we factor out language differences among frameworks and have them run tasks of equal duration. Nimbus tasks are implemented in C++ and 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 for C# implementations. We set the task duration as the fastest of the three frameworks, as it evaluates the highest task throughput. This is done by replacing the task computations with a spin-wait as long as C++ tasks in Naiad and Spark. To show that tasks in Naiad and Spark are not C# or Scala codes but rather tasks that run as fast as C++ ones, we label them *Naiad-opt* and *Spark-opt*. This evaluates the performance of these frameworks if their tasks called directly into native code with no overhead.

The Naiad and Nimbus implementations of global reductions include application-level, two-level reduction trees. Workers locally reduce their results before passing a single partial reduction to the global reducer. Application-level reductions in Spark harm completion time because they add more tasks that exacerbate the bottleneck at the centralized controller.

6.2 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 evenly into 100 worker templates, each with 80 tasks. Each worker has 8 cores, and so there

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

Table 6.1: Template installation is fast compared to scheduling. The 49μs per-task cost is evenly split between the controller and worker templates. Controller template has a one time cost for each basic block in the driver program. Installing a new worker template for each scheduling and tasks partitioning strategy has a per-task cost of 24μs, including generating preconditions at the controller. This is only an 18% overhead on centrally scheduling that task.

are 10 tasks per each core in the cluster. This is a conservative task granularity for straggler mitigation purposes [46, 97]. Higher task per core ratios would show better speedups from the templates, as costs are amortized over larger cached blocks per worker. Also, note that logistic regression has a single-stage basic block, as depicted in Figure 3.2. This is the worst case scenario for templates: basic blocks with multiple stages spawn greater number of tasks with a single instantiation message.

6.2.1 Installation Cost

Table 6.1 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, including the overhead of generating the worker template preconditions at the controller. In comparison to scheduling a task (134μs), this cost is small. Installing all templates has an overhead of 36% on centrally scheduling tasks.

Figure 6.1 shows the entire process of installing and instantiating templates in a scenario. The run starts with templates disabled. The control plane overhead of a

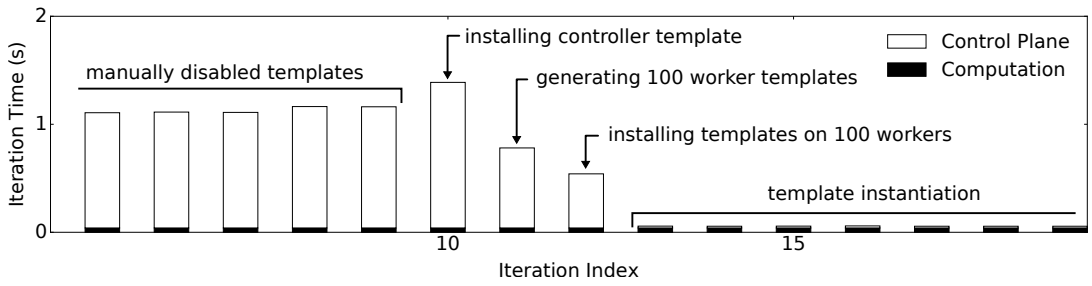


Figure 6.1: Execution templates are installed at runtime. After detecting a basic block, controller installs a controller template, generates worker templates, and then installs the worker templates on the workers. Once installed, templates are instantiated for consequent iterations of the basic block and help reduce the control plane overhead. Here a logistic regression job over 100GB of data executes on 100 workers.

centralized scheduler (134μs per tasks) dominates iteration time: each iteration takes 1.07s. Templates are manually disabled at the beginning to show the performance without templates as a reference point. At iteration 10, the driver starts using templates. Iteration 10 takes $\approx$ 1.3s, as installing each of the 8,000 tasks in the controller template adds 25μs. On iteration 11, the controller template has been installed, and the controller generates worker templates and their preconditions 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, controller sends tasks to and installs templates on the workers. On iteration 13 and afterwards, templates are instantiated with negligible control plane overhead.

6.2.2 Instantiation and Validation Cost

After templates are installed, executing a basic block has the cost of instantiating controller templates and their associated worker templates. Table 6.2 shows the costs of template instantiation for each template type. 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,

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

Table 6.2: 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.

and controller must fully validate the template. When executing the inner loop of a computation, Nimbus can skip validation.

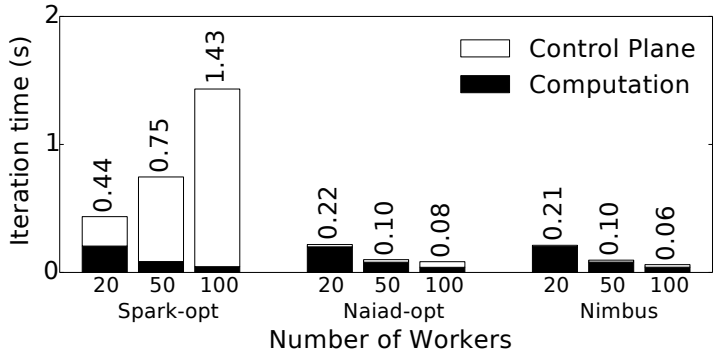
Instantiating a controller template is extremely cheap, 200ns per task. All this involves is copying the controller template structure and filling in the task identifiers and parameters. Instantiating a worker template that automatically validates takes 1.7μs per task, while with explicit validation it takes 7.3μs per task. When executing the inner loop of a computation, Nimbus's scheduling throughput is over 500,000 tasks per second (1 / (0.2μs + 1.7μs)), and when explicit validation is needed Nimbus schedules up to 130,000 tasks per second (1 / (0.2μs + 7.3μs)).

6.3 Data Analytics Benchmarks

This section evaluates the strong scalability of execution templates and it's impact on job completion time. Specifically, it evaluates the end-to-end application performance of data analytics applications, including two machine learning benchmarks, logistic regression and k-means, as representatives of supervise and unsupervised learning algorithms, and a graph processing benchmark, PageRank [100].

6.3.1 Machine Learning

Figure 6.2 shows the results of running logistic regression and k-means clustering jobs over a data set of size 100GB in Spark, Naiad, and Nimbus. As discussed in



(a) Logistic regression

(b) K-means clustering

Figure 6.2: Iteration time of logistic regression and k-means jobs 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 help centralized controller of Nimbus scale out almost linearly and deliver performance similar to Naiad's distributed control plane, while Spark's controller bottlenecks at scale.

Section 6.1, to create a level ground for all frameworks, we consider Spark-opt and Naiad-opt that run tasks as fast as C++ tasks in Nimbus (the back bars). The reported numbers are averaged over 30 iterations; we observed negligible variance in iteration times. The initial iterations are excluded from average to remove the overhead of data loading, JIT compilation in Spark/Naiad, data flow installation in Naiad and template installation in Nimbus. This gives a clear understanding of the steady-state performance without amortizing these one-time overheads over arbitrary number of iterations.

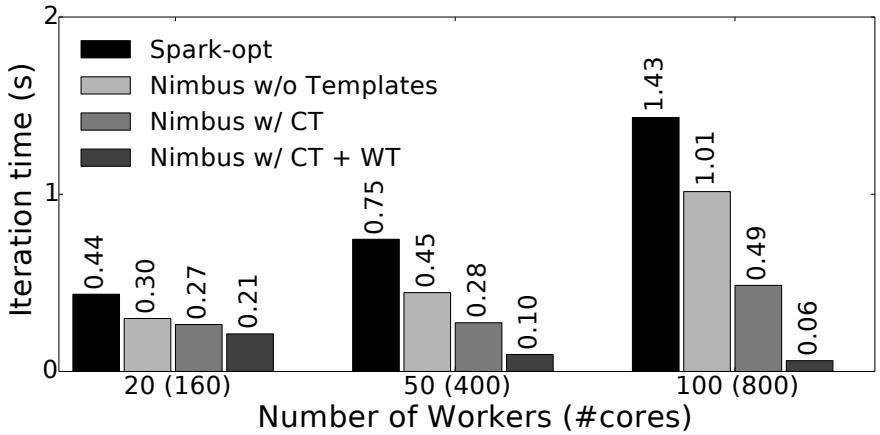


Figure 6.3: Iteration time of logistic regression over 100GB of data in Spark-opt, and Nimbus in three scenarios: 1) without any template, 2) with only controller templates, and 3) with both controller and worker templates. Templates help Nimbus scale; without them Nimbus shows performance similar to Spark, as control plane becomes a bottleneck. Performance benefits of templates are almost split equally between controller templates and worker templates.

Nimbus and Naiad have equivalent performance; with 20 workers, an iteration of logistic regression takes 210-220ms and 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 more slowly 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 more slowly. Over 100 workers, Spark's completion time is 15-23 times longer than Nimbus. The difference is entirely due to the control plane. Spark workers spend most of the time idle, waiting for the Spark controller to send them tasks. In contrast, although Nimbus has a centralized controller as well, execution templates generate and schedule tasks locally similar to Naiad, and so Nimbus does not bottleneck at the controller.

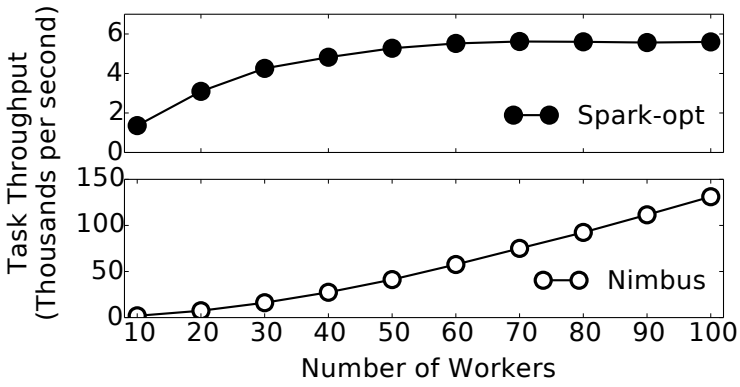


Figure 6.4: Task throughput of Nimbus and Spark as the number of workers increases. Spark saturates at about 6,000 tasks per second, while Nimbus grows at a superlinear rate: more parallelism demands more tasks and simultaneously the tasks become shorter. Note that the y-axis scale is different in the plots.

Figure 6.3 shows how execution templates help remove the control plane bottleneck for Nimbus's centralized controller. It considers three different cases for running under Nimbus: 1) no templates, 2) only controller template activated, 3) both controller and worker templates are activated. As you can see, without templates Nimbus performs similarly to Spark. The benefits from templates are almost equally split between controller and worker templates.

6.3.2 Control Plane Throughput

Figure 6.4 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. This measurement aligns with previously reported numbers [99]. For all measurements we have disabled all internal loggings in Spark: if enabled, the throughput drops to around 1,500 tasks per seconds. Also, the task throughput is directly affected by the size of the Java bytecode generated for the tasks. Writing a driver program with many local variables or large closure size, in case of a Scala program, could increase the bytecode size remarkably. In all experiments, we either used Spark library for application implementations or meticulously wrote the driver programs to avoid large bytecodes. In all case the task bytecode size and accompanied

RDD metadata did not exceed few Kilobytes.

Nimbus, however, 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 per second. It is only about 25% of Nimbus's throughput limits (500,000 tasks per second). Note that the y-scale in Figure 6.4 is different for Nimbus and Spark. In this strong-scaling setup, the ideal task rate grows quadratically, $O(N^2)$, where N is the number of workers. To see why that is the case, note that the tasks rate is the number of tasks in one iteration divided by the iteration time. The number of tasks increases linearly with the number of workers, $O(N)$, and the iteration time drops inversely, $O(1/N)$, since problem size remain the same and individual tasks get smaller. These both affect simultaneously, and lead to a quadratic growth in the tasks rate. In case of Nimbus, the tasks rate increase superlinearly, but it is not perfectly quadratic. This is due to global reduction that cannot be parallelized.

