OPTIMIZING RESPONSE TIME FOR DISTRIBUTED APPLICATIONS IN PUBLIC CLOUDS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Tao Zou

January 2015

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

OPTIMIZING RESPONSE TIME FOR DISTRIBUTED APPLICATIONS IN PUBLIC CLOUDS

Tao Zou, Ph.D.

Cornell University 2015

An increasing number of distributed data-driven applications are moving into public clouds. By sharing resources and operating at large scale, public clouds promise higher utilization and lower costs than private clusters. Also, flexible resource allocation and billing methods offered by public clouds enable tenants to control response time or time-to-solution of their applications.

To achieve high utilization, however, cloud providers inevitably place virtual machine instances non-contiguously, i.e., instances of a given application may end up in physically distant machines in the cloud. This allocation strategy leads to significant heterogeneity in average network latency between instances. Also, virtualization and the shared use of network resources between tenants results in network latency jitter. We observe that network latency heterogeneity and jitter in the cloud can greatly increase the time required for communication in these distributed data-driven applications, which leads to significantly worse response time.

To improve response time under latency jitter, we propose a general parallel framework which exposes a high-level, data-centric programming model. We design a jitter-tolerant runtime that exploits this programming model to absorb latency spikes transparently by (1) carefully scheduling computation and (2) replicating data and computation. To improve response time with heterogeneous mean latency, we present ClouDiA, a general deployment advisor that

selects application node deployments minimizing either (1) the largest latency between application nodes, or (2) the longest critical path among all application nodes.

We also describe how to effectively control response time for interactive data analytics in public clouds. We introduce Smart, the first elastic cloud resource manager for in-memory interactive data analytics. Smart enables control of the speed of queries by letting users specify the number of compute units per GB of data processed, and quickly reacts to speed changes by adjusting the amount of resources allocated to the user. We then describe SmartShare, an extension of Smart that can serve multiple data scientists simultaneously to obtain additional cost savings without sacrificing query performance guarantees. Taking advantage of the workload characteristics of interactive data analysis, such as think time and overlap between datasets, we are able to further improve resource utilization and reduce cost.

BIOGRAPHICAL SKETCH

Tao Zou was born in Changzhou, China, a modern city with more than 3000 years of history. Tao had been actively involved in math competition before middle school. He then changed his focus to programming contests mostly due to his interests in computer hardware and computer games. Tao was first introduced to computer science research by Prof Shuigeng Zhou, during his undergraduate study in Fudan University, Shanghai, China.

Since 2009, Tao Zou has been a Ph.D. student in the Department of Computer Science at Cornell University. His research focuses on building general optimization frameworks for data intensive applications in public clouds.

To my parents and Qingxi.

ACKNOWLEDGEMENTS

I am deeply grateful to my advisor, Johannes Gehrke, for his visionary guidance and active support during my graduate study at Cornell University. He led me into the world of database and system research, and kept encouraging me to tackle big problems using elegant solutions. In addition to enlightening me through numerous technical discussions and meetings, Johannes also patiently taught me how to find ideas, write papers and give presentations. I have also learned a lot from his style of communication and collaboration, as well as his attitude towards work-life balance.

I am especially fortunate to have had the opportunity to work with Alan Demers in all my research projects at Cornell. He spent countless hours with me in each project, helping me to accurately define the problem, nail down all implementation details, and finally deeply understand experimental results. It has been an invaluable experience for me to learn from Al how to build large-scale systems both in theory and in practice. In addition, I would like to thank the other members of my thesis committee, Dexter Kozen and David Bindel for their inspiring comments and advice.

I would like to give very special thanks to Marcos Vaz Salles. Marcos was my mentor when I was new to database research. Through our collaborations in research projects and in teaching the undergraduate database class, I started to taste the joy of research and teaching. I am also greatly influenced by his elegance as a researcher and as a friend.

I am also grateful to be part of the Cornell Database Group. In particular, I would like to thank Guozhang Wang and Walker White for their generous help and insightful suggestions in our research projects. I would also like to thank Lucja Kot, Michaela Gotz, Ben Sowell, Tuan Cao, Wenlei Xie and Bailu Ding

with whom I had many enlightening discussions.

I also would like to acknowledge my internship mentors: Vivek Narasayya at Microsoft Research Redmond, Mahesh Balakrishnan and Ted Wobber at Microsoft Research Silicon Valley, as well as Fatma Ozcan and Yuanyuan Tian at IBM Almaden Research Center. They expanded my horizons by introducing me to important real-world challenges, encouraging me to break out of my research comfort zone, and giving me enormous help to finish each internship project in a limited amount of time.

Finally, I would like to thank my parents for their unconditional love and support, without which I could not have come this far. I am also extremely lucky to have my girlfriend, Qingxi Li, who magically filled my graduate student life with full happiness and countless memorable moments.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Background

With advances in data center and virtualization technology, more and more distributed data-driven applications, such as High Performance Computing (HPC) applications [50, 78, 115, 167], web services and portals [58, 106, 116, 140], data analytics [160, 151] and even search engines [10, 13], are moving into public clouds [8].

Representative public cloud service providers include Amazon Elastic Compute Cloud (EC2) [4], Google Compute Engine [62], Windows Azure [148], and Rackspace Cloud Server [113]. Tenants allocate and then remotely access *instances* from public clouds. Each instance is a virtual machine allocated on a physical machine managed by public clouds. Multiple instances, possibly from different tenants, can share the same physical machine.

To address divergent application needs, public clouds offer a wide selection of instance types at different costs. Different instance types provide different combinations of CPU, memory, networking capacity, and other resources. A tenant can freely allocate and terminate instances of specified types at any time. For each instance, the tenant pays only for time actually used.

Public clouds represent a valuable platform for tenants due to their incremental scalability and reliability. Nevertheless, the most fundamental advantage of using public clouds is cost-effectiveness. Public clouds take advantage of the economies of scale in managing resources [108], and at the same time

obtain higher utilization by combining resource usage from multiple tenants, leading to cost reductions unattainable by dedicated private clusters.

In a distributed application, components located on different machines communicate and coordinate through a network. Communication latency can greatly affect the performance of a wide class of distributed applications, such as scientific simulations [143], key-value stores [46], search engines [10, 13] and web services and portals [58, 106]. These applications, when deployed in private clusters, make a critical assumption of the existence of stable, homogeneous, low-latency network connections among physical machines. However, such network connections no longer exist in public clouds due to sharing among a large number of tenants, which makes efficiently deploying these applications in the cloud a challenging task. In Section 1.1.1, we discuss the characteristics of network latency in public clouds and its effect on response time of latency-sensitive applications. This motivates our work on improving response time for latency-sensitive applications in public clouds, which is presented in Chapters 2 and 3.

Data analytics, another important use case for pubic cloud, is an iterative process of querying, observing results, and formulating new queries [69]. Recent advances in distributed in-memory data analytics engines, such as Shark [151], DB2 BLU [118] and SAP HANA [52], enable data scientists to run data analytics over large datasets interactively, with fast response time. Although they can be deployed directly in public clouds, these engines are designed to use in private clusters where a fixed amount of resources is given a priori. This means they cannot allocate additional resources when the fixed amount of resources is inadequate to maintaining query response time, nor can

they release resources when they are not needed. In Section 1.1.2, we discuss the workload characteristics of interactive data analytics, and argue that the flexibility in instance allocation and termination that the cloud offers is a good match for interactive data analytics. This motivates our work in building a cloud-aware elastic resource manager that exploits such flexibility to control response time for interactive data analytics at real time, which is presented in Chapter 4.

1.1.1 Network Latency in Public Clouds

Latency Mean

Giving tenants freedom in allocating and terminating instances at any time, public clouds face new challenges in choosing the placement of instances on physical machines. First, they must solve this problem at large scale, and at the same time take into account different tenant needs regarding latency, bandwidth, or reliability. Second, even if a fixed goal such as minimizing latency is given, the placement strategy still needs to take into consideration the possibility of future instance allocations and terminations.

Given these difficulties, public cloud service providers do not currently expose instance placement or network topology information to cloud tenants.[1] While these API restrictions ease application deployment, they may cost significantly in performance, especially for latency-sensitive applications. Unlike bandwidth, which can be quantified in a Service-Level Agreement (SLA) as a

[1]The only exception we know of is the notion of *cluster placement groups* in Amazon EC2 cluster instances. However, these cluster instances are much more costly than other types of instances, and only a limited number of instances can be allocated within a cluster placement group.

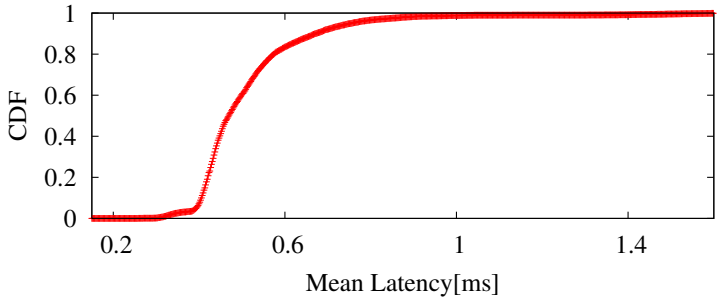


Figure 1.1: Latency Heterogeneity in EC2

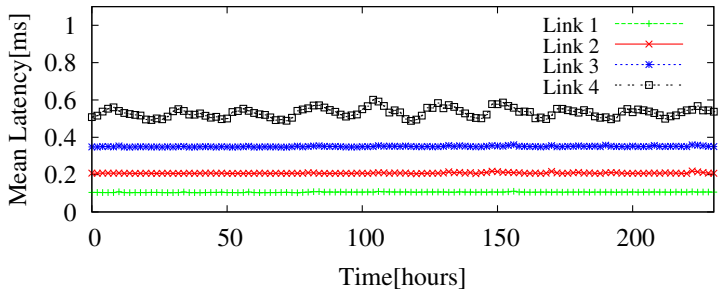


Figure 1.2: Mean Latency Stability in EC2

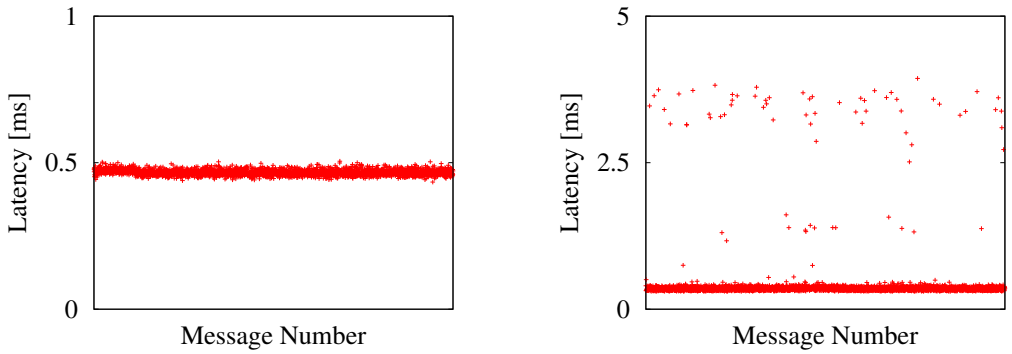
single number, network latency depends on message sizes and the communication pattern, both of which vary from one application to another.

In the absence of placement constraints specified by cloud tenants, cloud providers are free to assign instances to physical resources non-contiguously; i.e., instances allocated to a given application may end up in physically distant machines. This leads to heterogeneous network connectivity between instances: Some pairs of instances are better connected than other pairs in terms of latency, loss rate, or bandwidth. Figure 1.1 illustrates this effect. We present the cumulative distribution function (CDF) of the mean pairwise end-to-end latencies among 100 Amazon EC2 large instances (m1.large) in the US East Region, ob-

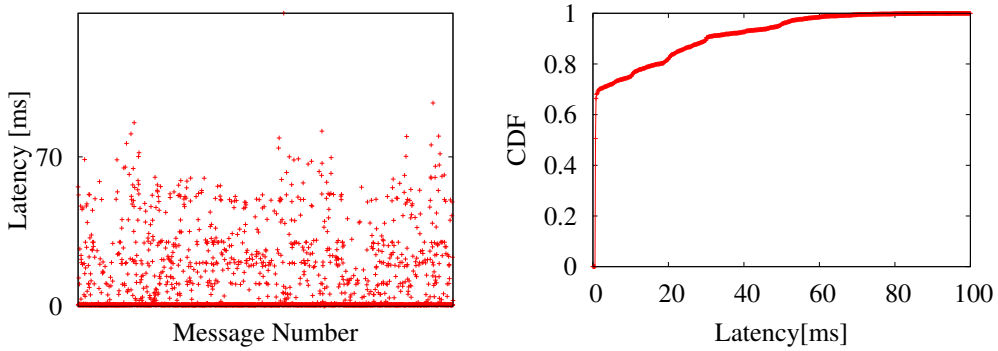
tained by TCP round-trip times of 1 KB messages. Around 10% of the instance pairs exhibit latency above 0.7 ms, while the bottom 10% are below 0.4 ms. This heterogeneity in network latencies can greatly increase the response time of distributed, latency-sensitive applications if instance pairs with high latency are in the critical path for the application to make progress. Figure 1.2 plots the mean latencies of four representative links over a 10-day experiment, with latency measurements averaged every two hours. The observed stability of mean latencies suggests that applications may obtain better performance over a significant period of time by selecting "good" links for communication.