6.3.3 Graph Processing

We consider PageRank [100], as a common graph processing benchmark. This experiment examines what happens when increasing task granularity leads to the network, rather than control plane, becoming the bottleneck. This is a common phenomenon in HPC workloads as they typically do not have a central controller and carefully balance computation with communication [32]. The iterative PageRank implementation has two stages per iteration: a scatter step that updates links (edges) with rank contributions from articles (nodes), and a gather step that collects contributions for every article.

We run PageRank over a graph of English Wikipedia articles and links, similar to the experiment described in Spark paper [128]. The Wikipedia dump [21] contains 12 million articles and 372 million links. We used Metis [16] to partition the graph into 400 (800) edge-cut partitions over 5 (10) workers, using k-way partitioning. Spark's PageRank implementation uses vertex-cut partitioning. PowerGraph [60] shows that vertex-cut partitions perform better on natural graphs with power-law degree distributions. We leave vertex-cut partitioning and other graph optimizations in Nimbus

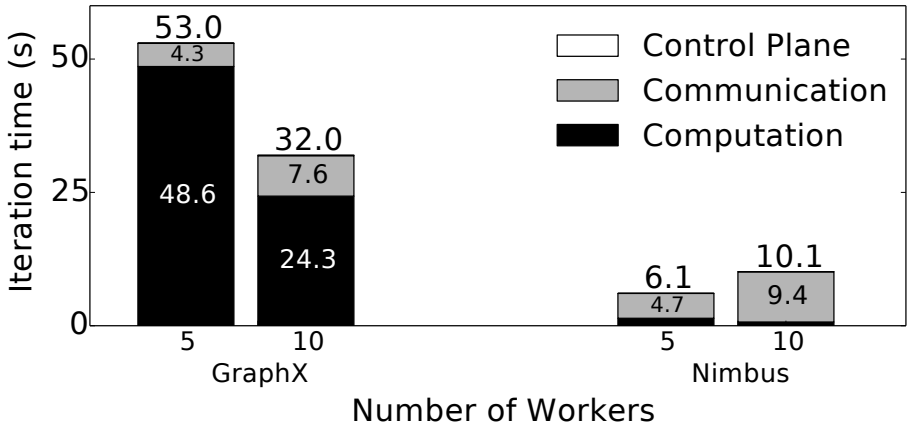


Figure 6.5: One iteration of PageRank in GraphX and Nimbus over a graph of English Wikipedia articles and links. The application with optimized tasks is communication-bound, and not control plane bound. Increasing the number of workers under Nimbus hurts the performance due to the increase in data exchange size. GraphX tasks are implemented in Scala (not optimized) and the application is still CPU-bound. Increasing the number of workers helps execute the CPU-bond tasks in GraphX faster but it sees similar increase in communication time.

for future work.

Figure 6.5 compares the results of running PageRank over 5 and 10 workers using Nimbus and GraphX [61] library from Spark. Individual tasks in Nimbus run 37 times faster than GraphX implementation in Scala (the black bars). This gives an immediate advantage to Nimbus. Note that we could not use the spin-wait trick here to equalize the computation times: actual task executions are required to generate the input for the gather stage. Without explicit computations GraphX would not generate required communication patterns between scatter and gather nodes.

Here, increasing the number of workers from 5 to 10, increases the communication overhead between scatter and gather stages. Increasing the number of partitions means that there are more nodes/edges that end up along the partition boundaries, and hence needed to be exchanged. The similar effect is seen in both GraphX and Nimbus, although communication overhead in Nimbus is slightly worst due to its edge-cut partitioning. For example, with 400 edge-cut partitions, each partition communicates with almost all other partitions, even with k-way partitioning, resulting

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

Table 6.3: A single edit to the logistic regression job takes 41μs Nimbus, and the cost scales linearly with the number of edits. Edits are still less expensive than full installation when migrating as high as 5% of the template's tasks. Any change in Naiad induces the full cost of data flow installation

in over 120,000 messages. As a result, the overall performance hurts under Nimbus, when increasing the number of workers. Faster iteration time in GraphX on 10 workers is only because the application is still in the CPU-bound range and more workers help run the bulky tasks faster.

In both frameworks the control plane overhead is negligible since the jobs become either network-bound in Nimbus, or is still CPU-bound in GraphX. Workers are mainly computing and exchanging data and do not fall idle waiting on the controller. Even though this application is not control bound, templates still help schedule optimized tasks in Nimbus faster. We measured that disabling execution templates slows Nimbus by 28%. Overall, Nimbus with execution templates runs PageRank over 5 times faster than the GraphX implementation.

6.4 Dynamic Scheduling

This section evaluates how well execution templates can support fine-grained scheduling, and keep the cost proportional to the size of changes. Table 6.3 shows per-edit costs for the logistic regression job. A single edit (removing or adding a task) takes 41μs. 800 edits (e.g., migrating 5% of the tasks) takes 35ms, fraction of complete installation cost. Edits allow execution templates to provide fine-grained scheduling without installing new templates for every minor change. For example, we measured

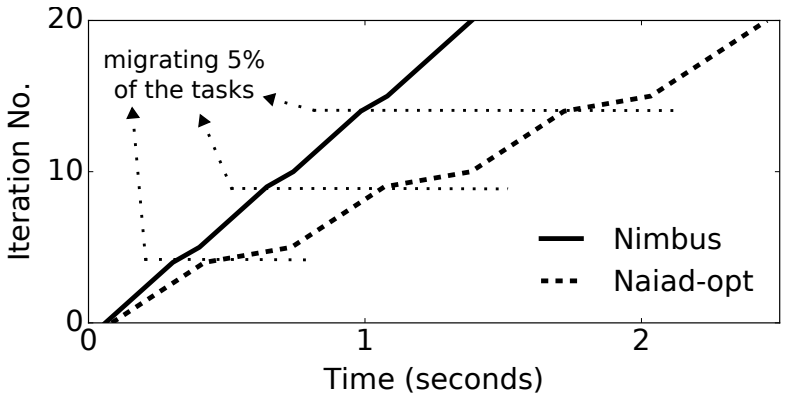


Figure 6.6: Logistic regression over 100 workers with task migration every 5 iterations. Nimbus shows negligible overhead by using edit mechanisms in templates, while Naiad requires complete data flow installation for migrations. Naiad curve is simulated from benchmark numbers, as current implementation does not allow any changes in the dataflow once the simulation starts.

the cost of installing physical graphs on Naiad is about 230ms, which would be induced for any changes.

Despite distributed frameworks that induce complete cost of dataflow installation for every minor change, execution templates show similar behavior as centralized frameworks. For example, Figure 6.6 shows the scenario of running a logistic regression job over 100 workers and migrating 5% of tasks every 5 iterations. Nimbus migration overhead is negligible, while Naiad requires complete dataflow installation for any change in scheduling. Note that, current Naiad implementation does not support any dataflow flexibility once the job starts, so the curve here is simulated from the numbers in Table 6.3 and Figure 6.2(a). The incremental edit cost allows Nimbus to finish 20 iterations almost twice as fast as Naiad.

Edits allow controllers to inexpensively make small-scale changes to worker templates. Figure 6.7 shows how the cost of edits compares to the cost of reinstalling templates. The cost of edits increases linearly with the number of edits involved. However, because the cost of an individual edit (41μs) is greater than the cost of installing a task in a worker template (29μs), when the change is large enough it is faster to install a new template. Note that extra cost of edits is due to the necessary

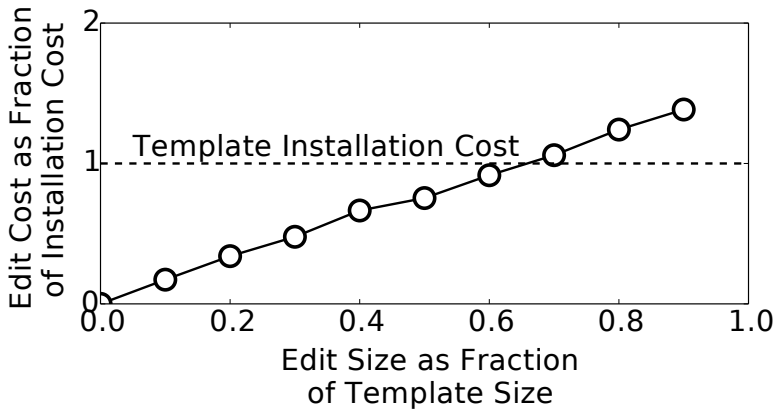


Figure 6.7: Edits support fine-grained scheduling because the cost of edits is small and scale linearly with the number of edits. Installing a new template is more efficient for large changes.

changes in the task graph and inserting extra copy tasks as depicted in Figure 5.10. Furthermore, since edits operate in place, they do not allow quickly switching between very different schedules, as discussed below in Section 6.5. Heuristics on when to edit versus install new templates are an area of future work.

6.5 Dynamic Resource Allocation

Workers save multiple versions of a worker template, according to various task partitioning, and cluster resource allocations for a job. The worker template is saved as a vector of indices for task and data identifiers in the execution plan and takes ≈1KB of metadata per core. This is orders of magnitude smaller than the in-memory data object that cores access for task execution. So workers can cache many worker templates with negligible overhead and controller can instantiate them according to the cluster manager decisions.

Figure 6.8 shows a scenario in Nimbus where the cluster manager adjusts the available resources. This scenario builds on the run in Figure 6.1: templates are already installed and being instantiated over 100 workers in the cluster. At iteration 20, the cluster resource manager revokes 50 workers from the job's allocation. On

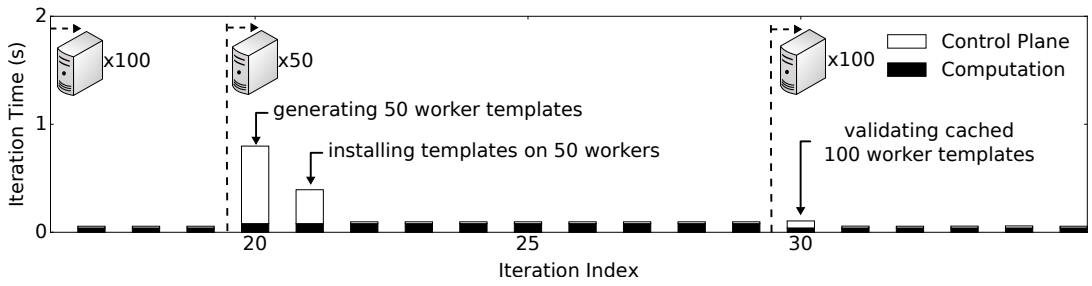


Figure 6.8: Execution templates can schedule jobs with high task throughputs while dynamically adapting as resources change. This experiment shows control overheads as a cluster resource manager allocates 100 nodes to a job, revokes 50 of the nodes, then later returns them. Workers can install different versions of templates, and controller can validate and instantiate them quickly, according to the available resources.

this iteration, the controller regenerates the worker templates and their preconditions, migrating tasks from evicted workers to the remaining half of workers. On iteration 21, the controller installs newly generated worker templates for the 50 workers. Later iterations instantiate the templates. Note that compared to the 100 worker cluster, computation time doubles because each worker is performing twice the work. To disambiguate the control plane and data migration overhead, the data sets on the evicted workers (50GB) are already loaded on the other half of workers (evenly among them).

At iteration 30, the cluster resource manager restores the 50 workers to the job's allocation. The controller reverts back to using the original worker templates and so does not need to install new templates. However, on this first iteration, it needs to validate the templates. As listed in Table 6.2,the instantiation cost with explicit validation in the first iteration (7.5μs per task), is higher than auto validation cost for later iterations (1.9μs per task). Accordingly, after slightly higher control plane overhead at iteration 30, each iteration time drops back to 60ms for later iterations.

Chapter 7

Graphical Simulations in Nimbus

To evaluate if execution templates can handle full applications with complex control flows and data flows, this chapter presents the details of running graphical simulations in Nimbus. The chapter starts by presenting a geometric-based data model that allows distributing the sequential kernels of simulation libraries automatically, by writing a simple driver program in Nimbus. It continues by evaluating the effects of running graphical simulations at scale on the control plane of Nimbus. It shows how execution templates enable Nimbus to support the task throughput requirements of these applications while adapting to their complex control flow with nested loops and data dependent branches.

Graphical simulation is a cornerstone of modern animation and special effects. These simulations are computationally intensive and so could greatly benefit from elastic scalability of the cloud. On-demand cloud computing has made high-performance computing clusters immediately available to anyone at very low cost. A `c3.2xlarge` Amazon Elastic Compute Cloud (EC2) instance with 8 virtual cores, 15GB of RAM, and 1Gbps full bisection bandwidth costs 40¢/hour [3]: running an 800-core simulation for an hour costs $40.

Despite all of its benefits, however, cloud computing has been mostly untapped by simulations. Simulations are hard to distribute. They use multiple geometric data representations, coupled through an intricate series of computational steps. A particle

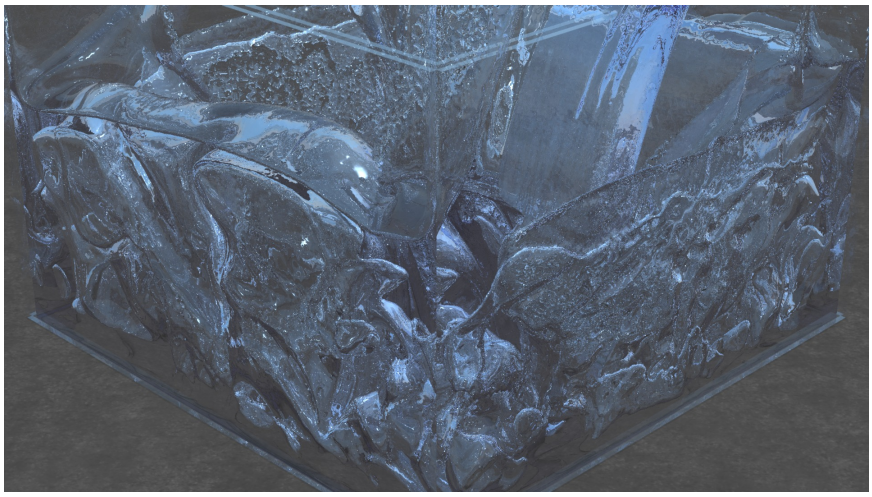


Figure 7.1: Still of a particle level-set water simulation.

level-set water simulation, as shown in Figure 7.1 for example, consists of a triple-nested loop with over a dozen different operations. It maintains an Eulerian MAC grid for the volume, four different sets of particles in a thin band at the interface, and a sparse matrix for solving the implicit terms of the Navier-Stokes equations [51]. Distributing this simulation requires rewriting each computational step to use low-level networking interfaces (such as MPI) and distributed debugging.

This chapter presents how Nimbus distributes an existing sequential grid-based or hybrid (e.g., particle level-set) simulation library across many multi-core nodes. A driver program, written by the simulation author is a simple, sequential program. Controller translates the driver program into individual tasks and distributes them across many workers. Workers receive tasks from the controller that invoke simulation library functions. Workers are unaware that they are part of a distributed computation and run seemingly stand-alone simulations. Nimbus automatically updates state shared between these sub-simulations, stitching them together into a single larger one. To support an existing simulation library, a library developer writes a small number of adapters that translate between the library calls and Nimbus's APIs.

To run a large-scale distributed simulation while providing a sequential execution model to the programmer, we propose 4 layers of data abstraction for Nimbus. In

the top layer, the driver executes simulation functions over *geometric* data objects, consisting of a simulation variable over a bounding box. To efficiently handle ghost regions and other shared subregions, the second layer subdivides the domain into disjoint *logical* objects. Logical objects present an abstraction of a large, shared memory to the driver program, allowing it to easily analyze and encode parallelism and data dependencies independently of how the simulation is distributed. *Physical* objects are actual data objects on compute nodes and are the unit of transfer between nodes. Each logical object can have multiple physical instances, residing on multiple nodes. The controller translates the logical commands received from the driver into physical commands for compute nodes. The logical/physical separation allows the controller to abstract away distribution, load balancing, and fault tolerance from the driver program. Finally, *application* objects are in the format and layout that the simulation library expects. Application objects are typically composed of many physical objects, as they include central as well as ghost regions. Nimbus detects inconsistencies between application objects, automatically updating them locally or over the network when needed.

7.1 Graphical Simulations

Graphical simulations use different data models and algorithms than what available cloud frameworks provide. This section gives an overview of the principal methods and algorithms used in graphical simulations, and explains the challenges of distributing these computations over multiple nodes.

As a concrete example of a graphical simulation, we focus on PhysBAM [51], an open source physics based software package for fluid and rigid body simulations. Movie studios such as ILM and Pixar use PhysBAM in production films, and the developers have won two Academy Awards for its contributions to special effects [20]. PhysBAM can simulate a huge range of phenomena, but in the rest of this chapter, we focus on a water simulation. Water simulation is a canonical example, as it is extremely difficult and employs methods that are required for other fluid simulations such as smoke and fire.

7.1.1 Fluid Models and Simulation Algorithms

There are two basic ways to computationally represent a fluid: a grid or particles. A grid divides simulated volume into cells. Per-cell state describes the state of the simulation, such as whether it contains fluid, pressure, and velocity. The second approach is to represent the fluid as a set of particles, each of which has its own (x, y, z) coordinates, velocity, and size. Grids and particles have different strengths and weaknesses. For example, a grid smooths out small ripples but does not model splashes well, while particles have difficulty representing fixed boundaries such as the edge of a glass.

The particle-levelset method [52], pioneered by PhysBAM, combines particle and grid representations and is why movie and special effect studios can simulate water, smoke, and fire today. The key insight is that the most important visual feature is the surface of the fluid. The particle level-set method uses a coarse grid, augmented with dense particles *only on the surface*. Since the number of particles increases with the square (surface), not cube (volume), of the grid size, it remains a small fraction of the simulation state. Combining these two methods, however makes simulations much more complex, as the grids and particles interact in subtle and interesting ways. For example, particles that leave the surface become drops in a splash, and must be correctly merged back with the water mass when they hit the surface again. Readers interested in a more complete description of the complexities can read the seminal book on the topic by Bridson [38].

A simulation is a loop: each iteration steps time forward. The length of the time step is determined by fluid velocity and grid resolution, so that fluid does not seem to leap through space. When time passes a frame boundary, the simulation outputs the visual state of the simulation for later rendering. An iteration has 22 distinct computational steps, which can be divided into three major categories: updating grid cells, updating particles, and solving a set of linear equations that enforce physical laws on the water (e.g., it does not compress or disappear). Solving the linear equations uses a sub-loop within the main loop. In a typical 256^3 water simulation, there are on average 20 main loop iterations per frame (24fps means 42ms/frame, the main loop time step is 2.1ms) and 100 iterations of the inner solver loop.

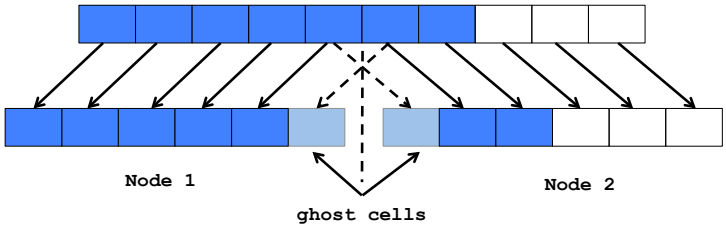


Figure 7.2: A 1D row of water represented in a grid. When partitioned across two processes, the two processes must exchange *ghost cells* of state so they can perform computations locally.

7.1.2 Current Distributed Simulations

Running a simulation across multiple nodes requires partitioning the simulation geometry across them. The basic challenge is that partitions are not independent. The state of water at any cell is dependent on its neighboring cells, some of which may be on a different node. Furthermore, solving the linear equations involves global operations.

Partitions can be distributed while minimizing data sharing with *ghost cells*. Consider a simple 1D simulation of a pipe with water, shown in Figure 7.2. Each partition is divided into five parts per axis: a large, central region that only the local computations need, two thin regions of ghost cells that are sent to neighbors, and two thin regions of ghost cells that are received from neighbors. Figure 7.3 shows a partition in a 1D and a 2D grid. For a 3D simulation, a partition consists of 125 separate regions (5^3). Each variable is partitioned in this manner, resulting in over 29 thousand data objects for 16 partitions, in a typical simulation with 21 different variables. In addition to computing on particles and grid cells, simulations also need to perform global reductions. For example, to compute the time step, or the residual of the linear solve, the simulation takes the maximum value across all of the partitions.

When a computational step (e.g., reseeding particles) completes, that node needs to send the updates it made to local ghost regions and receive updates for remote ghost regions. PhysBAM does this in lockstep: each worker process completes its computation, sends its results, then blocks on receiving results from neighbors. This

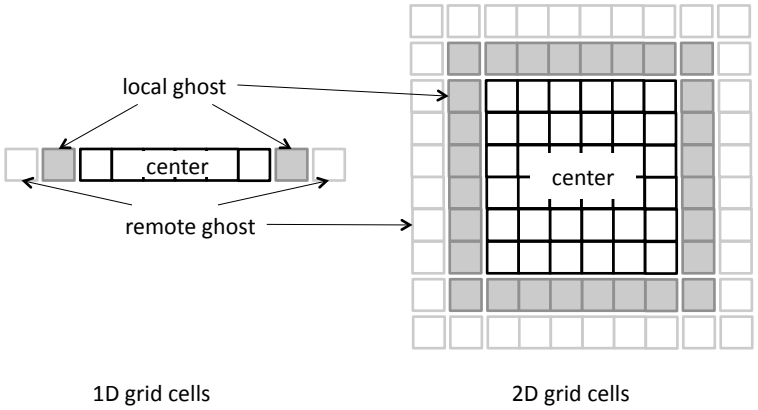


Figure 7.3: Ghost cell configurations in simulation grids. The local state on a node consists of 3^d objects, where d is the dimensionality of the grid, while the combined local and remote state a node must use consists of 5^d objects. A 3D grid is not shown for visual simplicity: per-node state is 27 objects and the total state is 125 objects.

approach tightly couples the control flow of the program with its state exchange. Furthermore, the partitions are set up statically at compile time and cannot move. If one node fails, the entire simulation fails. The simulation can run only as fast as the slowest node in the cluster, so stragglers are a major concern.

7.2 Simulation Data Abstraction in Nimbus

Nimbus provides a powerful data model to hide the complexities of distributed execution from simulation libraries, written by library developers, as well as a simulation main loop, written by a simulation author. A simulation main loop operates on simulation state as a volume, applying functions over bounding boxes. This provides a simple *geometric* data abstraction to a simulation author. The simulation library continues to use its own *application* data types and representations.