Latency Jitter

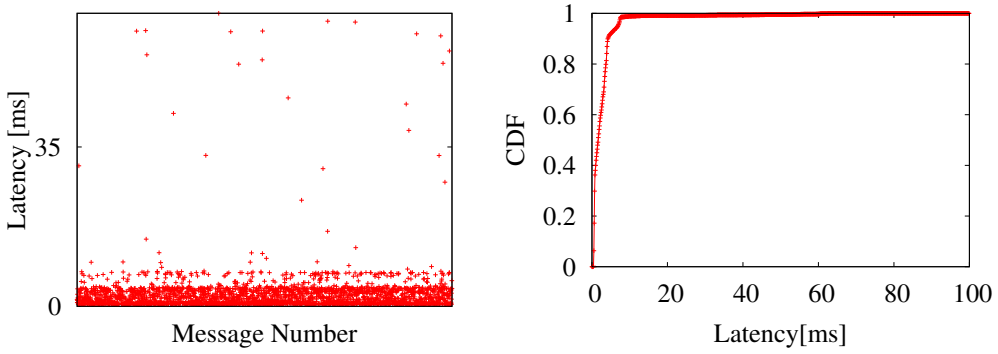
Besides heterogeneous mean latency within a group of instances in public clouds, recent experimental studies have demonstrated that each pair of cloud instances can suffer from high variance in network latency over time [126, 142]. We call such high latency variance over time *latency jitter*. We have confirmed the existence of latency jitter in public clouds by measuring the TCP round-trip times for 16 KB messages in several environments, as shown in Figure 1.3. These environments include the Cornell Weblab, which is a modest dedicated cluster of machines interconnected by Gigabit Ethernet, and Amazon EC2 cloud instances in the 32-bit "Small", 64-bit "Large" and 64-bit "Cluster Compute" categories. Communication in the Weblab Cluster is well-behaved, with latencies tightly distributed around the mean. The 32-bit EC2 instances have poor performance, with high average latency and high variance. The 64-bit EC2 instance categories show better average latency, but suffer frequent latency "spikes" more than an order of magnitude above the mean. Even the cluster



(a) Weblab Instances (b) EC2 Cluster Instances



(c) EC2 Small Instances



(d) EC2 Large Instances

Figure 1.3: Latency in Cornell Weblab Cluster and in EC2

compute instances, advertised for HPC applications, show the same effect. For 32-bit "Small" and 64-bit "Large" instance where latency jitter is more severe, we present the CDF of latency in correspondence to the latency measurements

over time. Between two "Small" instances, about 25% of the messages take more than 10 ms to finish the round-trip, whereas the mean latency is 8.6 ms. Between two "Large" instances, about 8% of the messages take more than 5 ms to finish the round-trip, whereas the mean latency is 2.6 ms.

Distributed applications that run frequent synchronization between instances suffer dramatically in the presence of latency jitter. For example, in a distributed application that uses bulk synchronous model [138], barrier synchronization is needed to make progress after each independent parallel computation phase. Suppose a message from process P_i to process P_j is delayed by a latency spike during barrier synchronization. P_j then blocks and cannot make progress until it gets unblocked by the arriving message. If computational load is balanced, this will cause P_j to be late sending its own messages for the next barrier synchronization. This in turn will create a *latency wave* eventually affecting all the processes, which is hard to compensate for in these applications.

1.1.2 Interactive Data Analytics in Public Clouds

As mentioned before, data analytics is another important use case for public clouds. The interactive usage model adopted by such systems implies that their generated workload changes significantly over time. It is therefore beneficial to manage compute resources elastically, allocating resources when users are active and releasing them during periods of inactivity.

Taking advantage of the flexible instance allocation provided by the cloud, we can maintain only datasets that are needed by currently active users in the aggregated memory of machines in the cloud. But memory alone is not suffi-

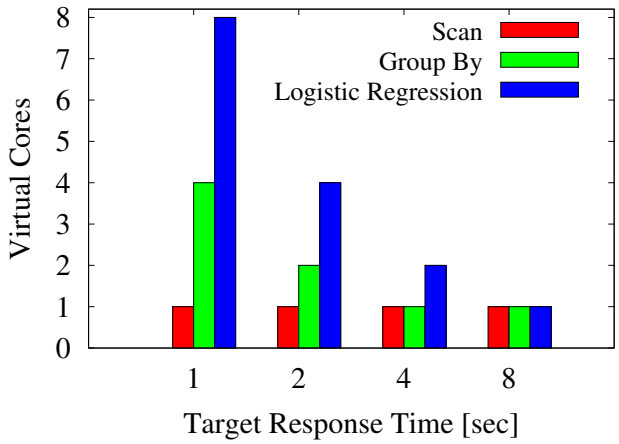


Figure 1.4: Virtual Cores Needed for Desired Interactivity

cient to guarantee interactivity of queries; compute power must be considered as well. Figure 1.4 illustrates this requirement, showing the number of virtual cores required to achieve specified levels of interactivity for several common data science operations: scan, group by, and one iteration of logistic regression applied to a 3 GB in-memory dataset on a Amazon EC2 compute optimized extra large (c1.xlarge) instance [4]. Clearly, some operators are more computationally heavy than others. Similar observations have been made for Spark [151] and for Hadoop [60]. Thus, operator cost should be taken into consideration for elastic resource management. Ideally, users should be able to control the response time of interactive data analysis *by tuning the amount of CPU resources associated with their in-memory datasets*. The diversity in the mix of resources that can be allocated in the cloud enables us to closely fit the changing needs of interactive users. By choosing the best suitable instance type initially and adjusting the instance type selection dynamically based on a user's desired interactivity, significant cost saving can also be potentially achieved.

1.2 Dissertation Outline

This remainder of this dissertation is organized as follows. In Chapters 2 and 3, we discuss how to improve response time in public clouds, under latency jitter and with heterogeneous mean latency respectively. Then we present how to control response time for interactive data analytics in the cloud in Chapter 4. Finally, we discuss related work in Chapter 5.

CHAPTER 2

IMPROVING RESPONSE TIME UNDER LATENCY JITTER

2.1 Introduction

Many important scientific applications are organized into logical time steps called *ticks*. Examples of such time-stepped applications include behavioral simulations, graph processing, belief propagation, random walks, and neighborhood propagation [7, 25, 34, 39, 92]. They also include classic iterative methods for solving linear systems and eigenvalue problems [16]. These applications are typically highly data parallel within ticks; however, the end of every tick is a logical barrier. Today these applications are usually implemented in the *bulk synchronous* model, which advocates global synchronization as a primitive to implement tick barriers [138].

The bulk synchronous model has allowed scientists to easily design and execute their parallel applications in modern HPC centers and large private clusters. However, the use of frequent barriers makes these codes very sensitive to fluctuations in performance. As a consequence, most modern HPC centers allocate whole portions of a cluster exclusively for execution of an application. This model works well for heavy science users, but is not ideal for mid-range applications that only need to use a few hundred compute nodes [115]. In particular, these mid-range users have to wait on execution queues for long periods, sometimes hours or even days, to get to run their jobs. This significantly lengthens the time-to-solution for a number of scientific research groups worldwide.

This chapter examines what happens when we take these scientific applica-

tions off those private, well-behaved, expensive computing platforms and run them in the cloud. As the next generation computing platform, the cloud holds both promise and challenges for large-scale scientific applications [50, 115]. On the one hand, the cloud offers scientists instant availability of large computational power at an affordable price. This is achieved via low-overhead virtualization of hardware resources [153, 154, 155]. On the other hand, the common practice of using commodity interconnects and shared resources in the cloud alters fundamental assumptions that scientific applications were based on in the past.

As we have discussed in Section 1.1.1, time-stepped applications that are programmed in the bulk synchronous model suffer dramatically from the high network latency jitter in the today's cloud. The HPC community has invested significant work in optimizing communication for time-stepped applications [2, 23, 83]. However, these optimization techniques were developed using a model of fixed, unavoidable latency for sending a message across a dedicated network, and not for the unstable, unpredictable latency that characterizes the cloud. Furthermore, many of these previous techniques can only be applied to applications whose computational logic can be formulated as a sparse linear algebra problem. [47]

This specialization significantly impairs the productivity of scientists who want to develop new applications without regard for which optimizations to use for communication. A general programming model for time-stepped applications that can abstract the messy latency characteristics of the cloud is currently missing.

Contributions of this Chapter. In this chapter we describe a *general*, *jitter-*

tolerant parallel framework for time-stepped scientific applications. By taking a data-centric approach, we shield developers from having to implement communication logic for their applications. Our data-driven runtime automatically provides multiple generic optimizations that compensate for network jitter. In summary, this work makes the following contributions:

1. We observe that logical barriers in time-stepped applications usually encode data dependencies between subsets of the application state. Our programming model allows developers to abstract application state as *tables*, and express the data dependencies as functions over *queries* (Section 2.4).

2. We present an efficient jitter-tolerant runtime, by which time-stepped applications specified in our programming model are executed in parallel. Our implementation uses two primary techniques: *scheduling* based on data dependencies and *replication* of data and computation (Section 2.5). A formal description of our model and correctness proofs of our algorithms appear in Appendix A.1.

3. In an experimental evaluation, we show that our runtime significantly improves the performance of a wide range of scientific applications in Amazon EC2. We observe gains of up to a factor of three in throughput for several time-stepped applications coded in our programming model (Section 2.6).

We start our presentation by defining time-stepped applications (Section 2.2) and summarizing our approach (Section 2.3).

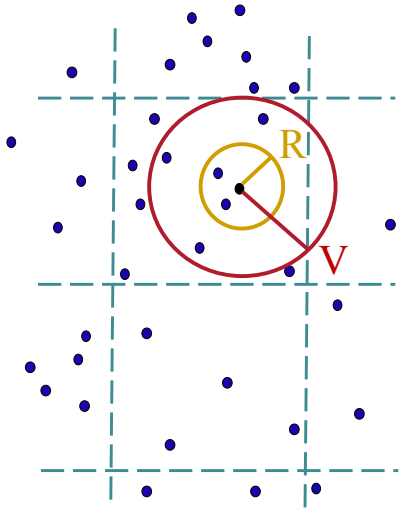
2.2 Time-Stepped Applications

A *time-stepped application* is a parallel scientific application organized into logical ticks. Processes in these applications proceed completely in parallel within a tick and exchange messages only at tick boundaries. Today, most of these applications are implemented in the bulk synchronous model, which introduces *logical* global barriers at the end of a tick [138]. The conceptual simplicity of this model has led to its widespread adoption by a large number of scientific applications [7, 16, 25, 34, 39, 92].

Time-stepped application developers typically follow proven design patterns to improve parallel efficiency. First, developers usually choose to exploit *data parallelism* within a tick, since it provides for very fine-grained parallel computations. Second, developers strive to architect their applications for high *locality* of access so that they can minimize the amount of information exchanged among processes at logical barriers. We illustrate these design patterns in the following example, which we use throughout this chapter.

Running Example: Behavioral Simulations. Behavioral simulations model complex systems of individual, intelligent agents, such as transportation networks and animal swarms [34, 39]. In these

simulations, time is discretized into ticks; within a tick, the agents concurrently gather data about the world, reason on this data, and update their states for the next tick [143]. For instance, Couzin et al. used this type of simulation to study information transfer in schools of fish [39]. An illustration of this simulation is shown on the right. Within a tick, each fish agent inspects the current velocities of other visible fish to determine its new velocity for the next tick. In addition, informed individuals balance these social interactions with a preferred direction (e.g., a food source) to determine movement. Two application parameters determine how far a fish can see or move within a tick. The former is termed visibility, denoted V, while the latter is termed reachability, denoted R.



Given visibility and reachability constraints, we can *partition* the simulated space and assign each spatial partition to a different process (dotted lines in the figure). The processing of a tick is data-parallel: each process executes the tick logic for each fish agent in its partition independently, calculating its new state. When all the fish agents in some partition have been updated, we say this partition has been *stepped* to the next tick. Notice, however, that processing of a fish requires access to the state of all neighbor fish within distance V as its *context*. Therefore, processing of a partition requires not only computing the new state for all fish within the partition, but also knowledge of which fish move between partitions. Such *dependencies* for a partition in space can be found by expanding the partition rectangle by both the visibility and reachability parameters. As these parameters typically represent a small fraction of the simulated space, it

is clear that the fish simulation exhibits strong locality.

Many other important applications, such as graph processing platforms [98] and iterative solvers [16], are time-stepped and therefore designed to exploit data parallelism and locality. We develop two additional examples of such applications in Appendixes A.2.1 and A.2.2. They correspond to an iterative method, Jacobi iteration [16], and a graph processing application, PageRank [25].

2.3 Our Approach

As we have mentioned before, due to their use of logical barriers bulk synchronous implementations of time-stepped applications are extremely vulnerable to latency. There has been significant work in the past to compensate for fixed latency in these applications [2, 23, 83]. However, applying these techniques to a new time-stepped application requires non-trivial redesign of the application's computational logic as well as its underlying communication logic. In addition, making these techniques work in the presence of large latency *variance* in the cloud remains a challenging task. We tackle both of these challenges simultaneously by providing a general parallel framework for scientists which exploits properties of time-stepped applications to hide all details of handling latency jitter. Our framework abstracts time-stepped applications into an intuitive *data-driven programming model* so that scientists only need to focus on the computational logic of their applications. The framework then executes the program in an associated *jitter-tolerant runtime* for efficient processing. By carefully modeling data dependencies and locality of the application in our program-

ming model, our jitter-tolerant runtime is able to schedule useful computation *automatically* and *efficiently* during latency spikes.

More specifically, our programming model abstracts the application state as a set of *relational tables*. Conceptually, each tick of the computation takes these tables from one version to the next. In order to capture data parallelism and locality, we let the application developers specify a partitioning function over these tables, as well as model the data dependencies necessary for correct computation.

Modeling data dependencies efficiently is not trivial. The naive approach would be to specify dependencies directly on the data, creating a large data dependency graph among individual tuples. Unfortunately, the overhead of tracking dependencies at such a fine granularity would be very large. Our programming model takes a different approach: We compactly represent sets of tuples by encoding them as *queries*. Data dependencies are then modeled by functions that define relationships between queries. This approach introduces the complexity of ensuring that dependency specifications on queries are equivalent to those on the underlying data. Once we formally prove the correctness of these relationships, however, we obtain a programming model that can naturally express locality and dependencies at very low overhead. All the complexity of managing dependency relationships on queries is hidden inside our runtime implementation.

As an example, consider the fish simulation above. The application state is a table containing each fish as a separate tuple. We model the partition assigned to each process as a query – encoded by a rectangle in the simulation space. As computation proceeds over several ticks, we automatically ensure that data

items are correctly updated and respect the partition query. To achieve this, we can apply a function to the partition query that returns another query corresponding to the partition's rectangle enlarged by how far fish can see. This query thus encodes the read dependencies of the partition. Similarly, we can apply another function to the latter query to obtain a rectangle further enlarged by how far fish can move. This third query encodes both the read and write dependencies of the partition. We describe our programming model in detail in Section 2.4. This programming model is not language-specific and we anticipate implementations in different languages will emerge.