To translate from geometry in the driver program to application data in the simulation library, Nimbus uses two intermediate representations. The first, *logical* objects, disjointly subdivide the simulation domain into smaller bounding boxes that precisely define how data is accessed and shared. Logical objects allow Nimbus to analyze a

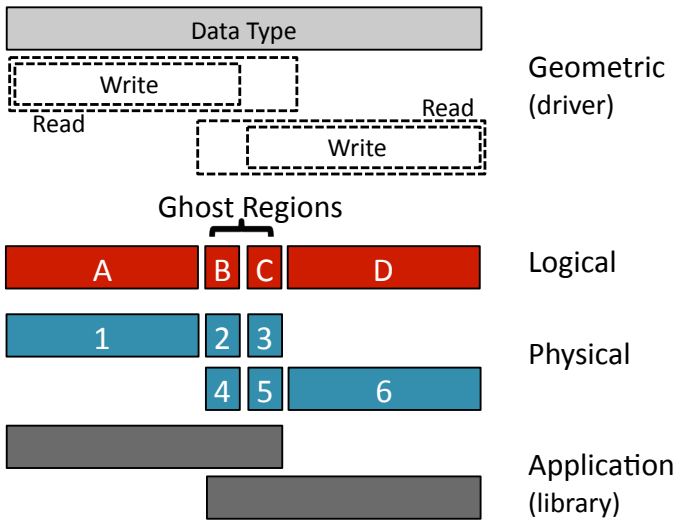


Figure 7.4: The Nimbus data model has four abstractions: geometric, logical, physical, and application. This example shows a 1D advection stencil to two partitions on two different nodes. The geometric view sees the complete simulation domain and applies the stencil to the two partitions. This defines 4 logical objects, which map to 6 physical objects on the two nodes. Nimbus assembles these disjoint physical objects into contiguous application objects before invoking simulation library functions.

simulation's execution in an abstract representation that is independent of its distribution; it uses these logical objects to determine which operations can be run in parallel and which must be ordered.

The second intermediate representation, *physical* objects, describes exactly how data is distributed across the system. Every logical object has one or more associated physical objects. In Figure 7.4 the central regions (objects A and D) each have a single physical instance, while the ghost regions (B and C) each have two physical instances, one on each node. Physical objects allow Nimbus to explicitly manage data placement and copies. Nimbus automatically generates and maintains application objects from their constituent physical objects. The rest of this section explains these four abstractions in greater detail. Section 7.3 describes how Nimbus uses each one and translates between them.

7.2.1 Geometric View

A simulation declares a variable as a data type over the simulation domain with a geometric partitioning and a ghost cell width. The simulation library defines the set of available types, such as floating point vectors, particles, or scalars. The simulation loop is a linear sequence of simulation library calls, such as advecting particles or calculating boundary conditions in projection. Each library call specifies its data reads and writes by a set of {variable, bounding box} pairs. The 1D stencil in Figure 7.4 for example, makes two library calls, one on the left bounding boxes and one on the right bounding boxes:

```
parallel {
    apply(advect, {var, lread_bb}, {var, lwrite_bb});
    apply(advect, {var, rread_bb}, {var, rwrite_bb});
}
```

This provides a simple, intuitive interface for the simulation author.

7.2.2 Logical View

Using the specified partitioning, Nimbus automatically translates each variable into a set of disjoint logical objects. Logical objects divide the simulation domain into the largest bounding boxes, which have the same sequence of reads and writes. In Figure 7.4, there are four logical objects. Object A is read and written by the left stencil, object B is read and written by the left stencil and read by the right stencil, object C is read and written by the right stencil and read by the left stencil, and object D is read and written by the right stencil. Nimbus transforms the two library calls into:

```
parallel {
    apply(advect, {A, B, C}, {A, B});
    apply(advect, {B, C, D}, {C, D});
}
```

Logical objects define a read/write order. Each object has a linear sequence of writes, with any number of parallel reads between a pair of writes. This allows Nimbus to analyze data dependencies between calls and enforce execution order. For example, as the two advection calls above are in parallel, Nimbus knows that both of them read the same data in B and C and they can execute in parallel. However, the next time this block executes, Nimbus knows that the calls read the output of the prior invocations, and so cannot run until the prior invocations complete. In simulation steps where ghost regions have parallel writes (e.g., particle advection), Nimbus transforms the operation into a linear sequence by creating one temporary object per writer and reducing them to the result.

Logical objects are independent of how the simulation is distributed. They present the abstraction of a single large, shared memory. This allows Nimbus to ensure correctness without considering the complexity of the current state of a particular execution.

7.2.3 Physical View

Each logical object has one or more associated *physical* objects. Physical objects are an actual simulation state, stored in memory on specific nodes: they define how data is distributed. Figure 7.4 shows how, when distributed across two nodes, the 4 logical objects map to 6 physical objects, 3 on each node ({1,2,3} and {4,5,6}). The logical library calls above are translated into these physical library calls:

```
apply(advect, {1, 2, 3}, {1, 2});
apply(advect, {4, 5, 6}, {5, 6});
```

Nimbus tracks the state of a physical object with a version number. The version is determined by the write lineage in the main loop. When it completes, a library call that writes to a logical object increments the version for the object. All subsequent calls use this version as input. Unlike the logical view of objects, which maintain the abstraction of a sequential execution of parallel operators, the physical view describes their actual state as the execution progresses. Physical objects are stored

as contiguous blocks of memory, optimized for network transfer while minimizing fragmentation.

7.2.4 Application View

Simulation computations are written over contiguous application objects for each partition. An application object over a partition holds data corresponding to underlying system objects from its partition and ghost regions from neighboring partitions. Application objects have the types and formats of the simulation library. For example, a PhysBAM application object can refer to scalar or vector arrays, or particles stored as linked lists of particle buckets.

Nimbus assembles application objects by interleaving data from physical objects. Nimbus uses the geometry metadata of the physical objects to interleave them correctly. Simulation library functions over these application objects can then be used directly, without rewriting the code to use a different data model. Figure 7.4 shows how the two nodes each assemble a contiguous application-level object out of 3 disjoint physical objects. Both nodes execute

```
advect(sim);
```

As the call above shows, the simulation library executing on the node is unaware that its application object can be modified by other threads, cores, or nodes. Instead, when a library routine completes, Nimbus automatically updates underlying physical objects, stitching together the seemingly independent simulations. For example, the task writing to logical object 1 will need the writes to object 5 the next time it runs. Similarly, the task writing to logical object 6 will need the writes to 2. Nimbus, ensures that 5 is copied to 3 and 2 is copied to 4 before the next execution.

7.3 Translation Among Data Abstractions

Nimbus translates between geometric, logical, physical, and application views of the data using the five architecture components shown in Figure 7.5. A driver program

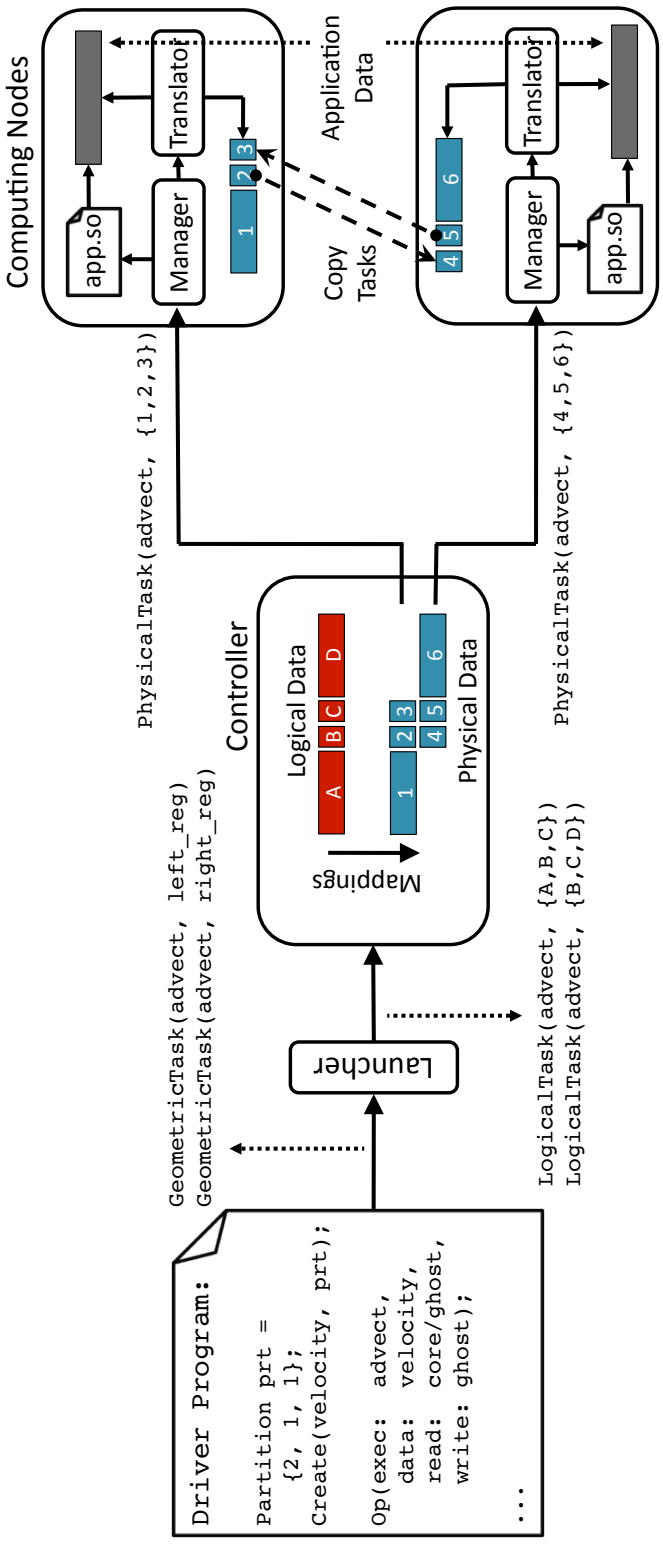


Figure 7.5: Data translation overview in Nimbus. A driver program defines a serialized lineage of operations over geometric data. The launcher turns the driver program into a series of parallel tasks over logical data objects and sends them to the controller. The controller assigns tasks to computing nodes for execution, mapping logical objects to specific physical instances. Automatically inserted copy operations synchronize physical instances when needed. The translator on each computing node assembles application objects out of physical objects, and the manager controls a thread pool to execute library functions over application objects.

describes the main simulation loop in the geometric view. The *launcher* specifies the logical tasks that each operate on a subset of the simulation domain, defined by logical objects. The launcher sends these logical tasks to a *controller*, which binds the logical objects to physical objects, computes dependencies, and sends task execution commands to workers. The *manager* running on a worker receives task execution commands, schedules them based on their dependencies, and invokes simulation library code. The *translator* on workers assembles the disjoint physical objects into application objects and keeps the both views consistent with a write-back cache.

7.3.1 Launcher

The initialization phase of the driver program specifies the simulation variables, geometric domain, and a partitioning configuration. Nimbus uses this information to generate the set of logical objects. Using logical objects separates data declaration from instantiation, placement, and layout.

Driver program also specifies the simulation operations over partitions and whether each variable they access is needed read-only or read/write. They also specify the geometric domain of access. The geometric domain defines how the operation accesses ghost values. Reverse semi-Lagrangian and Eulerian operations read ghost regions, while Lagrangian operations write ghost regions.

The launcher expands operations in the driver program into waves of *tasks*, assigning one task to each partition's geometric domain. Each task has a *read set* of the logical objects it reads and a *write set* of the logical objects it writes. To compute these sets, the launcher looks up the logical objects associated with each partition. It relaxes the sequential order of the driver program by computing data dependencies between tasks and adding a *before set* to each logical task, which specifies what tasks must complete before this task can safely run. The before set is calculated from read and write sets, enforcing that every task sees the most recent write in the driver's sequential order. The before sets define the *logical task graph*, a DAG which encodes the simulation's parallelism.

For large, distributed simulations with hundreds of partitions and millions of logical objects, computing and sending logical tasks can become a bottleneck. The launcher therefore caches the tasks it generates in controller templates; since simulations typically involve iterative loops with regular access patterns, they have both regular control flow and data flow. Cache misses only occur the first time a control path executes (e.g., the first iteration, the first time a frame is written to disk, etc.). The controller also caches logical tasks; a cache hit allows the launcher to send tens of thousands of tasks to the controller with a single, small message.

7.3.2 Controller

The controller takes the logical parallel program specified by the launcher and distributes it across computation nodes. It maintains a *context* for each logical task, which specifies the version number of each logical object. A context is computed by assigning, to each logical object, the maximum version of the contexts of all tasks in the before set. A task that writes to a logical object increments the version number; this newer version propagates to tasks that have it in their before set.

The controller instantiates physical instances of logical data at computing nodes. A physical instance is a contiguous block of memory that encodes the variable over a domain. Each instance has a version number. The controller takes task graph it receives from the launcher and transforms it into a *physical task graph*, which contains execution commands that invoke simulation library functions on specific nodes. Vertices of the physical task graph, like the logical task graph, have a read set, write set, and before set. The read set and write set are physical objects, however. For iterative applications controller memoizes this translation in worker templates.

When deciding how many physical instances to create, the controller trades off between memory footprint and parallelism with a simple heuristic. It makes copies of ghost regions but keeps a single instance of any central region. For the uniform partitioning of 256^3 simulation in Figure 7.1, for example, a 3-wide ghost region has 9-10,092 cells, while a central region has 195,112 cells.

7.3.3 Translator

The *manager* receives tasks from the controller and manages three task queues: blocked (on an incomplete member of the before set), ready, and running. It maintains a subset of the physical task graph, consisting of the tasks received from the controller. When a task completes, the manager removes it from all before sets; if this makes a before set empty, the task transitions to the ready queue. The manager maintains a thread pool equal to the number of available cores. When a task completes, it takes the next task of the ready queue and runs it. The ready queue uses two priorities, with each priority having a FIFO order. Copy tasks have higher priority. This starts asynchronous network transfers as early as possible, interleaving communication and computation.

Before the manager executes a simulation library function, it invokes the *translator* to generate the appropriate application objects. If there is already an application object whose data has the correct versions, the translator returns immediately. If there are portions of the object that are out of data (e.g., a ghost region written to by another node), it fills into the application object the content of the physical object specified in the task.

To prevent unnecessary copies for data that is only used locally, and to remove the need for two copies of central regions, the translator uses a write-back strategy. If a task writes to an application object, the translator does not immediately write out the result to the corresponding physical objects. Instead, it waits until a copy task reads the physical object, at which point it writes the result out to the physical object before the transfer starts. The translator frees the backing memory of physical objects that are out of date with their application object. Because central regions are only transferred when Nimbus load balances between compute nodes, there is only one copy of their data. This allows Nimbus to maintain both the system and application views with only a small ($< 10\%$) memory overhead.

```
1 class ScalarArray: public AppVar {
2   PhysBAMScalarArray *data ();
3   void Read(DataArray objects, BBox box);
4   void Write(DataArray objects, BBox box);
5   // Internal data members
6   Region bounding_box;
7   PhysBAMScalarArray *data;
8 };
```

Listing 7.1: Type definition for a float array application object.

7.4 Writing Simulations in Nimbus

Writing a simulation in Nimbus has two parts. First, a simulation library developer writes *adapters* that allow Nimbus to properly translate between the application and system views and *compute tasks* that encapsulate library function calls. This is a one-time effort. Second, the simulation author writes the driver program that specifies the variables and computational steps of a specific simulation.

This section explains Nimbus's API, using level-set advection as a running example. The application simulates multiple frames, and assumes one iteration per frame for simplicity (PhysBAM dynamically selects a *dt* inversely proportional to the maximum velocity in order to maintain the CFL condition). It explains both the APIs used for adapters as well as a simulation driver.

7.4.1 Simulation Types (Library Developer)

For Nimbus to translate between the application and system views, a simulation library developer has to write an adapter for the translator to convert them. This adapter is a class that holds the underlying application data. Listing 7.1 illustrates this API for a scalar array in PhysBAM (e.g., the signed distance). The `Read` method reads out of the application object, translating it into physical objects. The `box` parameter specifies a bounding box subset of the application object to copy. Each element of `objects` also has a bounding box; the data copied is the intersection of the application object bounding box, the physical object bounding box, and `box`. `Write`

```
1  class AdvectLevelset: public ComputeTask {
2    void Execute() {
3      // Retrieve application objects to compute on
4      float& dt = GetAppObject("dt");
5      Vec3fArray& velocity = GetAppObject("velocity");
6      FloatArray& signed_distance =
7        GetAppObject("signed_distance");
8      // Call into the PhysBAM library to run advection
9      PhysBAMAdvectLevelset(velocity, dt, signed_distance);
10   }
11 };
```

Listing 7.2: Compute task for advecting levelset calls into PhysBAM library.

copies from physical objects into the application object.

7.4.2 Compute Tasks (Library Developer)

Compute tasks encapsulate library function calls. They are responsible for setting any global variables or configuration that the simulation library expects. They also fetch the application objects from the translator that the library function needs. Listing 7.2 shows simplified code for the `AdvectLevelset` compute task. The compute task reads `velocity` and `signed_distance` over its partition and ghost regions from neighboring partitions, advects `signed_distance` with a call to `PhysBAMAdvectLevelset`, and writes to `signed_distance` in its partition.

7.4.3 Control Tasks (Simulation Author)

The driver program has three parts, shown in Listings 7.3–7.5. The first part (Listing 7.3) defines the parameters of the simulation, including the geometric domain, partitioning, and ghost cell widths. This initialization is the one point in the program when a simulation author must consider how to distribute the simulation. Unlike HPC simulations, which tend to perform a uniform computation over data, graphical simulations can have highly varying computations. For example, particle level-set water simulations perform far more computations on water cells than air cells, and water

```
1  // Define simulation domain
2  BondingBox sim_region = {{0,0,0},{256,256,256}};
3  // Partition the domain along three axes
4  Partitioning partitioning = {2,2,1};
5  // The width of the ghost region
6  int ghost_width = 2;
7  // A center region with ghost width 0
8  Region center({sim_region, partitioning, 0});
9  // Outer region includes center and ghost regions
10 Region center_plus_ghost(
11   {sim_region, partitioning, ghost_width});
```

Listing 7.3: Example initialization for a particle level-set simulation over of a 256^3 domain partitioned along the X and Y axes.

```
1  // Entry point for a simulation
2  class Main: public ControlTask {
3    void Execute() {
4      CreateData("signed_distance", FloatArray, sim_region,
5                 ghost_width, partitioning);
6      CreateData("velocity", Vec3fArray, sim_region,
7                 ghost_width, partitioning);
8      // Create more objects and launch Loop task
9    }
10 };
```

Listing 7.4: Main task for example particle level-set simulation.

cells on the interface are more computationally intensive than those deep within the volume. As a result, the optimal partitioning depends not only on the type of simulation, but also its initial conditions, and so this is best controlled by the simulation author.

The driver program is written as a *control task*. Control tasks do not directly invoke simulation functions; instead, they launch other compute and control tasks. A special control task, `Main` (Listing 7.4), is the entry point for simulation. Main defines the simulation variables. These variable definitions create logical objects at the controller. The controller does not create physical objects on compute nodes until it sends tasks to read or write them. This lazy instantiation allows the controller to

```
1  // A control task to loop until target_frame
2  class Loop: public ControlTask {
3    void Execute() {
4      // Spawn parallel AdvectLevelset tasks
5      LaunchTaskOverAllPartitions(
6        AdvectLevelset,
7        {{"signed_distance", center_plus_ghost},
8         {"velocity", center_plus_ghost}},  // read outer
9        {{"signed_distance", center}},  // write center
10       {}});
11     // Spawn other tasks to advect velocity, save data
12     // Spawn next iteration if needed
13     if (parameter.current_frame < parameter.target_frame)
14       LaunchTask(
15         Loop, {}, {}
16         { .current_frame = parameter.current_frame + 1,
17           .target_frame = parameter.target_frame });
18   }
19 };
```

Listing 7.5: Loop task for particle level-set simulation.

distribute data only after it has a full picture of task access patterns on data objects.

Main, after it launches tasks to initialize the simulation, it launches a `Loop` task, which corresponds to the outermost simulation loop. This task launches compute tasks `AdvectLevelset` and `AdvectVelocity` (Listing 7.5) to compute the next values of `signed_distance` and `velocity`. It uses parameters `current_frame` and `target_frame` to determine the end of a frame. If there are more frames to simulate, it launches another `Loop` control task.

When launching compute tasks, a control task must specify the data that the compute task reads and writes. For instance, the `Loop` task specifies read and write sets for `AdvectLevelset` on lines 5 to 10. It does this by specifying the variables and the partitioning to use for reading and writing – an `AdvectLevelset` task over a partition writes to `signed_distance` only over the same partition, by specifying `center` as the partition to write, but reads `signed_distance` and `velocity` from ghost regions over neighboring partitions. Nimbus automatically infers write after read dependencies (e.g., from `AdvectLevelset` to `AdvectVelocity`), and read after write dependencies

(from `AdvectVelocity` to `AdvectLevelset`). Based on these dependencies, it automatically inserts the necessary copy tasks and adds them to before sets, enforcing the correct execution order.

7.5 Evaluation

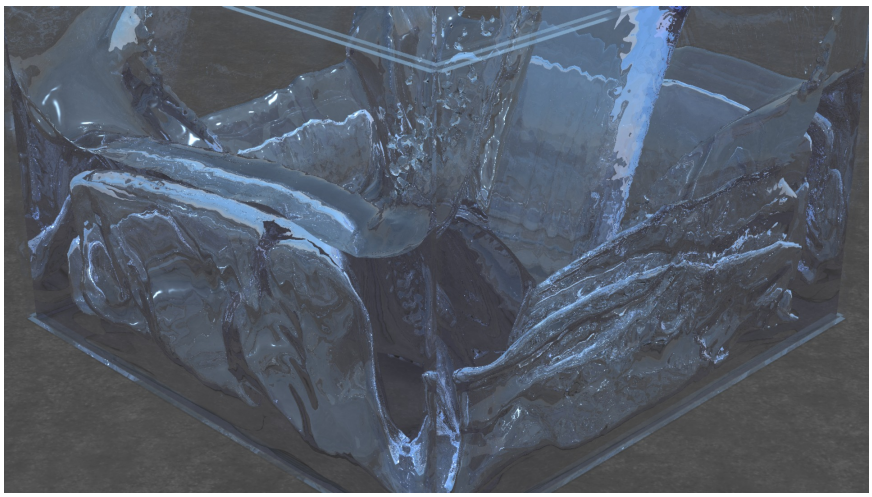
This section presents the results of porting and running PhysBAM fluid simulations [51] under Nimbus. The performance of the system is compared against hand-tuned distributed implementation of the simulation with MPI [113]. This section also explores the feasibility of porting graphical simulations to Nimbus.

Figure 7.6 shows simulated frames of a particle-levelset water simulation from PhysBAM library. The top image depicts the 128^3 cell simulation frames running without Nimbus serially. The bottom image shows the 256^3 cell simulation frames running with Nimbus, distributes over 64 cores. The 256^3 simulation takes 268 minutes under Nimbus faster than it takes to run the smaller simulation serially. Without Nimbus the 256^3 simulation takes more than two days. The speedup finishes bigger simulations in reasonable time, which brings noticeable difference in details.