Based on the dependencies abstracted as queries, the jitter-tolerant runtime controls all aspects of the data communication between processes on behalf of the application. The runtime ensures that the right data is available at the right time to unblock computation and overcome jitter. As we show in Section 2.5.1, the runtime takes advantage of the structure of dependency relationships to synchronize efficiently. First, our runtime restricts communication to only those processes that are dependency *neighbors*. This technique reduces the communication cost by replacing global synchronization by local synchronization. However, it neither removes nor relaxes synchronization points. To deal with variance in message latency during synchronization, our runtime further optimizes communication using two techniques: *dependency scheduling* and *computational replication*. The goal of dependency scheduling (Section 2.5.2) is to continue computation on subsets of the application state whose dependencies are locally satisfied when a latency spike occurs. In that case, we can advance computations to future steps on subsets of the state instead of getting blocked. Computational replication (Section 2.5.3) uses redundant data and its respective computation both to communicate less often *and* to unblock even more compu-

tation internal to a process. Hence, this technique can be used to complement dependency scheduling by providing additional flexibility at synchronization points.

2.4 Programming Model

In this section, we describe the programming model offered by our jitter-tolerant runtime. Table 2.1 summarizes the functions we require application developers to instantiate. We explain them in detail in the following subsections.

2.4.1 Modeling State and Computation

Global State. A time-stepped program logically has a *global state* that is updated as part of some iterative computation. We model this global state as a set of relational tables. Each tuple in a table is uniquely identified, and may contain a number of attributes. For example, the global state of the fish simulation introduced in Section 2.1 can be represented by the table:

Fish(<u>id</u>, x, y, vx, vy).

Here, *id* is a unique identifier for a fish. The attributes (*x*,*y*) and (*vx*,*vy*) represent a fish's position and velocity, respectively. For simplicity of presentation, we assume that the global state consists of a single table in first normal form, i.e., cells are single-valued [97]. Our techniques can be extended to multiple tables and structured attributes.

Table 2.1: Programming Model

$\mathbb{S}^t$ stands for any possible global state S in execution at time step t.

List<Query> PART(int n) Partitions the global state so that it can be distributed to n processes. The partitioning is represented by a list of n queries that select subsets of the global state which should be given to each process.	**State NEW(Query q)** Initializes the local state according to q. Typical implementations of this function read the local state selected by q from a distributed file system.
State STEP(State toStep, State context) Steps the application logic for every tuple in the toStep state by one tick and returns new values. The STEP function is only allowed to inspect tuples in the state given as context .	**Query R_D(Query q)** Calculates the read dependencies of q. It returns a query that captures all tuples needed in context to correctly step $q(\mathbb{S}^t)$.
Query R_X(Query q) Calculates the read exclusiveness of q. It returns a query that captures all tuples in $q(\mathbb{S}^t)$ that can be correctly stepped by only using $q(\mathbb{S}^t)$ as context .	**Query W_D(Query q)** Calculates the write dependencies of q. It returns a query p such that correctly stepping $p(\mathbb{S}^t)$ returns a state that contains all tuples in $q(\mathbb{S}^{t+1})$.
Query W_X(Query q) Calculates the write exclusiveness of q. It returns a query p such that correctly stepping $q(\mathbb{S}^t)$ returns a state that contains all tuples in $p(\mathbb{S}^{t+1})$.	**boolean DISJOINT(Query q0, Query q1)** Tests whether the queries q0 and q1 can have a nonempty intersection. It returns false if it is possible for q0 and q1 to ever select a tuple in common.

We remark that this table abstraction of state is purely *logical*. The *physical* representation of state could include additional data structures, such as a spatial index, to speed up processing. This separation allows us to model the state of a wide range of applications with tables, while not forfeiting the use of optimized representations in an actual implementation. In our programming model, we simply abstract state by an opaque State interface.

We denote the initial global state of the application by S^0, and the global state at the end of tick i by S^i. S^0 is typically generated dynamically or read from a file system. In the fish simulation, for example, the initial state of the fish school gets loaded from a checkpoint file. At each tick, updates to the state depend only on the state at the end of the previous tick, and not the history of past states. Thus, conceptually the time-stepped application logic encodes an *update function* GSTEP, s.t.:

$$S^{t+1} = \text{GSTEP}(S^t)$$

Partitioned Data Parallelism. Many time-stepped programs employ partitioned data parallelism, as observed in Section 2.2. Within a tick, we operate on partitions of the global state in parallel. At the end of the tick, we exchange data among processes to allow computation to advance again for the next tick. One has to make sure that such data parallel executions are equivalent to iterated applications of GSTEP to the global state.

To abstract data parallel execution in our programming model, the programmer firstly informs our framework of a partitioning method by implementing a partitioning function PART (Table 2.1). PART takes the number of processes n, optionally reads a global state, and outputs a list of n *selection queries*. A selection query Q (or *query*, for short) is a monotonic operation for selecting a subset

of tuples from the global state of the application.[1] It takes a global state S and obtains a subset $Q(S) \subseteq S$. The queries output by PART must form a partition of the global state. That is, at any tick, applying the queries to the global state S results in n disjoint subsets that completely cover S. For example, the fish simulation implements the following PART function:

```
List<Query> PART(int n) {
  File globalState = getGlobalStateFromCkpt();
  QuadTree qt = QuadTree(globalState,n);
  List<Query> queries = getLeafRectangles(qt);
  return queries;
}
```

As shown above, the fish simulation builds a quadtree structure containing exactly n leaves over the individual fish, while trying to balance the number of agents per leaf as much as possible [70, 124]. The result is a list of rectangles that partition the space. For this example, these rectangles are the implementation of our selection queries, which are distributed to n distinct processes. Periodic repartitioning may be required for load rebalancing, which can be implemented as reinvocations of function PART.

Now suppose we break up the global state S into n disjoint partitions $Q_i(S)$, s.t. $\bigcup_{i=1}^{n} Q_i(S) = S$. Unless the application is embarrasingly parallel, we cannot guarantee that GSTEP$(S^t) = \bigcup_{i=1}^{n}$ GSTEP$(Q_i(S^t))$. This is because the correct computation of partition $Q_i(S)$ may require GSTEP to inspect data from other partitions as context.

[1] Monotonic queries maintain the containment relationship between input states [97]. So adding tuples to a state cannot make a selection query over this state return less tuples.

To address this problem, we introduce two more functions: a *local initialization function* NEW(Q) and a *local update function* STEP(A, B). The local initialization function NEW(Q) takes a query Q calculated by the partition function PART. It creates the *local state* of a process P_i, denoted S_i, by applying Q to the global state. Details on how Q is calculated are presented in Section 2.5.

The local update function STEP(A, B) takes as input two states: a *state A to compute on* and a *context state B*. Note that tuples in both A and B are read-only, while the output state contains updated tuples in A and any other newly generated tuples from the result of the computation. This function agrees with the standard update in that:

$$\text{STEP}(S, S) = \text{GSTEP}(S), \ \forall \text{global states } S$$

In addition, we require STEP to be both *partitionable* and *distributive* for correct execution:

Property 1 (Partitionable)**.** *Let $\pi_{id}(S)$ denote the set of unique identifiers in S. Then for any states $S_a, S_b \subseteq S$ such that $S_a \cap S_b = \emptyset$,*

$$\pi_{id}(\text{STEP}(S_a, S)) \cap \pi_{id}(\text{STEP}(S_b, S)) = \emptyset \tag{2.1}$$

Property 2 (Distributive)**.** *For any states $S_a, S_b \subseteq S$ such that $S_a \cap S_b = \emptyset$,*

$$\text{STEP}(S_a, S) \cup \text{STEP}(S_b, S) = \text{STEP}(S_a \cup S_b, S) \tag{2.2}$$

Property 1 guarantees that the outputs of computations on partitions still forms a partition of the global state. Property 2 ensures that independent computations on the subsets of the global state can be recombined simply. These two properties are the key to parallelizing the computation. In practice, many

of our time-stepped applications perform updates on individual tuples while preserving their key values, which respects the above two properties.

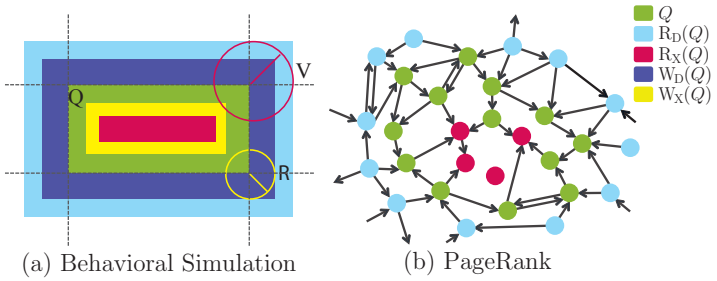
Returning to the fish simulation example, a single tick consists of each fish inspecting other fish that it can see to decide its own velocity for the next tick. This logic is coded in the following STEP function:

```
State STEP(State toStep, State context) {
  State result = getCleanState();
  for (Fish f in toStep) {
    for (Fish g in context, g visible to f) {
        ... // compute influence of g in f
    }
    if (isInformed(f)) {
       ...// balance with preferred direction
    }
    result.addFish(f, influence, balance);
  }
  return result;
}
```

The function STEP is applied to subsets of the fish relation, which are composed of tuples representing individual fish. It is easy to see that this STEP function is both partitionable and distributive.

2.4.2 Modeling Data Dependencies

Everything we have specified so far would be required for any data parallelization of a time-stepped application, and is not unique to our programming



(a) Behavioral Simulation (b) PageRank

Figure 2.1: R_D, R_X, W_D and W_X Functions

model. However, we still need to model a key aspect of time-stepped applications: data dependencies.

For applications that exhibit locality, stepping partition $Q_i(S)$ of a process P_i may not require the entire global state as context. Yet, the context has to be large enough to contain all data which the computation over $Q_i(S)$ needs to read. As long as all such data is included in the context state at every tick, STEP will generate the same result as having the entire global state given as context. In this case, we say that $Q_i(S)$ is *correctly* stepped.

Some of the context data required by STEP may not be in the local partition, and thus needs to be gathered and replicated from other processes. Therefore, the local state S_i generated by NEW(Q_i) must at a minimum include this replicated data, in addition to the corresponding partition data $Q_i(S)$.

In a classic data parallel implementation, the developer would need to hand-code this communication pattern for replication. However, in our programming model, developers are only asked to specify simple and intuitive dependency relationships between queries, which are declared in the functions R_D, R_X, W_D, and W_X (Table 2.1).

Figure 2.1(a) illustrates the implementation of the data dependency functions for the fish simulation. Formal definitions of the properties these func-

tions must respect can be found in Appendix A.1. As mentioned in Section 2.2, a fish can only see as far as its visibility range V. In this case, $R_D(Q)$ returns a query that contains all fish visible to some fish in $Q(S)$, for any global state S. In other words, $R_D(Q)$ comprises the read dependencies for computations of fish contained in $Q(S)$, and thus can be used as the context to step $Q(S)$. Similarly, $R_X(Q)$ returns a query that contains all the fish that cannot see (and thus do not depend on reads of) fish outside $Q(S)$:

```
Query RD(Query q) {
  Rect qr = (Rect) q;
  return new Rect(qr.lowLeftX - V, qr.lowLeftY - V,
                  qr.upperRightX + V, qr.upperRightY + V);
}

Query RX(Query q) {
  Rect qr = (Rect) q;
  return new Rect(qr.lowLeftX + V, qr.lowLeftY + V,
                  qr.upperRightX - V, qr.upperRightY - V);
}
```

In our experience, these functions are easy to specify; indeed, developers typically think in these terms when developing parallel applications.

As fish in our example are partitioned by their spatial locations, movement of a fish over the course of the simulation can change its responsible process. Therefore, computations over the local partition may need to write new data to other partitions within a tick. Such write dependencies can be captured through functions W_D and W_X. For example, suppose that the maximum distance a fish can move within a tick is given by a reachability parameter R. In this case,

$W_D(Q)$ can return a query that extends Q by the reachability R. In other words, $W_D(Q)$ selects the set of tuples such that correctly stepping this set produces all tuples that satisfy Q in the next tick. Similarly, $W_X(Q)$ returns a query that shrinks Q by the reachability R. Correctly stepping $Q(S)$ produces all tuples that satisfy $W_X(Q)$ in the next tick:

```
Query WD(Query q) {
  Rect qr = (Rect) q;
  return new Rect(qr.lowLeftX - R, qr.lowLeftY - R,
                  qr.upperRightX + R, qr.upperRightY + R);
}

Query WX(Query q) {
  Rect qr = (Rect) q;
  return new Rect(qr.lowLeftX + R, qr.lowLeftY + R,
                  qr.upperRightX - R, qr.upperRightY - R);
}
```

Not all time-stepped applications require the specification of all four functions above. For example, consider a standard PageRank computation. We can represent vertices in the graph as database tuples, and implement PART with a graph partitioning algorithm, such as METIS [82]. Figure 2.1(b) illustrates R_D and R_X in this problem. $R_D(Q)$ includes Q plus any vertex with an edge outgoing to a vertex contained in Q; $R_X(Q)$ includes only those elements of Q whose incoming edges all come from vertices of Q. However, since the graph structure is static and vertices only need to read from their incoming neighbors, W_D and W_X are simply identity functions. We include the full specification of PageRank using our programming model in Appendix A.2.2.

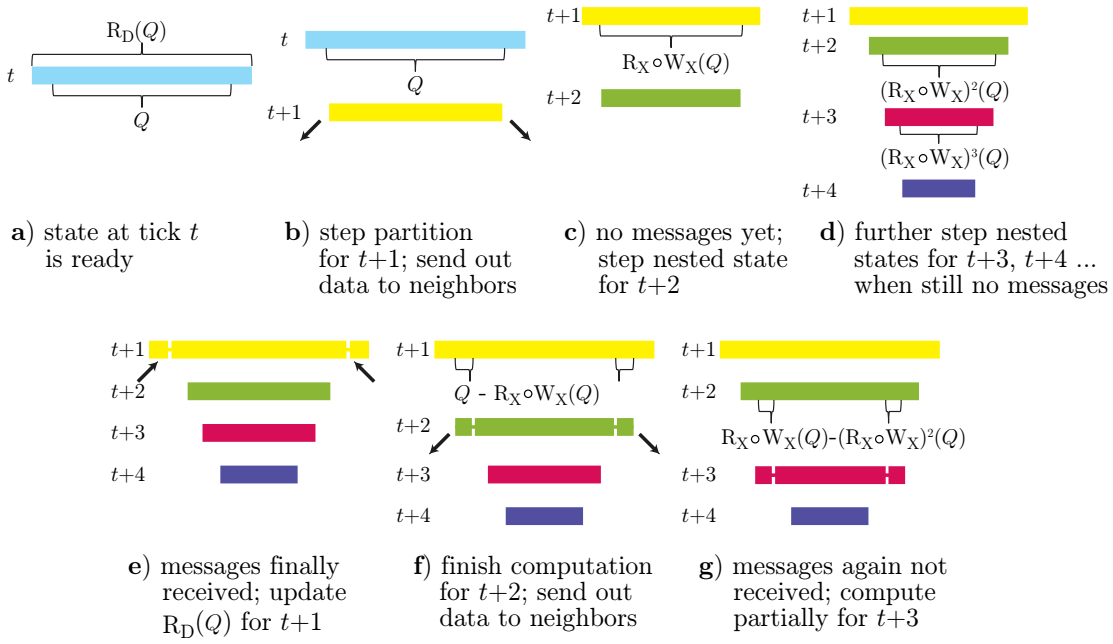


Figure 2.2: Dependency Scheduling.

Finally, our framework may need to operate on selection queries in order to automatically set up the communication pattern for the application. So we require programmers to specify one additional function, DISJOINT, to test for query disjointness. In the case of the fish simulation, the operation DISJOINT is easy to implement: It is just rectangle disjointness.

2.5 Jitter-Tolerant Runtime

We now describe how our jitter-tolerant runtime automatically implements communication and schedules computation given the primitives of our programming model. We assume that messages are reliably delivered (i.e., packets are never lost), and that messages between any pair of processes are not re-

ordered. However, as we have discussed previously, messages may be delayed by latency spikes. For simplicity, whenever it is clear from the context, in this section we slightly abuse notation and use Q_i to denote the subset $Q_i(S^t)$ of a global state S at tick t.

2.5.1 Local Synchronization

Traditional bulk synchronous implementations of time-stepped applications introduce global barriers between ticks: At the end of each tick, processors need to block while synchronizing their updated data with each other. The cost of these barriers is determined by the arrival time of the last message in the tick. If we can reduce the number of processes that need to synchronize at a barrier, we can reduce their cost. We observe that the groups of processes that need to synchronize with each other can be determined automatically by leveraging the data dependencies among states encoded in our programming model. If we can assert that a process will never read from or write to another process during the computation, no message exchanges are necessary between them.

The general condition under which two processes P_i and P_j must synchronize falls naturally from this observation. Suppose that after applying the partitioning function PART, we associate its i-th and j-th output queries Q_i and Q_j with P_i and P_j, respectively. Then, to determine if P_i should send messages to P_j, we invoke the following test:[2]

$$\neg\textsc{Disjoint}(Q_i, \mathrm{W_D} \circ \mathrm{R_D}(Q_j))$$

The idea is that $\mathrm{R_D}(Q_j)$ is complete upon having all tuples generated by correctly

[2]The operator $\circ$ is the classic function composition operator, applied from right to left.

Algorithm 1: Local Synchronization at Process P_i

Input: User-defined $\mathrm{R_D}$, $\mathrm{W_D}$, DISJOINT
Input: Q_i and $S_i^0 = \textsc{New}(\mathrm{R_D}(Q_i))$
Input: Number of timesteps T , Number of processors N

```
1:  for t = 1 to T do
2:    S_i^t = STEP(Q_i(S_i^{t-1}), S_i^{t-1});
3:    for j = 1 to N do
4:      if ¬DISJOINT(Q_i, W_D ∘ R_D(Q_j)) then
5:        SEND((W_D ∘ R_D(Q_j))(S_i^t), j)
6:      end if
7:    end for
8:    for j = 1 to N do
9:      if ¬DISJOINT(Q_j, W_D ∘ R_D(Q_i)) then
10:       S' = RECEIVE(j)
11:       S_i^t = S_i^t ∪ S'
12:     end if
13:   end for
14: end for
```

stepping $\mathrm{W_D} \circ \mathrm{R_D}(Q_j)$. So we can safely assert that process P_i will never need to communicate with P_j unless Q_i may ever include tuples in $\mathrm{W_D} \circ \mathrm{R_D}(Q_j)$. If the above test succeeds, we call P_i a *neighbor* of P_j.

Algorithm 1 shows how to replace global barriers with local synchronization using user-instantiated $\mathrm{R_D}$ and $\mathrm{W_D}$. The data is partitioned among processes according to PART, and we initialize the local state S_i of a process P_i to its partition data along with the corresponding read dependency. The STEP function is applied in parallel for each tick (Line 2). At tick boundaries, each process only exchanges messages with other processes that satisfy the condition above (Lines 3 to 13). To ensure correct execution, processes synchronize their appropriate read and write dependencies.

Correctness and Efficiency. Theorem 1 in Appendix A.1.2 states the correct-

ness of Algorithm 1 by demonstrating that it is equivalent to iteratively applying GSTEP to the global state. We expect Algorithm 1 to suffer performance degradation in the presence of network jitter. We explore how to address this deficiency in the next sections.

2.5.2 Dependency Scheduling

Note that although the communication pattern we derive above may avoid global barriers, processes with dependencies still need to synchronize at the end of every tick. As a result, network jitter in the cloud may still lead to long waits for incoming messages at these synchronization points. To deal with this problem, we introduce dependency scheduling, which advances partial computations over subsets of the tuples that do not depend on those incoming messages. We can find these subsets by making use of functions W_X and R_X: if a process is responsible for a partition Q, then the set returned by $W_X(Q)$ cannot be affected by data generated from other processes within a tick. $R_X \circ W_X(Q)$ further refines this set to tuples that only depend on data inside of $W_X(Q)$ for their computation. Therefore, it is safe to advance computation on $R_X \circ W_X(Q)$ before receiving messages from any processes.

For concreteness, suppose process P at tick t computes the partition specified by query Q, as illustrated in Figure 2.2(a). We can safely advance Q to the next tick $t+1$ (Figure 2.2(b)). At the end of this computation, we can check if messages have been received. If not, we can apply the above construction recursively, leading to the series:

$$R_X \circ W_X(Q), (R_X \circ W_X)^2(Q), \ldots, (R_X \circ W_X)^d(Q)$$

where parameter d, provided by the application developer, is the maximum depth allowed for scheduling. This idea is illustrated in Figures 2.2(c) and (d). Note that it is possible that for some $d' < d$, $(\mathrm{R_X} \circ \mathrm{W_X})^{d'}(Q)$ is already an empty set. If this is the case, we can stop further applying $\mathrm{R_X} \circ \mathrm{W_X}$.

When the messages from neighbors finally arrive, we can use them to update the tuples in $\mathrm{R_D}(Q)$ to the next tick $t + 1$ (Figure 2.2(e)). Now, we can finish the remaining computation in Q for $t + 2$ (Figure 2.2(f)). Intuitively, finishing computation for earlier ticks has higher priority over advancing computation even further to future ticks. This is because we want to send messages to our neighbors to unblock their computations as early as possible.

In order to advance the remainder $Q - \mathrm{R_X} \circ \mathrm{W_X}(Q)$ to $t + 2$, however, we may need to inspect data in $\mathrm{R_X} \circ \mathrm{W_X}(Q)$ at $t + 1$ as context. The maintenance of these multiple versions is illustrated in Figure 2.2 by the multiple horizontal bars.

Suppose at this point the next messages from our neighbors are again delayed. We can then continue the computation by stepping $\mathrm{R_X} \circ \mathrm{W_X}(Q) - (\mathrm{R_X} \circ \mathrm{W_X})^2(Q)$, advancing the contained tuples to tick $t + 3$ (Figure 2.2(g)).

Algorithm 2 shows the detailed description of the distributed dependency scheduling algorithm for each process P_i. As with Algorithm 1, we assume a total number of ticks T for the computation. The maximum scheduling depth is specified by d. The algorithm maintains a book-keeping array DEPTH, which holds the depth of the computation for each tick t, as well as a window of tick numbers $[t_w, t_c]$ (Line 1). t_w is the tick still waiting for messages from neighbors, while t_c is the tick to advance next.

At each iteration, P_i schedules computation whenever possible by calling the

Algorithm 2: Dependency Scheduling at Process P_i

Input: User-defined $\mathrm{W_X}$, $\mathrm{R_X}$, DISJOINT
Input: Q_i and $S_i^0 = \textsc{New}(\mathrm{R_D}(Q_i))$
Input: Number of timesteps T, Scheduling depth d

1: Initialize $t_c = t_w = 1$, DEPTH[1] = 0, DEPTH[2..T] = -1
2: **while** $t_w \leq T$ **do**
3: **if** $t_c \leq T$ **then**
4: /* schedule next computation to execute */
5: $Q_i^1 = (\mathrm{R_X} \circ \mathrm{W_X})^{\mathrm{DEPTH}[t_c]}(Q_i)$
6: **if** DEPTH[$t_c + 1$] = -1 **then** /* not initialized */
7: $S_i^{t_c} = \textsc{Step}(Q_i^1(S_i^{t_c-1}), S_i^{t_c-1})$
8: **else**
9: $Q_i^2 = (\mathrm{R_X} \circ \mathrm{W_X})^{\mathrm{DEPTH}[t_c+1]-1}(Q_i)$
10: $S_i^{t_c} = S_i^{t_c} \cup \textsc{Step}((Q_i^1 \setminus Q_i^2)(S_i^{t_c-1}), S_i^{t_c-1})$
11: **end if**
12: /* send data if t_c's computation finished */
13: **if** DEPTH[t_c] = 0 **then**
14: $\textsc{Send}(S_i^{t_c})$
15: **end if**
16: DEPTH[$t_c + 1$] = DEPTH[t_c] + 1; $t_c = t_c + 1$
17: **end if**
18: **repeat** /* wait if nothing is schedulable */
19: **if** $\textsc{TryReceive}(t_w)$ **then**
20: Update $S_i^{t_w}$ from messages received.
21: $t_w = t_w + 1$; $t_c = t_w$; DEPTH[t_c] = 0
22: **end if**
23: **until** $-1 <$ DEPTH[t_c] $\leq d$
24: **end while**

STEP function (Lines 3 to 17). To decide what to schedule next, P_i first obtains the subset at the current depth of t_c (Line 5). If the next tick $t_c + 1$ has not been scheduled yet, the whole subset at t_c can be advanced (Lines 6 to 8). Otherwise, P_i needs to update the difference between this subset and the one currently at $t_c + 1$ (Lines 9 to 11). In either case, the computation of STEP requires the state as

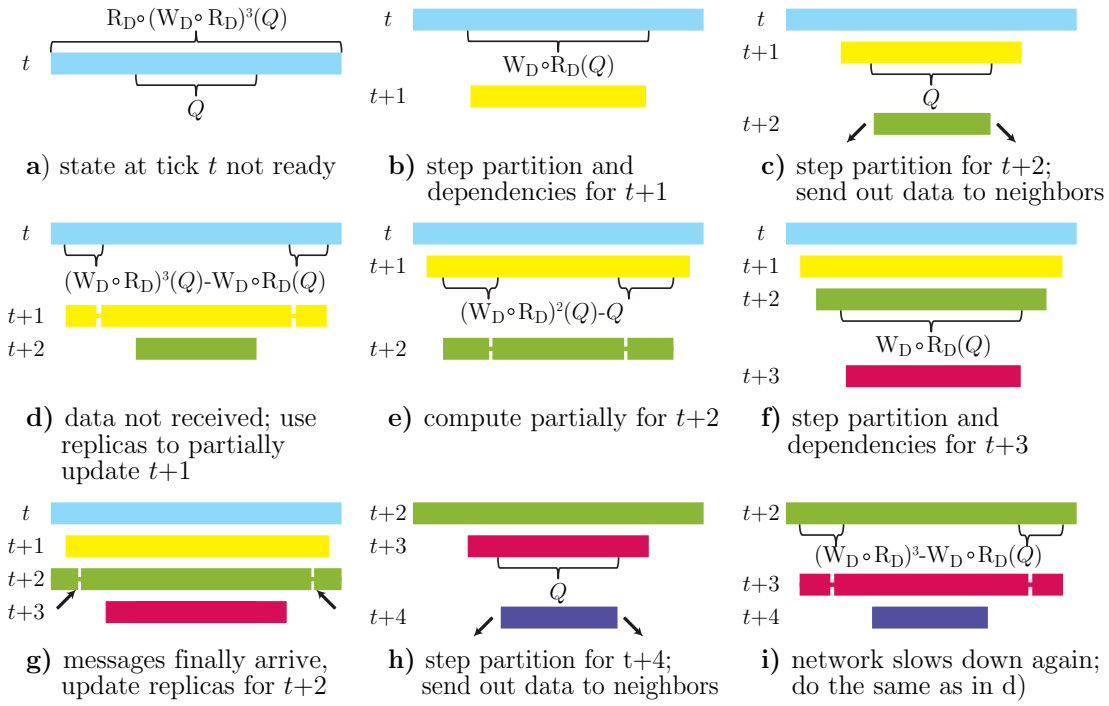


Figure 2.3: Computational Replication.

of time $t_c - 1$ as its context.

Whenever the depth of a tick reaches zero, its computation is complete. This implies P_i can send the corresponding update messages out to its neighbors and start working on the next tick (Lines 13 to 16). Finally, P_i waits for messages from neighbors until some computation can be scheduled (Lines 18 to 23). When all incoming messages for tick t_w have arrived, we update $S_i^{t_w}$ and set t_c such that the whole of $Q_i(S_i^{t_w+1})$ becomes ready for computation. Note that the user-instantiated DISJOINT function is called implicitly in the TRYRECEIVE and SEND operations. Therefore, P_i only blocks if nothing can be scheduled (i.e., the computation at t_c has already reached the maximum allowable depth). The latter condition could easily be replaced by a check of whether the subset at the

depth of t_c contains any data.

Correctness and Efficiency. Theorem 2 in Appendix A.1.2 states the correctness of Algorithm 2. With efficient query implementation and proper precomputation of dependency functions (Lines 5 and 9), the overhead of Algorithm 2 itself is negligible, since the body of the outer loop always schedules one STEP invocation, except when the process is blocked by communication.