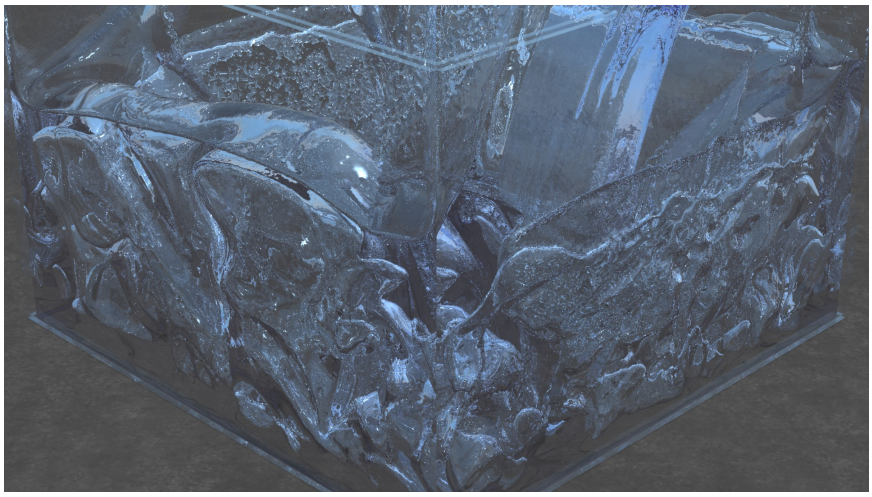
7.5.1 Scalability and Speedup

Nimbus speeds up simulations by scaling out over multiple nodes. Figure 7.7 shows the results of running a 256^3-cell simulation under Nimbus with three different settings: 1) serialized version over a single core, 2) distributed over 8 cores of a single node, and 3) distributed over a cluster of 8 nodes, each with 8 cores (64 cores in total). For each setting the average length of the main iteration loop is reported. Once an application is ported to Nimbus, the runtime automatically parallelizes the simulation depending on the available resources.

Each simulation iteration is composed of an explicit solver, and an iterative preconditioned conjugate gradient (PCG) implicit solver. While the explicit solver scales almost linearly with more cores, the iterative implicit solver scales sublinearly. This



(a) 128^3 cells, with Nimbus: 43 minutes, without Nimbus: 172 minutes



(b) 256^3 cells, with Nimbus: 268 minutes, without Nimbus >48 hours

Figure 7.6: Particle level-set water simulations with and without Nimbus. The top simulation has 128^3 cells, runs on a single-core and takes 172 minutes to simulate 30 frames. The bottom simulation uses Nimbus to automatically distribute this single-core simulation over 8 nodes (64 cores) in Amazon's EC2, simulating with greater detail: 30 frames of a 256^3 cell simulation takes 268 minutes. Without Nimbus the 256^3 cell simulation takes more than two days. Running the 128^3 simulation in Nimbus takes only 43 minutes.

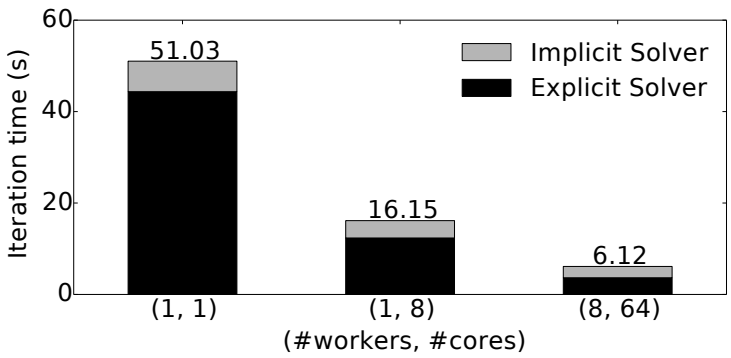


Figure 7.7: Running 256^3-cell PhysBAM water simulation under Nimbus with three different cluster settings. For each setting the length of main iteration is reported (averaged over 1200 iterations). Nimbus automatically parallelizes the simulation over more resources.

# partitions	iteration time	# iterations	total time
1	111 (ms)	60	6.67 (s)
8	46 (ms)	82	3.79 (s)
64	25 (ms)	98	2.48 (s)

Table 7.1: Performance of the iterative PCG implicit solver for various parallelism settings. While increasing partitions speeds up each iteration of the solver, the preconditions become less effective resulting in more iterations before convergence.

is because locally computed preconditions become less effective with increasing partitions, resulting in more iterations for the implicit solver. Table 7.1 shows the average number of iterations and the time it takes for the solver in different settings. Overall, parallelism over 64 cores results in a 8.3x speedup compared to the serialized version.

Comparison against MPI

Currently, the common approach in distributing the graphical simulations is through MPI [113]. Application writers leverage the MPI interface to implement the data exchange required for distributing the application state. The control and data exchange logic is interleaved within the application logic and statically compiled. This approach requires intensive coding effort by the developers. However, the application

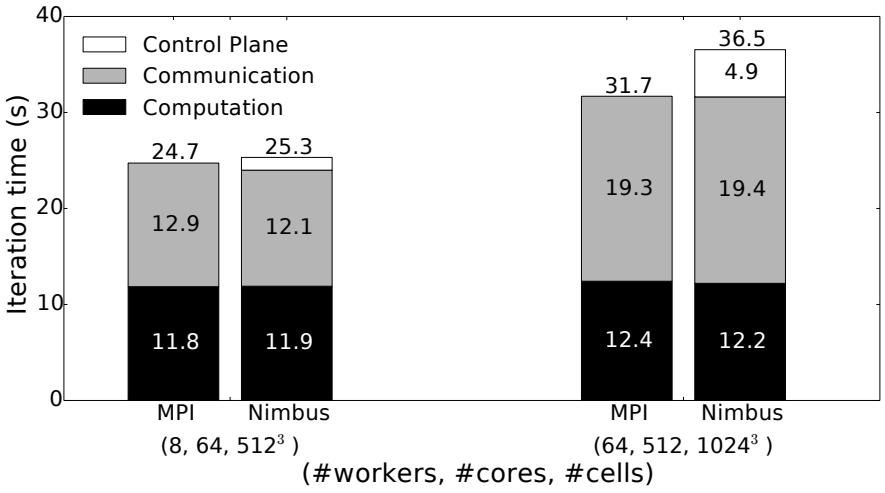


Figure 7.8: Execution time of a PhysBAM water simulation using MPI and Nimbus. For the standard simulation size used today (512^3), Nimbus imposes an overhead of only 3%. For the largest PhysBAM water simulation ever run (1024^3), the overhead is 15%. The overhead Nimbus imposes is entirely from a central controller; the computation and communication time matches the MPI performance closely.

specific implementation enables hand-tuned optimizations in favor of performance. The statically compiled control flow induces minimum overhead during the runtime, although it is not robust against failure and load imbalance.

Nimbus, on the other hand, provides an application independent interface. The control plane is dynamically planned out during the runtime. While this approach removes the distribution burden from the developer's shoulders and is robust against the stragglers and failures, the overhead of the dynamic runtime could be a concern. To this end we compare the runtime overhead of Nimbus against the base MPI implementation. As we will see, execution templates reduce the runtime overhead of the controller significantly.

We ran two 3D PhysBAM water simulations, one with 512^3 (64-128GB of RAM) cells run on 8 workers, and one with 1024^3 cells (512GB-1TB of RAM) run on 64 workers. 512^3 is the common size run today because it can fit in memory on high-end simulation nodes, 1024^3 is the same size as the largest PhysBAM simulation that has ever been run. While the majority of execution time is spent in tasks that take

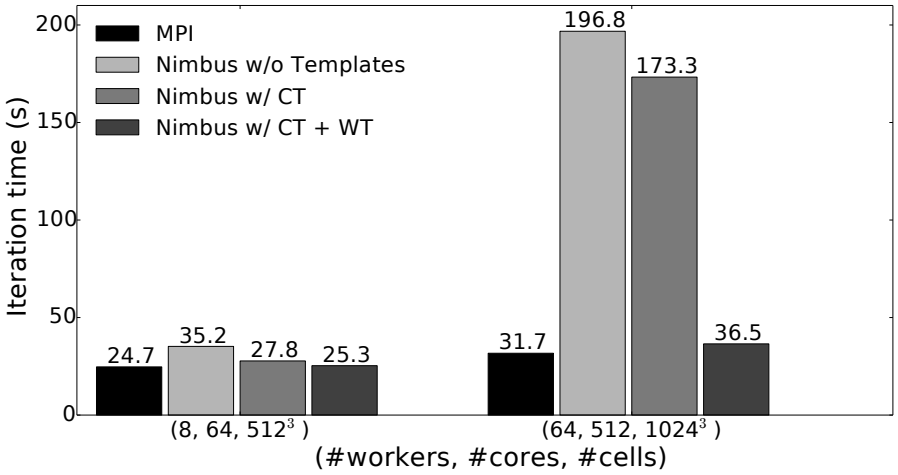


Figure 7.9: Iteration time of PhysBAM water simulation using MPI, and Nimbus in three scenarios: 1) without any template, 2) with only controller templates, and 3) with both controller and worker templates. Templates help Nimbus scale; without them Nimbus controller shows excessive overhead of 40–520% when running on 64–512 cores. The MPI performance is shows for comparison, as well.

60-70ms, the median task length is 13ms, 10% of tasks are <3ms and some tasks are as short as 100μs.

Figure 7.8 shows a detailed breakdown of the MPI and Nimbus implementation execution times. Their compute and data exchange (network) times are equivalent: the slight variations are due to threading differences in the two runtimes. Compared to the MPI implementation, all of Nimbus' overhead comes directly from having a centralized scheduler. Because the MPI nodes run in lock step with one another, their current positions in the program encode the overall program control flow. In contrast, Nimbus' scheduler manages the control flow across workers. When an MPI worker fails, the application crashes and must be carefully restarted manually.

7.5.2 Benefit of Execution Templates

Without execution template, Nimbus cannot deliver the performance numbers close to MPI as the control plane becomes a bottleneck. Figure 7.9 shows the results of running Nimbus under three different setting: 1) without any templates, 2) with only

controller template, and 3) with both controller and worker template. For a 512^3 water simulation, a central controller without templates imposes a 40% slowdown. Using templates, the 512^3 simulation can run within 3% of the MPI implementation. For the larger simulation, the aggregate task rate is 8 times higher and so overwhelms the scheduler; without templates, it take 520% longer than MPI. With templates, it runs within 15% of the MPI implementation.

7.5.3 Nimbus in Practice

For running the water simulation from PhysBAM library we had to implement the adapters as explained in Section 7.4.1 in less than 1,500 semicolons of C++ code. Note that this is a fairly complex particle-levelset simulation, including three main data types for face arrays, scalar arrays, and particles. This is a one time cost, and other simulations (e.g. smoke) could reuse the adaptors.

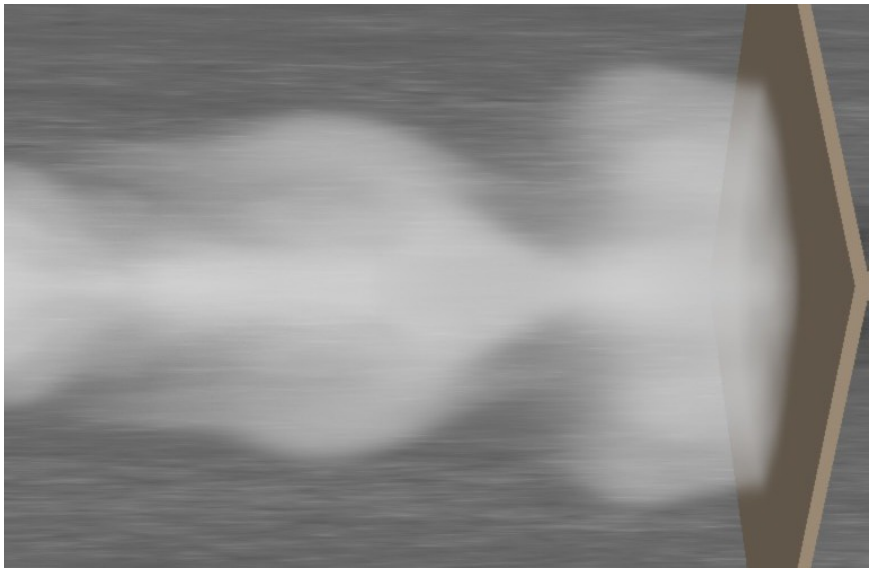
To monitor the code development for Nimbus in practice, we had one of the undergraduate summer interns import a smoke simulation from PhysBAM library into Nimbus. The adaptor code was directly reusable from the water simulation. Writing the driver program took less than a month for the student, who had no prior knowledge of graphics or simulation methods. In fact, the majority of the time was spent in understanding the original code and determining the required tasks. We expect that for developers with complete understanding of the methods, porting the tasks in to the Nimbus's API would be fairly straight forward.