2.5.3 Computational Replication

With dependency scheduling, we can overlap part of the future computation of a process with communication in the presence of a latency spike. However, since the data that depends on incoming messages cannot be updated, dependency scheduling does not allow the process to finish all the computations of the current tick and send out messages to its neighbors for the current tick. Consequently, latency waves still get propagated to other processes. To tackle this problem, we explore the idea of computational replication, which pays some extra computation to allow processes to complete the current tick in the absence of incoming messages. With computational replication, we redundantly perform the computation of neighbors locally, i.e., we *emulate message receipts* from them. This of course assumes that the computation of a tick can be made deterministic. Gladly, the STEP function already respects Properties 1 and 2. So in all time-stepped applications we studied, achieving determinism only required us to additionally ensure that the state of pseudorandom number generators were included in the state of the application.

Recall that at tick t, a process P_i steps Q_i and waits for messages from its

neighbors to update $R_D(Q_i)$. In order to emulate the receipt of these messages, the process needs to locally store $R_D \circ W_D \circ R_D(Q_i)$. The outermost layer of read dependency allows us to correctly step $W_D \circ R_D(Q_i)$. This computation produces all the writes necessary to obtain the state for the next tick of $R_D(Q_i)$. We can apply this idea recursively with more replicated data: By having m layers of replicas (i.e., $R_D \circ (W_D \circ R_D)^m(Q)$), we can proceed to tick $t+m$ without receiving any messages.

Since layers of replicas allow a process to step multiple ticks without receiving any messages, we can use them to reduce the frequency of message rounds from every tick to only every k ticks. Of course, if we have m layers of replicas, processes must exchange messages at least every $k = m + 1$ ticks. When k is exactly $m + 1$, computational replication corresponds to a generalization of the "ghost cells" technique from the HPC community (see Section 5.1). However, we observe that sometimes it may be more profitable to have $k \leq m$ in the cloud, since this allows for a second use of emulating message receipts: to unblock computation local to our process during a latency spike.

Again, we first illustrate this idea through an example, in which $k = 2$ and $m = 3$. Figure 2.3(a) shows three layers of replicas, up to $R_D \circ (W_D \circ R_D)^3(Q)$ for a process with partition query Q at tick t. As we only send messages every two ticks, we need to emulate message receipts for tick $t + 1$. This implies stepping $W_D \circ R_D(Q)$ so that $R_D(Q)$ reaches $t + 1$. After that, we can step Q to $t + 2$ (Figures 2.3(b) and (c)). At this point, we send messages to our neighbors. Suppose now the incoming messages from our neighbors are delayed by a latency spike. We can then use the additional layers of replicas to run useful computation over Q. The first step is to compute over $(W_D \circ R_D)^3(Q) - W_D \circ R_D(Q)$ at tick t and

then over $(W_D \circ R_D)^2(Q) - Q$ at tick $t + 1$ (Figures 2.3(d) and (e)). Note that here we compute over two layers of replicas at a time because we need to ensure that write dependencies are resolved for the innermost layer. Now we are again ready to compute over $W_D \circ R_D(Q)$, advancing $R_D(Q)$ to $t + 3$ (Figure 2.3(f)).

Suppose at this moment the messages from our neighbors finally arrive. We can append the data in those messages for tick $t + 2$ to the corresponding tuples (Figure 2.3(g)). As we had $R_D(Q)$ at $t + 3$, we can proceed and step Q to $t + 4$, sending messages out again to our neighbors (Figure 2.3(h)). The above procedure can be repeated if another latency spike again delays incoming messages (Figure 2.3(i)).

Algorithm 3 describes the distributed computational replication algorithm for each process P_i. The input to this algorithm is the same as for Algorithm 2, except that we have the two parameters k and m instead of parameter d. Given these parameters, process P_i will only communicate with its neighbors every k steps, but keep m replica layers. Similarly to Algorithm 2, we keep a WIDTH book-keeping array. This time it indicates the amount of replication at each tick. Tick number t_w represents the tick we are waiting on data from our neighbors (Line 1). Finally, as in Algorithm 2, TRYRECEIVE and SEND operate over all appropriate neighbors, and implicitly make use of DISJOINT.

At each tick in the computation, P_i first processes the data in Q_i and sends messages to its neighbors if the current tick is a multiple of k (Lines 3 to 6). Then P_i tries to receive incoming messages. If the messages from its neighbors have arrived, P_i can set the width of t_w to the full replication width m, as all replicas are updated with the data in the messages. After that, P_i advances t_w by k, since the next message will only come k steps later. If the messages are not available,

Algorithm 3: Computational Replication at Process P_i

Input: User-defined $\mathrm{R_D}$, $\mathrm{W_D}$, DISJOINT, k, m
Input: Q_i and $S_i^0 = \textsc{New}(\mathrm{R_D} \circ (\mathrm{W_D} \circ \mathrm{R_D})^m(Q_i))$
Input: Number of timesteps T

1: Init $t_w = 1$, WIDTH$[1..T] = -1$, WIDTH$[0] = m$
2: **for** $t = 1$ **to** T **do**
3: $S_i^t = \textsc{Step}(Q_i(S_i^{t-1}), S_i^{t-1})$
4: **if** $t \bmod k = 0$ **then**
5: $\textsc{Send}(S_i^t)$
6: **end if**
7: **while** WIDTH$[t] = -1$ **do**
8: **if** $\textsc{TryReceive}(t_w)$ **then**
9: Update $S_i^{t_w}$ from messages received.
10: WIDTH$[t_w] = m$; $t_w = t_w + k$
11: **else** /* try to calculate incoming updates */
12: $p = t - 1$
13: **while** WIDTH$[p] \leq$ WIDTH$[p+1] + 1$ **and** $p \geq t - m$ **do**
14: $p = p - 1$
15: **end while**
16: **if** WIDTH$[p] >$ WIDTH$[p+1] + 1$ **then**
17: $Q_i^1 = (\mathrm{W_D} \circ \mathrm{R_D})^{\text{WIDTH}[p+1]+2}$
18: $Q_i^2 = (\mathrm{W_D} \circ \mathrm{R_D})^{\text{WIDTH}[p+1]+1}$
19: $S_i^{p+1} = S_i^{p+1} \cup \textsc{Step}((Q_i^1 \setminus Q_i^2)(S_i^p), S_i^p)$
20: WIDTH$[p+1] =$ WIDTH$[p+1] + 1$
21: **end if**
22: **end if**
23: **end while**
24: **end for**

P_i needs to emulate their receipts by first finding the innermost replica layer that can be advanced (Lines 11 to 15). The difference in width between this replica layer and the subsequent layer must be at least two, as otherwise processing the replica layer will not unblock the subsequent layer. When P_i finds such a replica layer, it can process the tuples in this layer and increase its width (Lines 16 to 20).

After enough replica layers are processed and the width of tick t drops to zero, P_i can then advance Q_i to the next tick without blocking on communication.

For ease of exposition, Algorithm 3 presents pure replication without combining it with dependency scheduling; however, these two techniques can work together and we show their combined effect in our experiments (Section 2.6).

Correctness and Efficiency. Theorem 3 in Section A.1.2 states the correctness of Algorithm 3. Similarly to Algorithm 2, the overhead of Algorithm 3 can be reduced by efficient query implementation and proper precomputation of dependency functions (Lines 17 to 19). In addition, the redundant computations performed by the algorithm are designed to be executed only during the time that the process would be idle waiting on messages. Thus, as we expect the value of m to be a small constant, we anticipate that the remaining overhead of this algorithm be negligible.

2.6 Experiments

In this section, we present experimental results for three different time-stepped applications using our jitter tolerant runtime. The goals of our experiments are two-fold: (i) We want to validate the effectiveness of the various optimization techniques introduced in Section 2.5 in a real cloud environment; (ii) We want to evaluate how the optimizations introduced by our runtime can improve the parallel scalability of these applications.

2.6.1 Setup

Implementation. We have built a prototype of our jitter-tolerant runtime in C++. The runtime exposes the programming model described in Section 2.4 as its API. All the communication is done using MPI. In order to focus on the effects of network communication, all our application code is single-threaded and we ran one runtime process per virtual machine.

Application Workloads. We have implemented three realistic time-stepped applications: a fish school behavioral simulation [39], a linear solver using the Jacobi method [16], and a message-passing algorithm that computes PageRank [25]. The fish simulation has already been explained throughout this chapter. Regarding parallel processing, we use two-dimensional grid partitioning to distribute the fish agents across processes. The implementation of this simulation follows closely the example pseudocode shown in Section 2.4.

The Jacobi solver is a common building block used to accelerate Krylov subspace methods such as conjugate gradients and to derive smoothers for multigrid methods. It follows a communication pattern among cells of the matrix with high spatial locality: At each step, each cell needs to communicate its values to its spatial neighbors. In our experiments, we implemented a 2D head diffusion solver, in which each process is allocated a fixed-size 1,000 x 1,000 block of the matrix. Pseudocode for our implementation of this method can be found in Appendix A.2.1.

For the PageRank algorithm, we used the U.S. Patent Citation Network graph with 3,774,768 vertices and 16,518,948 edges in our experiments [89]. In addition, we used the popular METIS graph partitioning toolkit in PART to com-

pute a per-vertex partitioning of the input graph [82]. Pseudocode for PageRank is given in Appendix A.2.2.

Our techniques target compensating for network jitter, and not delays caused by systematic load imbalance. The reader is referred to Hendrickson and Devine for a description of techniques for the latter problem [70]. Nevertheless, our techniques may still be helpful when latency spikes exceed the delays caused by imbalanced load. To fairly measure the contribution of our techniques to performance, we have tuned the applications above so that load would be as well balanced as possible among the executing processes. We could achieve nearly perfect load balance for both the fish simulation and the Jacobi solver. For PageRank, however, we were limited to the quality of the partitioning generated by METIS. In addition, as the fish simulation and Jacobi solver applications follow a spatial communication pattern, by analyzing data dependencies we can bound the number of neighbors for each process by a constant. However, the same is not true of PageRank: the small-world property of the graph structure of our dataset results in a nearly all-to-all communication pattern. As a consequence, we expect the effectiveness of our optimizations on this application to be reduced.

We tuned state sizes by partitioning the state up until we started to observe diminishing returns on parallel efficiency. All of the applications above operate over a modest-sized state smaller than a few tens of megabytes per process. Even though our algorithms may need to keep multiple versions of the updated parts of the state, these additional copies fit comfortably in main memory.

In all the experiments, our metric is the overall tick throughput, in agent

(fish simulation) or cell (Jacobi solver) or edge (PageRank) ticks per second.

Hardware Setup. We ran experiments in the Amazon Elastic Compute Cloud (Amazon EC2). In order to conduct large scale experiments within our limited budget, we chose to use large instances (m1.large) in all experiments. Each large instance has two 2.26GHz Xeon cores with 6MB cache and 7.5GB main memory. We also forced all instances to reside in the same availability zone. Given the similar distribution of message latencies between these instances and cluster instances (Figure 1.3), we believe our results will be qualitatively similar to runs in these other instances as well. Unless otherwise stated, all our experiments are run with 50 large instances.

We have packaged our experimental setup as a public Amazon Linux AMI; documentation and source code are available at [55].

2.6.2 Methodology

Clearly, our measurements are affected by the network conditions at Amazon EC2. Given that this is a cloud environment, we cannot guarantee identical network conditions across multiple experiments. As a result, absolute measurements are not repeatable. So we must devise a scheme to obtain repeatable relative rankings of the techniques we evaluate.

We exercised care in a number of aspects of the experiment setup. First, as we mentioned previously, we only allocated instances within the same availability zone. In addition, we made sure to use the same set of instances for all of our measurements of all methods. The rationale is that we wish to get a network

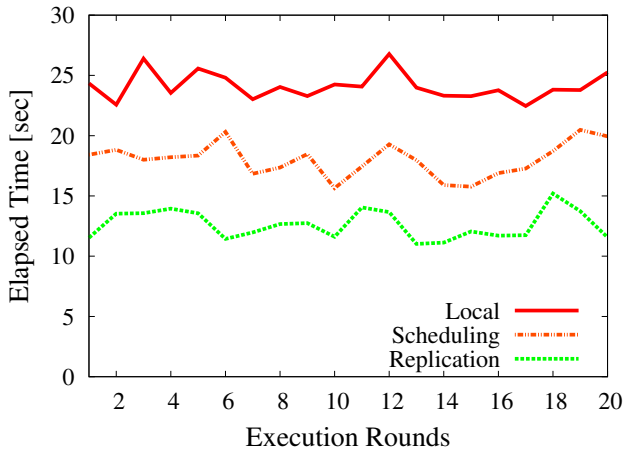


Figure 2.4: Variance Over Time: Jacobi

setup that is as invariant as possible across measurements.

Unfortunately, this is not enough. Even with the same set of instances in the same availability zone, we have observed that the Amazon EC2 network is not only unstable with a high rate of abnormal message delays, but also exhibits high median latency variation over time. In order to conduct meaningful comparisons between different techniques, we must account for this temporal variation in network performance. As a result, the following procedure is carried out to obtain the performance measurements. First, we execute all techniques in rounds of fixed order. A performance measurement consists of at least 20 consecutive executions of these rounds. We report standard deviation with error bars. This methodology seeks to ensure that each round sees roughly comparable distributions of message latencies. We had to tune manually the maximum running time of each round so that it was smaller than the time it took for the network to exhibit large changes in message delay distributions. Nevertheless, we were still able to ensure the execution of each technique in each round to be of at least 500 ticks.

Figure 2.4 illustrates this temporal variation effect. We compare the different techniques from Section 2.5 on the Jacobi solver application. As explained above, we alternate the execution of these techniques in each round. The x-axis plots 20 executions of such rounds, while the y-axis shows the raw elapsed time for each technique at each round. We can observe that the results of different techniques exhibit the same temporal trends due to variance in network performance; at the same time, the measurements still clearly demonstrate which techniques are superior.

While relative rankings among techniques are made comparable by the above methodology, we stress that the absolute values of results shown in the figures in this section are not directly comparable with each other. This is the case even if they are from the same application and use the same technique with the same parameters, given that we cannot control variations in network load over longer time scales.

2.6.3 Results

Effect of Individual Optimizations. Figure 2.5 shows the performance of the fish simulation with dependency scheduling. When the depth of scheduling is allowed to reach only a single tick forward in time, tick throughput already increases by roughly 30% when compared to local synchronization (i.e., Algorithm 1; we are not comparing with the naive bulk synchronous implementation) since the first layer of scheduling enables computation to overlap with communication. Allowing even larger depth of scheduling does not significantly improve throughput. That is because after the messages are received

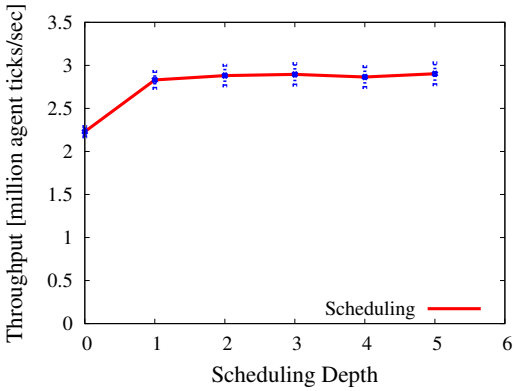


Figure 2.5: Scheduling: Fish Sim

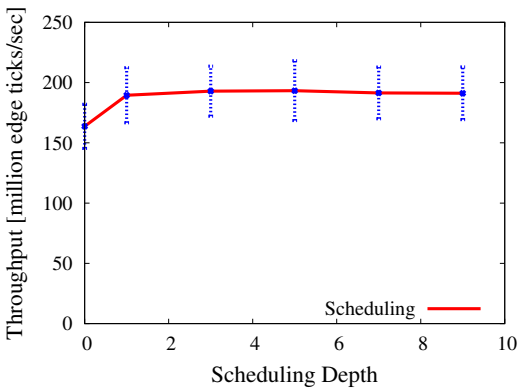


Figure 2.6: Scheduling: PageRank

from our neighbors, only a small amount of computation is left to let us send messages for the next tick. However, as we synchronize every tick, the benefit of computing the second layer earlier, rather than over the communication time of the next tick, is minimal. As shown in Figure 2.6, we observe similar behavior for PageRank. The results for the Jacobi solver are omitted: They are also similar to the results of fish simulation.

Given the above effect, we need to communicate less often than every tick in order to realize the potential benefits of scheduling. This can be achieved by computational replication, which we first evaluate independently. Figures 2.7 to 2.9 display the results for the fish simulation. Each figure shows a different setting for the communication avoidance parameter k. Given the value of k, we vary the number of layers of replication m. In Figure 2.7, throughput reaches its peak at $m = 2$, dropping significantly after that point. The reason is that as we communicate every tick, the message sizes are small and communication cost is dominated by latency. Thus, it is beneficial to send more than one layer of replicas together to better compensate for jitter. However, as we increase the degree of replication, the overhead in message sizes overshadows the benefits

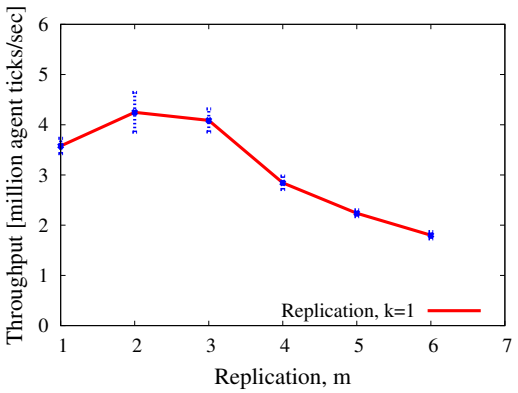


Figure 2.7: Replication, $k = 1$: Fish Sim

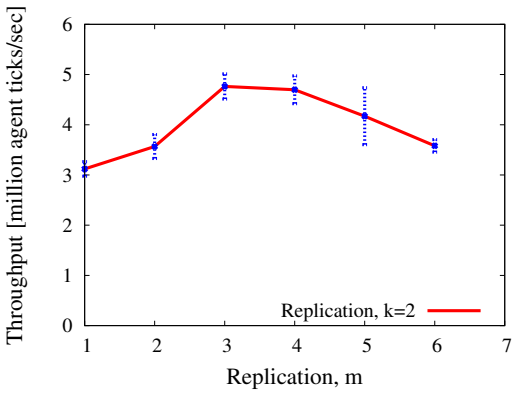


Figure 2.8: Replication, $k = 2$: Fish Sim

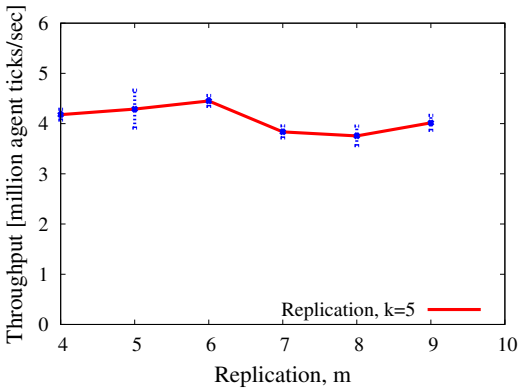


Figure 2.9: Replication, $k = 5$: Fish Sim

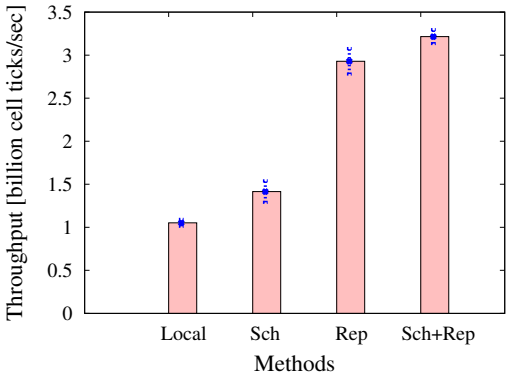


Figure 2.10: Effect of Combination: Jacobi

in tolerance to jitter.

Figures 2.8 and 2.9 exhibit similar patterns. However, in Figure 2.9, throughput increases from $m = 8$ to $m = 9$. In this situation, we increase the size of replication information by only one eighth; however, now we are able to redundantly compute enough to send messages for the next communication round. This unblocks other processes earlier, increasing performance. We have also tested many other parameter settings for both the fish simulation and the Jacobi solver. The best setting we could devise for the former was of $k = 2$ and $m = 3$; for the latter, $k = 3$ and $m = 5$.

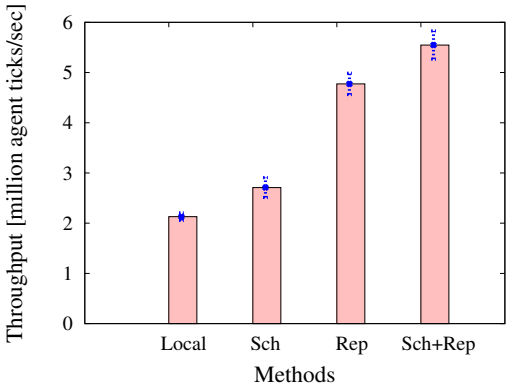


Figure 2.11: Effect of Combination: Fish Sim

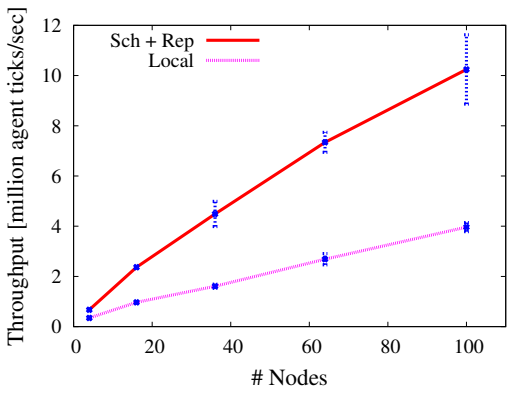


Figure 2.12: Scalability: Fish Sim

For all of our experiments, we also measured separately the breakdown of execution time into time spent in calls to the STEP function, communication wait time, and time spent in all other parts of the runtime. We observed that the latter time always corresponded to at most 0.02% of execution time. This confirms our expectations with respect to the efficiency of Algorithms 2 and 3 (Sections 2.5.2 and 2.5.3).

Finally, for the PageRank application, the small-world structure of the graph we use implies a nearly all-to-all communication pattern. In addition, even a one-hop dependency of a graph partition can lead to a significant fraction of the whole graph. As replication is obviously counterproductive in this situation, we do not show experimental results with replication for this application.

Effect of Combined Optimizations.

Figures 2.10 and 2.11 show the effect of combinations of multiple optimizations. As we have seen before, increasing the scheduling depth does not hurt throughput, so in order to take maximum advantage of scheduling we set its depth to 10. For replication, we use the best setting from the previous experi-

ment.

While replication brings the largest benefit as an individual technique to both applications, replication combined with scheduling shows even better performance. The improvement in throughput using this combined technique is over a factor of 3 for the Jacobi solver and around a factor of 2.5 for the fish simulation. This comes from the fact that scheduling can absorb part of a latency spike without increasing the size of messages exchanged among processes. Therefore, it can help replication achieve higher throughput without introducing any extra communication overhead.

Impact on Parallel Scalability. We measure the parallel scaleup performance of our runtime by varying the number of instances from 4 to 100 while keeping the average workload of each instance constant. Figure 2.12 shows scalability results for the fish simulation; the Jacobi solver shows similar trends. One can see that our best combination of techniques can further improve the near-linear scalability compared with local synchronization.

2.7 Conclusions

We have shown how time-stepped applications can deal with large variance in message delivery times, a key characteristic of today's cloud environments. Our novel data-driven programming model abstracts the state of these applications as tables and expresses data dependencies among sets of tuples as queries. Based on our programming model, our runtime achieves jitter-tolerance transparently to the programmer while improving throughput of several typical applications by up to a factor of three. As future work, we plan to investigate

how to apply our framework to legacy code automatically or with little human input. Another interesting direction is quantifying the energy impact of our redundant computation techniques and analyzing the resulting trade-off with time-to-solution. Finally, we will investigate jitter-tolerance techniques for a much wider class of applications, e.g., transactional systems and replicated state machines.

CHAPTER 3

IMPROVING RESPONSE TIME UNDER LATENCY HETEROGENEITY

3.1 Introduction

In Section 1.1.1, we have shown the heterogeneous network connectivity between instances in Amazon EC2. In Appendix B.3, we show that this intuition is not restricted to Amazon EC2: Similar observations of latency heterogeneity and mean latency stability can be made in other main public cloud service providers, namely Google Compute Engine and Rackspace Cloud Server.

This chapter examines how developers can carefully tune the deployment of their distributed applications in public clouds. At a high level, we make two important observations: (i) If we carefully choose the mapping from nodes (components) of distributed applications to instances, we can potentially prevent badly interconnected pairs of instances from communicating with each other; (ii) if we over-allocate instances and terminate instances with bad connectivity, we can potentially improve application response times. These two observations motivate our general approach: A cloud tenant has a target number of components to deploy onto x virtual machines in the cloud. In our approach, she allocates x instances plus a small number of additional instances (say $x/10$). She then carefully selects which of these $1.1 \cdot x$ instances to use and how to map her x application components to these selected virtual machines. She then terminates the $x/10$ over-allocated instances.

Our general approach could also be directly adopted by a cloud provider – potentially at a price differential – but the provider would need to widen its API

to include latency-sensitivity information. Since no cloud provider currently allows this, we take the point of view of the cloud tenant, by whom our techniques are immediately deployable.

Contributions of this Chapter. In this chapter, we introduce the problem of deployment advice in the cloud and instantiate a concrete deployment advisor called ClouDiA (*Clou*d *D*eployment *A*dvisor).

1. ClouDiA works for two large classes of data-driven applications. The first class, which contains many HPC applications, is sensitive to the worst link latency, as this latency can significantly affect total time-to-solution in a variety of scientific applications [2, 23, 47, 83]. The second class, represented by search engines as well as web services and portals, is sensitive to the longest path between application nodes, as this cost models the network links with the highest potential impact on application response time. ClouDiA takes as input an application communication graph and an optimization objective, automatically allocates instances, measures latencies, and finally outputs an optimized node deployment plan (Section 3.2). To the best of our knowledge, our methodology is the first to address deployment tuning for latency-sensitive applications in public clouds.

2. We formally define the two node deployment problems solved by ClouDiA and prove the hardness of these problems. The two optimization objectives used by ClouDiA – largest latency and longest critical path – model a large class of current distributed cloud applications that are sensitive to latency (Section 3.3).

3. We present an optimization framework that can solve these two classes

of problems, and explore multiple algorithmic approaches. In addition to lightweight greedy and randomization techniques, we present different solvers for the two problems based on mixed-integer and constraint programming. We discuss optimizations and heuristics that allow us to obtain high-quality deployment plans over the scale of hundreds of instances (Section 3.4).

4. We discuss methods to obtain accurate latency measurements (Section 3.5) and evaluate our optimization framework with both synthetic and real distributed applications in Amazon EC2. We observe 15%-55% reduction in time-to-solution or response times. These benefits come exclusively from optimized deployment plans, and require no changes to the specific application (Section 3.6).

This chapter extends and subsumes its earlier conference version [166]. The additional contributions of this chapter are: (a) the exploration of lightweight algorithmic approaches to solve the node deployment problem under the two optimization objectives used by ClouDiA (Sections 3.4.3 and 3.4.5 as well as experimental results in Section 3.6.5); (b) the exploration of additional metrics to model communication cost other than mean latency, namely mean latency plus standard deviation, and latency at the 99^{th} percentile (Section 3.3.2 and experimental results in Section 3.6.4); (c) a discussion of overlapped execution of ClouDiA with target applications (Section 3.2.2); (d) the confirmation of the same effects of latency heterogeneity and mean latency stability in public cloud providers other than Amazon Web Services, namely Google Compute Engine and Rackspace Cloud Server (Appendix B.3).

3.2 Tuning for Latency-Sensitive Applications in the Cloud

To give a high-level intuition for our approach, we first describe the classes of applications we target in Section 3.2.1. We then describe the architecture that ClouDiA uses to suggest deployments for these applications in Section 3.2.2.

3.2.1 Latency-Sensitive Applications

We can classify latency-sensitive applications in the cloud into two broad classes: high-performance computing applications, for which the main performance goal is *time-to-solution*, and service-oriented applications, for which the main performance goal is *response time for service calls*.