Figure 7.10 shows the results of running the smoke simulation with and without Nimbus for 128^3-cell and 256^3-cell simulations. Nimbus enables bigger simulations with finer details in a reasonable time, while without Nimbus it would take more than a day to simulate the 256^3-cell simulation without parallelism.

7.5.4 Load Balancing and Fault Tolerance

Nimbus's controller monitors the nodes and reacts to stragglers and failures in the cluster. Figure 7.11 shows two different scenarios for running a 256^3-cell water simulation over a cluster of 8 nodes. In the first scenario, the load balancing and fault

(a) 128^3 cells, with Nimbus 28 minutes, without Nimbus: 94 minutes



(b) 256^3 cells, with Nimbus: 132 minutes, without Nimbus >30 hours

Figure 7.10: Smoke simulations with and without Nimbus. The top simulation has 128^3 cells, runs on a single-core and takes 94 minutes to simulate 70 frames. The bottom simulation uses Nimbus to automatically distribute this single-core simulation over 8 nodes (64 cores) in Amazon's EC2, simulating with greater detail: 70 frames of a 256^3 cell simulation takes 132 minutes. Without Nimbus the 256^3 cell simulation takes more than a day. Running the 128^3 simulation in Nimbus takes only 28 minutes.

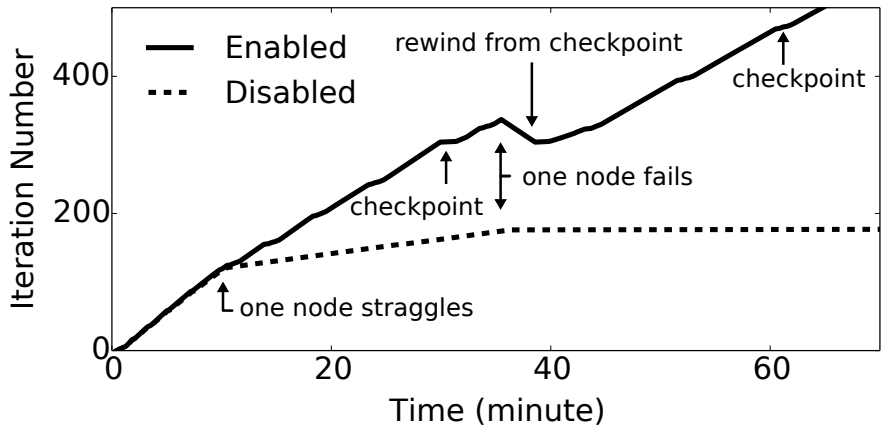


Figure 7.11: Running a 256^3-cell PhysBAM water simulation in a cluster of 8 Nimbus nodes in two cases: load balancing and fault tolerance enabled and disabled. Nimbus reacts to the straggling node by rebalancing the load and rewinds from latest checkpoint upon failure. Without these features the progress speed is bound by the speed of the straggler and any fault halts the simulation.

tolerance is enabled. Nimbus automatically creates checkpoints every 30 minutes and reacts to any load imbalance among the nodes by rebalancing the load dynamically. In the second scenario, both features are disabled; this mimics the more traditional approach in parallelizing such workloads (e.g. using MPI), where the control plane is statically compiled within the application logic. This removes any scheduling flexibility – resources are allocated once in the beginning of the simulation.

At the beginning of the simulation, all nodes are running at the full speed, until after 10 minutes when one of the nodes starts straggling. Straggling is a common and well-studied phenomenon in the cloud setting and it could happen for variety of reasons from data skew and networking congestion, to over-subscription of virtual machines [29] [130]. Here, we mimicked the straggler by launching CPU-bound background processes on the node to create contention for compute resources. With load balancing features of the Nimbus enabled, the controller rebalances the load to get around the straggling node, while in the other scenario the straggler limits the progress speed. Any solution that requires the static scheduling and partitioning decisions (e.g MPI implementations) would suffer from the same problem.

After about 35 minutes, one node fails. In this case, since the in-memory state is lost, the progress is halted. With automatic checkpointing enabled, Nimbus successfully rewinds back to a previous state and resumes computation with remaining available resources. Currently, the common approach in dealing with failures is human intervention to evaluate the situation and manually relaunch a simulation from a checkpoint with new parameters. Nimbus does this automatically and can also adapt the checkpoint creation rate dynamically. From our experience, failures such as memory corruptions in application and disk I/O failures are quite common for long running applications, and this is exacerbated in a distributed setting at higher scales.

Chapter 8

Conclusion and Discussion

This dissertation argues that the control plane of data analytics frameworks has become an emerging bottleneck. It also introduces a new abstraction, execution templates, for the control plane that supports orders of magnitude higher task throughputs while keeping the dynamic scheduling. This enables cloud computing frameworks to scale out for data analytics applications and also tackle traditional HPC applications. This section summarizes the dissertation's contributions, and provides a discussion on the limitations of execution templates and their implications. It concludes with our view on the design trends in cloud computing frameworks.

8.1 Contributions

In Chapter 1, this dissertation claimed

> Execution templates realize orders of magnitude higher task throughput than centralized frameworks, without sacrificing fine-grained, dynamic scheduling. By caching control plane decisions in parametrisable blocks, an execution template can dynamically schedule high performance computations. Execution templates are general enough to not only support traditional data analytics, but also complex applications with nested loops and data dependent branches that results in nondeterministic control flow.

This claim has been supported throughout this dissertation by demonstrating how control plane of cloud frameworks has become a bottleneck, introducing the execution template abstraction and their mechanisms for high task throughput and dynamic scheduling, describing the design and implementation of Nimbus framework that embeds execution templates, and evaluating execution template not only for traditional data analytics benchmarks, but also for graphical simulations ported into a cloud framework for the first time.

8.1.1 Cloud Control Plane Analysis

Chapter 3 went through in great details why traditional data analytics could run orders of magnitude faster if implemented in lower level languages. Shorter tasks implies higher task throughput at the control plane. Current frameworks either support high task throughput or allow dynamic, flexible scheduling, but not both.

8.1.2 Execution Templates

Execution templates enable a framework to provide high task throughput and dynamic scheduling simultaneously. By caching task graphs on the controller and workers, execution templates are able to schedule half a million tasks per second. At the same time, controllers can cheaply edit templates in response to scheduling changes or dynamic control flow.

8.1.3 Nimbus

Chapter 5 laid out the design and implementation of Nimbus, an analytics frameworks that supports high performance computations. Nimbus design is motivated by execution template requirements such that embedding execution templates in Nimbus requires almost no changes in the driver program. Also, Nimbus provides novel optimization for execution templates deployment to benefit the most from their instantiation and patching mechanisms. Chapter 6 evaluates the performance and flexibility of execution templates under Nimbus, and show Nimbus can reach the performance of

frameworks with distributed frameworks, while keeping reactive, fine-grained scheduling of frameworks with centralized control plane.

8.1.4 Graphical Simulations in Nimbus

Execution templates are not bound by simple data flows or static program flow control. To show the generality of execution templates, Chapter 7 introduced the novel data abstraction of Nimbus that enables automatic distribution of graphical simulations in the cloud. Simulation workloads are traditionally considered only in the HPC domain and it was for the first time that they were evaluated in a cloud frameworks. The evaluations show Nimbus with execution template matches the performance of hand-tuned MPI implementation within 15%, while providing automatic load balancing and fault tolerance that is normally left to be dealt with by application developers manually under MPI. Execution templates allow Nimbus controller to sustain the high tasks throughput requirements of graphical simulations. Without execution templates, Nimbus would run 6x slower, as the controller becomes a bottleneck.

8.2 Discussion

Execution templates described in this dissertation introduce a strong abstraction for the control plane. However, they have limitations, as well as certain requirements when incorporated in available systems.

8.2.1 Execution Templates in Other Frameworks

Execution templates are a general control plane abstraction. However, the requirements listed in Section 4.1.1 are simpler to incorporate in some systems than others. We design Nimbus such that execution templates are readily deployable. Embedding execution templates in other frameworks requires more extensive changes.

Incorporating execution templates into Spark [128] would require three significant changes to its data model and execution model, particularly its lazy evaluation and scheduling. First, it would need to support mutable data objects. When data is

immutable, each execution of a template is on new data object identifiers. Second, the Spark controller needs to be able to proactively push updates to each worker's block manager. In other words, workers need to be able to exchange data directly. Otherwise, every access of a new data object requires a lookup at the controller. Third, in Spark the controller is completely responsible for ensuring tasks run in the correct order, and so tasks sent to workers do not contain any dependency information. Adding execution templates would require adding this metadata to tasks as well as worker control logic. Spark workers need to queue their tasks and ordering metadata. While these changes are all quite tractable, together they involve a significant change to Spark's core execution model.

Naiad's [94] dataflow graphs can be thought of as an extreme case of execution templates, in which the flow graph describes a very large, long-running basic block. In other words, the entire tasks graph is a single basic block. Allowing a driver to store multiple dataflow graphs, edit them in place at the task granularity, and dynamically triggering them (with validation) would bring the dynamic scheduling benefits to Naiad.

Same argument holds for TensorFlow [22], although the central master in TensorFlow makes it easier to implement the execution templates mechanisms upon the available code base. Additionally, breaking up the driver program at the boundary of basic block would allow TensorFlow to keep the data dependent branches implemented through its multiplexer interfaces.

8.2.2 Limitations of Execution Templates

Execution templates assume that a driver program repeats the same basic blocks many times. For this reason, they are not valuable for short-lived jobs. Streaming computations, however, can benefit from standing templates that a driver periodically triggers. Execution templates also assume that a job's schedule, while dynamic, is mostly stable: Figure 6.8 showed that switching between cached schedules induces significant validation costs. If each iteration requires large edits or new templates, caching control decisions will not help. This erratic behavior is rare, however, because

such erratic schedules require large inter-worker data transfers.

8.2.3 Going Forward

A controller template has no understanding of the driver program's control flow. Since validation is centralized and unparallelizable, its cost in extremely dynamic control flows can be significant. Static analysis of the driver program, however, could provide better hints, similarly to how compilers allocate and manage registers across many control paths. The boundaries between the driver, controller, and workers, as well as the relationship between program analysis and runtime algorithms, remain open and interesting research problems.

8.3 Cloud Computing Design Projection

High performance computing has a long history of handling millions of tasks per second by completely distributing the control flow in the application. While cloud systems such as Naiad [94] and TensorFlow [22] have moved in this direction, it is worth noting that HPC systems have moved in the opposite direction due to the resulting complexity. When jobs run on tens of thousands of cores in heterogeneous architectures (CPUs, GPUs, GPGPUs, NUMA), reasoning about their performance and where execution time goes can be very difficult. As a result, modern HPC frameworks such as Legion [32] and Charm++ [77] have moved towards a more centralized model. Both communities have shifted to use techniques much closer to where the other started; this overshoot suggests that both may retreat to middle ground. Execution templates provide a valuable compromise between these two regimes by keeping the driver program as a centralized, sequential point of execution but allowing workers to schedule at the granularity of basic blocks.

Bibliography

[1] Apache Spark as a Compiler: Joining a Billion Rows per Second on a Laptop Deep dive into the new Tungsten execution engine. https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html.

[2] Apache Spark: Preparing for the Next Wave of Reactive Big Data . https://www.lightbend.com/blog/apache-spark-preparing-for-the-next-wave-of-reactive-big-data.

[3] Amazon EC2. http://aws.amazon.com/ec2.

[4] Amazon Machine Learning. https://aws.amazon.com/machine-learning/.

[5] Apache Flink. http://flink.apache.org.