Goal: Time-to-solution. A number of HPC applications simulate natural processes via long-running, distributed computations. For example, consider the simulation of collective animal movement published by Couzin et al. in Nature [40]. In this simulation, a group of animals, such as a fish school, moves together in a two-dimensional space. Animals maintain group cohesion by observing each other. In addition, a few animals try to influence the direction of movement of the whole group, e.g., because they have seen a predator or a food source. This simulation can be partitioned among multiple compute nodes through a spatial partitioning scheme [143]. At every time step of the simulation, neighboring nodes exchange messages before proceeding to the next time step. As the end of a time step is a logical barrier, worst-link latency essentially determines communication cost [2, 23, 83, 167]. Similar communication patterns are common in multiple linear algebra computations [47]. Another ex-

ample of an HPC application where time-to-solution is critical is dynamic traffic assignment [145]. Here traffic patterns are extrapolated for a given time period, say 15 min, based on traffic data collected for the previous period. Simulation must be faster than real time so that simulation results can generate decisions that will improve traffic conditions for the next time period. Again, the simulation is distributed over multiple nodes, and computation is assigned based on a graph partitioning of the traffic network [145]. In all of these HPC applications, time-to-solution is dramatically affected by the latency of the worst link.

Goal: Service response time. Web services and portals, as well as search engines, are prime cloud applications [10, 58, 140]. For example, consider a web portal, such as Yahoo! [116] or Amazon [140]. The rendering of a web page in these portals is the result of tens, or hundreds, of web service calls [106]. While different portions of the web page can be constructed independently, there is still a critical path of service calls that determines the server-side communication time to respond to a client request. Latencies in the critical path add up, and can negatively affect end-user response time.

3.2.2 Architecture of ClouDiA

Figure 3.1 depicts the architecture of ClouDiA. The dashed line indicates the boundary between ClouDiA and public cloud tenants. The tuning methodology followed by ClouDiA comprises the following steps:

1. Allocate Instances: A tenant specifies the communication graph for the application, along with a maximum number of instances at least as great as the required number of application nodes. CloudDiA then automatically allo-

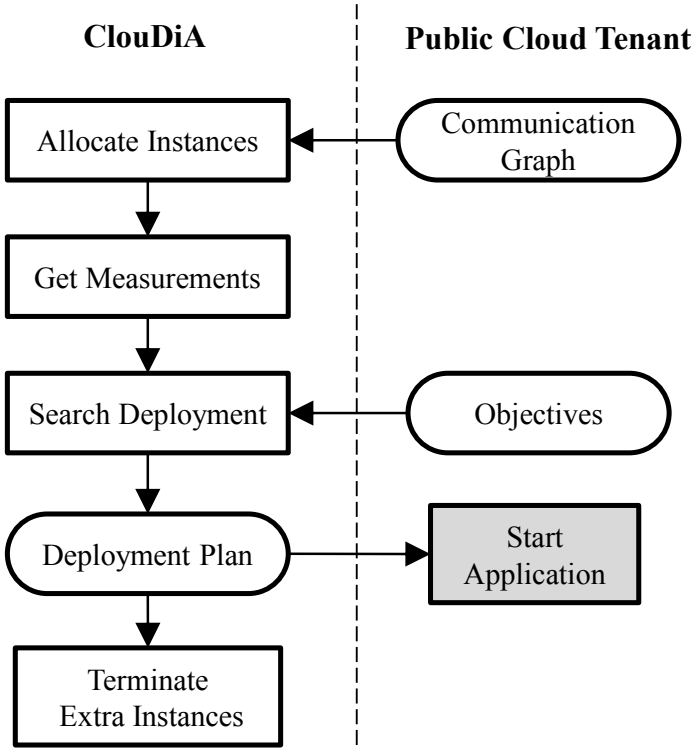


Figure 3.1: Architecture of ClouDiA

cates cloud instances to run the application. Depending on the specified maximum number of instances, ClouDiA will over-allocate instances to increase the chances of finding a good deployment.

2. Get Measurements: The pairwise latencies between instances can only be observed after instances are allocated. ClouDiA performs efficient network measurements to obtain these latencies, as described in Section 3.5. The main challenge is reliably estimating the mean latencies quickly, given that time spent in measurement is not available to the application.

3. Search Deployment: Using the measurement results, together with the optimization objective specified by the tenant, CloudDiA searches for a "good" deployment plan: one which avoids "bad" communication links. We formalize

this notion and pose the node deployment problem in Section 3.3. We then formulate two variants of the problem that model our two classes of latency-sensitive applications. We prove the hardness of these optimization problems in Appendix B.1. Given the hardness of these problems, traditional methods cannot scale to realistic sizes. We propose techniques that significantly speed up the search in Section 3.4.

4. Terminate Extra Instances: Finally, ClouDiA terminates any over-allocated instances and the tenant can start the application with an optimized node deployment plan.

Adapting to changing network conditions. The architecture outlined above assumes that the application will run under relatively stable network conditions. We believe this assumption is justified: The target applications outlined in Section 3.2.1 have long execution time once deployed, and our experiments in Figure 1.2 show stable pairwise latencies in EC2. In the future, more dynamic cloud network infrastructure may become the norm. In this case, the optimal deployment plan could change over time, forcing us to consider dynamic re-deployment. We envision that re-deployment can be achieved via iterations of the architecture above: getting new measurements, searching for a new optimal plan, and re-deploying the application.

Two interesting issues arise with iterative re-deployment. First, we need to consider whether information from previous runs could be exploited by a new deployment. Unfortunately, previous runs provide no information about network resources that were *not* used by the application. In addition, as re-deployment is triggered by changes in network conditions, it is unlikely that network conditions of previous runs will be predictive of conditions of future

runs. Second, re-deployment should not interrupt the running application, especially in the case of web services and portals. Unfortunately, current public clouds do not support virtual machine live migration [4, 113, 148]. Without live migration, complicated state migration logic would have to be added to individual cloud applications.

Overlapping ClouDiA with application execution. We envision one further improvement to our architecture that will become possible if support for VM live migration or state migration logic becomes pervasive: Instead of wasting idle compute cycles while ClouDiA performs network measurements and searches for a deployment plan, we could instead begin execution of the application over the initially allocated instances, in parallel with ClouDiA. Clearly, this strategy may lead to interference between ClouDiA's measurements and the normal execution of the application, which would have to be carefully controlled. In addition, this strategy would only pay off if the state migration cost necessary to re-deploy the application under the plan found by ClouDiA would be small enough compared to simply running ClouDiA as outlined in Figure 3.1.

3.3 The Node Deployment Problem

In this section, we present the optimization problems addressed by ClouDiA. We begin by discussing how we model network cost (Sections 3.3.1 and 3.3.2). We then formalize two versions of the node deployment problem (Section 3.3.3).

3.3.1 Cost Functions

When a set of instances is allocated in a public cloud, in principle any instance in the set can communicate with any other instance, possibly at differing costs. Differences arise because of the underlying physical network resources that implement data transfers between instances. We call the collection of network resources that interconnect a pair of instances the *communication link* between them. We formalize communication cost as follows.

Definition 1 (Communication Cost)**.** Given a set of instances S, we define $C_{\mathcal{L}} : S \times S \rightarrow \mathbb{R}$ as the communication cost function. For a pair of instances $i, j \in S$, $C_{\mathcal{L}}(i, j)$ gives the communication cost of the link from instance i to j.

$C_{\mathcal{L}}$ can be defined based on different criteria, e.g., latency, bandwidth, or loss rate. To reflect true network properties, we assume costs of links can be asymmetric and the triangle inequality does not necessarily hold. In this chapter, given the applications we are targeting, we focus solely on using *network latency* as a specific instance for $C_{\mathcal{L}}$. Extending to other network cost measurements is future work.

Our definition of communication cost treats communication links as essentially independent. This modeling decision ignores the underlying implementation of communication links in the datacenter. In practice, however, current clouds tend to organize their network topology in a tree-like structure [19]. A natural question is whether we could provide more structure to the communication cost function by reverse engineering this underlying network topology. Approaches such as Sequoia [119] deduce underlying network topology by mapping application nodes onto leaves of a virtual tree. Unfortunately, even though

these inference approaches work well for Internet-level topologies, state-of-the-art methods cannot infer public cloud environments accurately [17, 35].

Even if we could obtain accurate topology information, it would be non-trivial to make use of it. First, there is no guarantee that the nodes of a given application can all be allocated to nearby physical elements (e.g., in the same rack), so we need an approach that fundamentally tackles differences in communication costs. Second, datacenter topologies are themselves evolving, with recent proposals for new high-performance topologies [103]. Optimizations developed for a specific tree-like topology may no longer be generally applicable as new topologies are deployed. Our general formulation of communication cost makes our techniques applicable to multiple different choices of datacenter topologies. Nevertheless, as we will see in Section 3.6, we can support a general cost formulation, but optimize deployment search when there are uniformities in the cost, e.g., clusters of links with similar cost values.

To estimate the communication cost function $C_{\mathcal{L}}$ for the set of allocated instances, CloudDiA runs an efficient network measurement tool (Section 3.5). Note that the exact communication cost is application-dependent. Applications communicating messages of different sizes can be affected differently by the network heterogeneity. However, for *latency-sensitive applications*, we expect application-independent network latency measurements can be used as a good performance indicator, although they might not precisely match the exact communication cost in application execution. This is because our notion of cost need only discriminate between "good" and "bad" communication links, rather than accurately predict actual application runtime performance.

3.3.2 Metrics for Communication Cost

Even if we focus our attention solely on network latency, there are still multiple ways to measure and characterize such latency. The most natural metric, shown in Figures 1.1 and 1.2, is mean latency, which captures the average latency behavior of a link. However, some applications are particularly sensitive to latency jitter, and not only heterogeneity in mean latency [167]. For these applications, an alternate metric which combines the mean latency with the standard deviation on latency measurements may be the most appropriate. Finally, demanding applications may seek latency guarantees at a high percentile of the latency distribution.

While all the above metrics provide genuine characterizations of different aspects of network latency, two additional considerations must be taken into account before adopting a latency metric. First, since in our framework communication cost is used merely to differentiate "good" from "bad" links, correlated metrics behave effectively in the same way. So a non-straightforward metric other than mean latency only makes sense if it is not significantly correlated with the mean. Second, any candidate latency metric should guide the search process carried out by ClouDiA such that lower cost deployments represent deployments with lower actual time-to-solution or response time. Since most latency-sensitive applications tend to be significantly affected by mean latency, it is not clear whether other candidate latency metrics will lead to deployments with better application performance. We explore these considerations experimentally in Section 3.6.

3.3.3 Problem Formulation

To deploy applications in public clouds, a mapping between logical application nodes and cloud instances needs to be determined. We call this mapping a *deployment plan*.

Definition 2 (Deployment Plan)**.** Let S be the set of instances. Given a set N of application nodes, a deployment plan $\mathcal{D} : N \to S$ is a mapping of each application node $n \in N$ to an instance $s \in S$.

In this dissertation, we require that $\mathcal{D}$ be an injection, that is, each instance $s \in S$ can have at most one application node $n \in N$ mapped to it. There are two implications of this definition. On one hand, we do not collocate application nodes on the same instance. In some cases, it might be beneficial to collocate services, yet we argue these services should be merged into application nodes before determining the deployment plan. On the other hand, it is possible that some instances will have no application nodes mapped to them. This gives us flexibility to over-allocate instances at first, and then shutdown those instances with high communication cost.

In today's public clouds, tenants typically determine a deployment plan by either a default or a random mapping. CloudDiA takes a more sophisticated approach. The deployment plan is generated by solving an optimization problem and searching through a space of possible deployment plans. This search procedure takes as input the communication cost function $C_{\mathcal{L}}$, obtained by our measurement tool, as well as a *communication graph* and a *deployment cost function*, both specified by the cloud tenant.

Definition 3 (Communication Graph)**.** Given a set N of application nodes, the

communication graph $G = (V, E)$ is an undirected graph where $V = N$ and $E = \{(i, j) | i, j \in N \wedge \mathsf{talks}(i, j)\}$.

The talks relation above models the application-specific communication patterns. When defining the communication graph through the talks relation, the cloud tenant should only include communication links that have impact on the performance of the application. For example, those links first used for bootstrapping and rarely used afterwards should not be included in the communication graph. An alternative formulation for the communication graph would be to add weights to edges, extending the semantics of talks. We leave this to future work.

We find that although such a communication graph is typically not hard to extract from the application, it might be a tedious task for a cloud tenant to generate an input file with $O(|N|^2)$ links. ClouDiA therefore provides *communication graph templates* for certain common graph structures such as meshes or bipartite graphs to minimize human involvement.

In addition to the communication graph, a deployment cost function needs to be specified by the cloud tenant. At a high level, a deployment cost function evaluates the cost of the deployment plan by observing the structure of the communication graph and the communication cost for links in the given deployment. The optimization goal for CloudDiA is to generate a deployment plan that minimizes this cost.

Given a deployment plan $\mathcal{D}$, a communication graph G, and a communication cost function $C_{\mathcal{L}}$, we define $C_{\mathcal{D}}(\mathcal{D}, G, C_{\mathcal{L}}) \in \mathbb{R}$ as the deployment cost of $\mathcal{D}$. $C_{\mathcal{D}}$ must be monotonic on link cost and invariant under exchanging nodes that

are indistinguishable using link costs.

Now, we can formally define the node deployment problem:

Definition 4 (Node Deployment Problem)**.** Given a deployment cost function $C_{\mathcal{D}}$, a communication graph G, and a communication cost function $C_{\mathcal{L}}$, the node deployment problem is to find the optimal deployment
$\mathcal{D}_{OPT} = \arg\min_{\mathcal{D}} C_{\mathcal{D}}(\mathcal{D}, G, C_{\mathcal{L}})$.

In the remainder, we focus on two classes of deployment cost functions, which capture the essential aspects of the communication cost of latency-sensitive applications running in public clouds.

In High Performance Computing (HPC) applications, such as simulations, matrix computations, and graph processing [167], application nodes typically synchronize periodically using either global or local communication barriers. The completion of these barriers depends on the communication link which experiences the longest delay. Motivated by such applications, we define our first class of deployment cost function to return the highest link cost.

Class 1 (Deployment Cost: Longest Link)**.** Given a deployment plan $\mathcal{D}$, a communication graph $G = (V, E)$ and a communication cost function $C_{\mathcal{L}}$, the longest link deployment cost $C^{\mathsf{LL}}_{\mathcal{D}}(\mathcal{D}, G, C_{\mathcal{L}}) = \max_{(i,j)\in E} C_{\mathcal{L}}(\mathcal{D}(i), \mathcal{D}(j))$.

Another class of latency-sensitive applications is exemplified by search engines [10, 13] as well as web services and portals [58, 106, 116, 140]. Services provided by these applications typically organize application nodes into trees or directed acyclic graphs, and the overall latency of these services is determined by the communication *path* which takes the longest time [133]. We therefore

define our second class of deployment cost function to return the highest path cost.