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

[7] Apache Hive. https://hive.apache.org.

[8] Apache Mahout. http://mahout.apache.org/.

[9] Apache Spark MLLib. https://spark.apache.org/mllib/.

[10] Apache Spark, Preparing for the Next Wave of Reactive Big Data. http://goo.gl/FqEh94.

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

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

[13] Google Cloud Platform. https://cloud.google.com.

[14] Google Servers - The Old, The Small and The Container . http://www.websitepulse.com/blog/google-servers-the-old-the-small-and-the-container.

[15] Introducing Titan, Advancing the Era of Accelerated Computing. https://www.olcf.ornl.gov/titan/.

[16] Metis Graph Partitioning. http://glaros.dtc.umn.edu/gkhome/metis/metis/overview.

[17] Microsoft Azure. https://azure.microsoft.com.

[18] Microsoft Azure Machine Learning. https://azure.microsoft.com/en-us/services/machine-learning/.

[19] Oryx 2. http://oryx.io/.

[20] Oscar Aci-Tech Awards. http://www.oscars.org/sci-tech.

[21] Wikipedia Dumps. https://dumps.wikimedia.org/enwiki/.

[22] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael 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.

[23] Sameer Agrawal, Davies Liu, and Reynold Xin. Apache spark as a compiler: Joining a billion rows per second on a laptop. https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html.

[24] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.

[25] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *NSDI*, volume 10, pages 19–19, 2010.

[26] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective straggler mitigation: Attack of the clones. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 185–198, 2013.

[27] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective straggler mitigation: Attack of the clones. In *NSDI*, volume 13, pages 185–198, 2013.

[28] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Wang, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. Pacman: Coordinated memory caching for parallel jobs. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 20–20. USENIX Association, 2012.

[29] Ganesh Ananthanarayanan, Srikanth Kandula, Albert G Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in map-reduce clusters using mantri. In *OSDI*, volume 10, page 24, 2010.

[30] Reid Andersen and Kevin J Lang. Communities from seed sets. In *Proceedings of the 15th international conference on World Wide Web*, pages 223–232. ACM, 2006.

[31] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali 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.

[32] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex 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.

[33] Doug Beaver, Sanjeev Kumar, Harry C Li, Jason Sobel, Peter Vajgel, et al. Finding a needle in haystack: Facebook's photo storage. In *OSDI*, volume 10, pages 1–8, 2010.

[34] Gilbert Louis Bernstein, Chinmayee Shah, Crystal Lemire, Zachary Devito, Matthew Fisher, Philip Levis, and Pat Hanrahan. Ebb: A dsl for physical simulation on cpus and gpus. *ACM Transactions on Graphics (TOG)*, 35(2):21, 2016.

[35] Guy E Blelloch. Scans as primitive parallel operations. *IEEE Transactions on computers*, 38(11):1526–1538, 1989.

[36] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong 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.

[37] Thorsten Brants, Ashok C Popat, Peng Xu, Franz J Och, and Jeffrey Dean. Large language models in machine translation. In *In Proceedings of the Joint*

Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning. Citeseer, 2007.

[38] Robert Bridson. Legion: Expressing locality and independence with logical regions. In *Fluid Simulation for Computer Graphics*, 2008.

[39] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, 30(1):107–117, 1998.

[40] Kevin J Brown, HyoukJoong Lee, Tiark Rompf, Arvind K Sujeeth, Christopher De Sa, Christopher Aberger, and Kunle 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.

[41] Bradford L Chamberlain, David Callahan, and Hans P Zima. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[42] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R Henry, Robert Bradshaw, and Nathan Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. In *ACM Sigplan Notices*, volume 45, pages 363–375. ACM, 2010.

[43] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Acm Sigplan Notices*, volume 40, pages 519–538. ACM, 2005.

[44] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I Jordan, and Ion Stoica. Managing data transfers in computer clusters with orchestra. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 98–109. ACM, 2011.

[45] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[46] Jeff Dean. Building software systems at google and lessons learned, slide 99. http://goo.gl/oK7xGE.

[47] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

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

[49] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. Hawk: hybrid datacenter scheduling. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 499–510, 2015.

[50] Christina Delimitrou, Daniel Sanchez, and Christos 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.

[51] Pradeep Dubey, Pat Hanrahan, Ronald Fedkiw, Michael Lentine, and Craig Schroeder. Physbam: Physically based simulation. In *ACM SIGGRAPH 2011 Courses*, SIGGRAPH '11, pages 10:1–10:22, New York, NY, USA, 2011. ACM.

[52] Douglas Enright, Ronald Fedkiw, Joel Ferziger, and Ian Mitchell. A hybrid particle level set method for improved interface capturing. *Journal of Computational Physics*, 183(1):83–116, 2002.

[53] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, volume 1. Springer series in statistics Springer, 2009.

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

[55] J Davison de St Germain, John McCorquodale, Steven G Parker, and Christopher R Johnson. Uintah: A massively parallel problem solving environment. In *High-Performance Distributed Computing, 2000. Proceedings. The Ninth International Symposium on*, pages 33–41. IEEE, 2000.

[56] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM, 2003.

[57] Luca Gherardi, Davide Brugali, and Daniele Comotti. A java vs. c++ performance evaluation: a 3d modeling benchmark. In *Simulation, Modeling, and Programming for Autonomous Robots*, pages 161–172. Springer, 2012.

[58] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, volume 11, pages 24–24, 2011.

[59] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven 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.

[60] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, volume 12, page 2, 2012.

[61] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *Proceedings of OSDI*, pages 599–613, 2014.

[62] Sergei Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In *European Conference on Parallel Processing*, pages 401–408. Springer, 1996.

[63] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 455–466. ACM, 2014.

[64] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. Do the hard stuff first: Scheduling dependent computations in data-analytics clusters. *arXiv preprint arXiv:1604.07371*, 2016.

[65] Francis H Harlow, J Eddie Welch, et al. Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface. *Physics of fluids*, 8(12):2182, 1965.

[66] Donald Olding Hebb. *The organization of behavior: A neuropsychological theory*. Psychology Press, 2005.

[67] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[68] Chao Huang, Orion Lawlor, and Laxmikant V Kale. Adaptive mpi. In *Languages and Compilers for Parallel Computing*, pages 306–322. Springer, 2004.

[69] Po-Sen Huang, Haim Avron, Tara N Sainath, Vikas Sindhwani, and Bhuvana Ramabhadran. Kernel methods match deep neural networks on timit. In *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, pages 205–209. IEEE, 2014.

[70] Robert Hundt. Loop recognition in c++/java/go/scala. *Proceedings of Scala Days*, 2011:38, 2011.

[71] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8, page 9, 2010.

[72] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 59–72. ACM, 2007.

[73] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew 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.

[74] Glen Jeh and Jennifer Widom. Scaling personalized web search. In *Proceedings of the 12th international conference on World Wide Web*, pages 271–279. ACM, 2003.

[75] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Changhoon Kim, and Albert Greenberg. Eyeq: Practical network performance isolation at the edge. *REM*, 1005(A1):A2, 2013.

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

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

[78] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 485–497, 2015.

[79] Qifa Ke, Michael Isard, and Yuan 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.

[80] Richard E Ladner and Michael J Fischer. Parallel prefix computation. *Journal of the ACM (JACM)*, 27(4):831–838, 1980.

[81] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[82] Jure Leskovec and Julian J Mcauley. Learning to discover social circles in ego networks. In *Advances in neural information processing systems*, pages 539–547, 2012.

[83] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of massive datasets*. Cambridge University Press, 2014.

[84] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *OSDI*, volume 14, pages 583–598, 2014.

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

[86] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, 2014.

[87] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA., 1967.

[88] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[89] Christopher D Manning, Prabhakar Raghavan, Hinrich Schütze, et al. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

[90] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[91] Frank McSherry, Michael Isard, and Derek Gordon Murray. Scalability! but at what cost? In *HotOS*. Citeseer, 2015.

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

[93] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, et al. f4: Facebooks warm blob storage system. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, pages 383–398. USENIX Association, 2014.

[94] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013.

[95] Derek Gordon Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. Ciel: A universal execution engine for distributed data-flow computing. In *NSDI*, volume 11, pages 9–9, 2011.

[96] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[97] Kay Ousterhout, Aurojit Panda, Josh Rosen, Shivaram Venkataraman, Reynold Xin, Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. The case for tiny tasks in compute clusters. In *HotOS*, volume 13, pages 14–14, 2013.

[98] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, Byung-Gon Chun, and VMware 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.

[99] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 69–84. ACM, 2013.

[100] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[101] Shoumik Palkar, James Thomas, and Matei Zaharia. Nested vector language: Roofline performance for data parallel code. http://livinglab.mit.edu/wp-content/uploads/2016/01/nvl-poster.pdf.

[102] Shoumik Palkar, James J Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, Matei Zaharia, and Stanford InfoLab. Weld: A common runtime for high performance data analytics.

[103] David Patterson and John L Hennessy. *Computer architecture: a quantitative approach*. Elsevier, 2012.

[104] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with sawzall. *Scientific Programming*, 13(4):277–298, 2005.

[105] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. Faircloud: sharing the network in cloud

computing. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 187–198. ACM, 2012.

[106] Russell Power and Jinyang Li. Piccolo: Building fast, distributed programs with partitioned tables. In *OSDI*, volume 10, pages 1–14, 2010.

[107] Alexander Rasmussen, Michael Conley, George Porter, Rishi Kapoor, Amin Vahdat, et al. Themis: an i/o-efficient mapreduce. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 13. ACM, 2012.

[108] Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

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

[110] Tom Seymour, Dean Frantsvog, and Satheesh Kumar. History of search engines. *International Journal of Management and Information Systems*, 15(4):47, 2011.

[111] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *IEEE Transactions on pattern analysis and machine intelligence*, 22(8):888–905, 2000.

[112] Elliott Slaughter, Wonchan Lee, Sean Treichler, Michael Bauer, and Alex Aiken. Regent: A high-productivity programming language for hpc with logical regions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 81. ACM, 2015.

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

[114] Guy L Steele Jr. Rabbit: A compiler for scheme. 1978.

[115] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156. ACM, 2014.

[116] Jakob Uszkoreit, Jay M Ponte, Ashok C Popat, and Moshe Dubiner. Large scale parallel document mining for machine translation. In *Proceedings of the 23rd International Conference on Computational Linguistics*, pages 1101–1109. Association for Computational Linguistics, 2010.

[117] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[118] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Ali Ghodsi, Michael J Franklin, Benjamin Recht, and Ion Stoica. Drizzle: Fast and adaptable stream processing at scale.

[119] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion 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.

[120] Paul Werbos. Beyond regression: New tools for prediction and analysis in the behavioral sciences. 1974.

[121] Reynold 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.

[122] Reynold Xin and Josh 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.

[123] Reynold S Xin, Josh Rosen, Matei Zaharia, Michael J Franklin, Scott Shenker, and Ion Stoica. Shark: Sql and rich analytics at scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of data*, pages 13–24. ACM, 2013.

[124] Jaewon Yang and Jure Leskovec. Overlapping community detection at scale: a nonnegative matrix factorization approach. In *Proceedings of the sixth ACM international conference on Web search and data mining*, pages 587–596. ACM, 2013.

[125] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, volume 8, pages 1–14, 2008.

[126] Matei Zaharia. Apache spark 2.0: Keynote talk at spark summit 2016, san francisco. https://spark-summit.org/2016/events/keynote-1-day-2/.

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

[128] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[129] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 423–438. ACM, 2013.

[130] Matei Zaharia, Andy Konwinski, Anthony D Joseph, Randy H Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *OSDI*, volume 8, page 7, 2008.

Omid Mashayekhi

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Philip Levis) Principal Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Mendel Rosenblum)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Alex Aiken)

Approved for the University Committee on Graduate Studies