Class 2 (Deployment Cost: Longest Path)**.** Given a deployment plan $\mathcal{D}$, an acyclic communication graph G and a communication cost function $C_{\mathcal{L}}$, the longest path deployment cost $C_{\mathcal{D}}^{\mathsf{LP}}(\mathcal{D}, G, C_{\mathcal{L}}) = \max_{\text{path } P \subseteq G} \left(\Sigma_{(i,j) \in P} C_{\mathcal{L}}(\mathcal{D}(i), \mathcal{D}(j)) \right)$.

Note that the above definition assumes the application is sending a sequence of causally-related messages along the edges of a path, and summation is used to aggregate the communication cost of the links in the path.

Although we believe these two classes cover a wide spectrum of latency-sensitive cloud applications, there are still important applications which do not fall exactly into either of them. For example, consider a key-value store with latency requirements on average response time or the 99.9^{th} percentile of the response time distribution [46]. This application does not exactly match either of the two deployment cost functions above, since average response time may be influenced by multiple links in different paths. We discuss the applicability of our deployment cost functions to a key-value store workload further in Section 3.6.1. We then proceed to show experimentally in Section 3.6.4 that even though Longest-Link is not a perfect match for such a workload, use of this deployment cost function still yields a 15%-31% improvement in average response time for a key-value store workload. Given the possibility of utilizing the above cost functions even if there is no exact match, CloudDiA is able to automatically improve response times of an even wider range of applications.

We prove that the node deployment problem with longest-path deployment cost is NP-hard. With longest-link deployment cost, it is also NP-hard and cannot be efficiently approximated unless P=NP (Appendix B.1). Given the

hardness results, our solution approach consists of Mixed-Integer Programming (MIP), Constraint Programming (CP) formulations and lightweight algorithmic approaches (Section 3.4). We show experimentally that our approach brings significant performance improvement to real applications. In addition, from the insight gained from the theorems above, we also show that properly rounding communication costs to *cost clusters* can be heuristically used to further boost solver performance (Section 3.6.3).

3.4 Search Techniques

In this section, we propose two encodings to solve the Longest Link Node Deployment Problem (LLNDP) using MIP and CP solvers (Sections 3.4.1 and 3.4.2), as well as one formulation for the Longest Path Node Deployment Problem (LPNDP) using a MIP solver (Section 3.4.4). In addition to solver-based solutions, we also explore alternative lightweight algorithmic approaches to both problems (Sections 3.4.3 and 3.4.5).

3.4.1 Mixed-Integer Program for LLNDP

Given a communication graph $G = (V, E)$ and a communication cost function $C_{\mathcal{L}}$ defined over any pair of instances in S, the *Longest Link Node Deployment Problem* can be formulated as the following Mixed-Integer Program (MIP):

$$
\begin{aligned}
&(MIP)\ \min\ c \\
&s.t. \sum_{i \in V} x_{ij} = 1 && \forall j \in S && (3.1)\\
&\sum_{j \in S} x_{ij} = 1 && \forall i \in V && (3.2)\\
&c \geq C_{\mathcal{L}}(j, j')(x_{ij} + x_{i'j'} - 1) && \forall (i, i') \in E, \forall j, j' \in S && (3.3)\\
&x_{ij} \in \{0, 1\} && \forall i \in V, j \in S \\
&c \geq 0
\end{aligned}
$$

In this encoding, the boolean variable x_{ij} indicates whether the application node $i \in V$ is deployed on instance $j \in S$. The constraints (3.1) and (3.2) ensure that the variables x_{ij} represent a one-to-one mapping between the set V and S. Note that the set V might need to be augmented with dummy application nodes, so that we have $|V| = |S|$. Also, the constraints (3.3) require that the value of c is at least equal to $C_{\mathcal{L}}(j, j')$ any time there is a pair (i, i') of communicating application nodes and that i and i' are deployed on j and j', respectively. Finally, the minimization in the objective function will make one of the constraints (3.3) tight, thus leading to the desired longest link value.

3.4.2 Constraint Programming for LLNDP

Whereas the previous formulation directly follows from the problem definition, our second approach exploits the relation between this problem and the subgraph isomorphism problem, as well as the clusters of communication cost values. The algorithm proceeds as follows. Given a goal c, we search for a deployment that avoids communication costs greater than c. Such a deployment exists

if and only the graph $G_c = (S, E_c)$ where $E_c = \{(i, j) : C_{\mathcal{L}}(i, j) \leq c\}$ contains a subgraph isomorphic to the communication graph $G = (V, E)$. Therefore, by finding such a subgraph, we obtain a deployment whose deployment cost c' is such that $c' \leq c$. Assume c'' is the largest communication cost strictly lower than c'. Thus, any improving deployment must have a cost of c'' or lower, and the communication graph $G = (V, E)$ must be isomorphic to a subgraph of $G_{c''} = (S, E_{c''})$ where $E_{c''} = \{(i, j) : C_{\mathcal{L}}(i, j) \leq c''\}$. We proceed iteratively until no deployment is found. Note that the number of iterations is bounded by the number of distinct cost values. Therefore, clustering similar values to reduce the number of distinct cost values would improve the computation time by lowering the number of iterations, although it approximates the actual value of the objective function. We investigate the impact of *cost clusters* in Section 3.6. For a given objective value c, the encoding of the problem might be expressed as the following Constraint Programming (CP) formulation:

$$
\begin{aligned}
&(CP)\ \texttt{alldifferent}((u_i)_{1 \leq i \leq |V|}) \\
&\quad (u_i, u_{i'}) \neq (j, j') \quad \forall (i, i') \in E, \forall j, j' \in S : C_{\mathcal{L}}(j, j') > c \\
&\quad u_i \in \{1, ..., |S|\} \qquad\qquad\qquad\qquad\qquad\qquad \forall 1 \leq i \leq |V|
\end{aligned}
$$

This encoding is substantially more compact that the MIP formulation, as the binary variables are replaced by integer variables, and the mapping is efficiently captured within the `alldifferent` constraint. In addition, at the root of the search tree, we perform an extra filtering of the domains of the x_{ij} variables that is based on compatibility between application nodes and instances. Indeed, as the objective value c decreases, the graph G_c becomes sparse, and some application nodes can no longer be mapped to some of the instance nodes. For example, a node in G needs to be mapped to a node in G_c of equal or higher

Algorithm 4: G1

Input: Instances S
Input: Communication Graph $G = (V, E)$

Find $(u_0, v_0) \in S \times S$ of lowest cost
Find an arbitrary edge $(x, y) \in E$
$\mathcal{D}(x) = u_0$, $\mathcal{D}(y) = v_0$
for $i = 1$ to $|V| - 2$ **do**
  $c_{\min} = \infty$
  for $(u, v) \in S \times S$ **do**
    if $\mathcal{D}^{-1}(v)$ is undefined and
      $\mathcal{D}^{-1}(u)$ has unmatched neighbors **then**
      if $C_{\mathcal{L}}(u, v) < c_{\min}$ **then**
        $c_{\min} = C_{\mathcal{L}}(u, v)$
        $u_{\min} = u, v_{\min} = v$
      end if
    end if
  end for
  w = one of $\mathcal{D}^{-1}(u_{\min})$'s unmatched neighbors
  $\mathcal{D}(w) = v_{\min}$
end for

degree. Similar to [161], we define a labeling based on in- and out-degree, as well as information about the labels of neighboring nodes. This labeling establishes a partial order on the nodes and expresses compatibility between them. For more details on this labeling, please refer to [161].

3.4.3 Lightweight Approaches for LLNDP

Randomization and greedy approaches can also be applied to the LLNDP. We explore each of these approaches in turn.

Randomization. The easiest approach to finding a (suboptimal) solution for

LLNDP is to generate a number of deployments randomly and select the one with the lowest deployment cost. Compared with CP or MIP solutions, generating deployments randomly explores the search space in a less intelligent way. However, since generating deployments is computationally cheaper and easier to parallelize, it is possible to explore a larger portion of the search space given the same amount of time.

Greedy Algorithms. Greedy algorithms can also be used as lightweight approaches to quickly find a (suboptimal) solution for LLNDP. We present two greedy approaches:

(G1) Recall that the deployment plan $\mathcal{D}$ is a mapping from application nodes to instances. Let $\mathcal{D}^{-1}$ be the inverse function of $\mathcal{D}$, mapping each instance $s \in S$ to an application node $n \in V$. The first greedy approach, shown in Algorithm 4, works as follows:

1. Find a link $(u_0, v_0) \in S \times S$ of lowest-cost, and for an arbitrary edge $(x, y) \in E$, let $\mathcal{D}(x) = u_0, \mathcal{D}(y) = v_0$ (Lines 1–3);
2. Find a link $(u, v) \in S \times S$ of lowest cost s.t. instance u is mapped to a node in the current partial deployment that still has unmatched neighbors, and instance v is not mapped in the current deployment (Lines 5-13);
3. Add the instance v to the partial deployment by letting $\mathcal{D}(w) = v$, where $(\mathcal{D}^{-1}(u), w)$ is one of the unmapped edges in E (Lines 14 and 15);
4. Repeat Steps 2 and 3 until all nodes are included (Lines 4–16).

This greedy approach is simple and intuitive, but it has one potential drawback: Although the links explicitly picked by the algorithm typically have low

Algorithm 5: G2

Input: Instances S
Input: Communication Graph $G = (V, E)$

Find $(u_0, v_0) \in S \times S$ of lowest cost
Find an arbitrary edge $(x, y) \in E$
$\mathcal{D}(x) = u_0$, $\mathcal{D}(y) = v_0$
for $i = 1$ to $|V| - 2$ **do**
  $c_{\min} = \infty$
  for $(u, v) \in S \times S$ **do**
    for w where $(\mathcal{D}^{-1}(u), w) \in E$ **do**
      $c_{uv} = C_{\mathcal{L}}(u, v)$
      for $(w, x) \in E$ **do**
        if $\mathcal{D}(x)$ is defined and $C_{\mathcal{L}}(v, \mathcal{D}(x)) > c_{uv}$ **then**
          $c_{uv} = C_{\mathcal{L}}(v, \mathcal{D}(x))$
        end if
      end for
      if $c_{uv} < c_{\min}$ **then**
        $c_{\min} = c_{uv}$
        $v_{\min} = v, w_{\min} = w$
      end if
    end for
  end for
  $\mathcal{D}(w_{\min}) = v_{\min}$
end for

cost, the implicit links introduced while selecting a partial solution following the lowest-cost-edge criterion can have substantial cost. This is because the mapping of a node to an instance v implies that other nodes already in the deployment are then connected to v by the corresponding underlying links. We address this issue in the refined greedy approach below.

(G2) In order to avoid selecting high-cost links implicitly when mapping a low-cost link, we revise the lowest-cost-edge criterion in Step 2 above as shown in Algorithm 5. Instead of costing a particular $(u, v) \in S \times S$ simply by the cost

of the corresponding link, we take the highest cost among the cost of (u, v) and of all links between $D(w)$ and v assuming node w is added to the current partial deployment (Lines 7–18). Intuitively, we consider not only the explicit cost of a given link that is a candidate for addition to the deployment, but also the costs of all other links which would be implicitly added to the deployment by this candidate mapping. By selecting the candidate with the minimum cost among both explicit and implicit link additions, this greedy variant locally minimizes the longest link objective at each decision point.

3.4.4 Mixed-Integer Programming for LPNDP

As previously mentioned, the Node Deployment Problem is intrinsically related to the Subgraph Isomorphism Problem (SIP). In addition, the Longest Link objective function allows us to directly prune the graph that must contain the communication graph G and therefore, can be encoded as a series of Subgraph Isomorphism Satisfaction Problems. This plays a key role in the success of the CP formulation. On the contrary, the objective function of the *Longest Path Node Deployment Problem* (LPNDP) interferes with the structure of the SIP problem and rules out sub-optimal solutions only when most of the application nodes have been assigned to instances. As a result, this optimization function barely guides the systematic search, and makes it less suitable for a CP approach. Consequently, we only provide a MIP formulation for the LPNDP.

Given a communication graph $G = (V, E)$ and a communication cost function $C_{\mathcal{L}}$ defined over any pair of instances in S, the *Longest Path Node Deployment Problem* (LPNDP) can be formulated as the following Mixed-Integer Program

(MIP):

$$
\begin{aligned}
&(MIP)\ min\ t \\
&s.t. \sum_{i \in V} x_{ij} = 1 && \forall j \in S \\
&\sum_{j \in S} x_{ij} = 1 && \forall i \in V \\
&c_{ii'} \geq C_{\mathcal{L}}(j, j')(x_{ij} + x_{i'j'} - 1) && \forall (i, i') \in E, \forall j, j' \in S \\
&t \geq t_i, t_i \geq 0 && \forall i \in V \\
&t_{i'} \geq t_i + c_{ii'} && \forall (i, i') \in E \\
&x_{ij} \in \{0, 1\} && \forall i \in V, j \in S \\
&c_{ii'} \geq 0 && \forall (i, i') \in E \\
&t \geq 0
\end{aligned}
$$

As in the previous MIP encoding, the boolean variable x_{ij} indicates whether the application node i is deployed on instance j in a one-to-one mapping. In addition, the variable $c_{ii'}$ captures the communication cost from application node i to node i' that would result from the deployment specified by the x_{ij} variables. The variable t_i represents the longest directed path in the communication graph G that reaches the application node i. Finally, the variable t appears in the objective function, and corresponds to the maximum among the t_i variables.

3.4.5 Lightweight Approaches for LPNDP

As with LLNDP, we explore both randomization and greedy approaches.

Randomization. Similarly to LLNDP, we can find a (suboptimal) solution for LPNDP by generating a number of random deployments in parallel and selecting one with the lowest deployment cost.

Greedy Heuristic Approach. Since the communication graph for LPNDP can be any directed acyclic graph containing paths of different lengths, the effect of adding a single node to a given partial deployment cannot be easily estimated. Therefore, the greedy algorithms described in Section 3.4.3 cannot be directly extended to LPNDP. However, given a LPNDP with communication graph G, we can still solve LLNDP with G greedily and use the resulting mapping as a heuristic solution for LPNDP. We experimentally study the effectiveness of these lightweight approaches in Section 3.6.5.

3.5 Measuring Network Distance

Making wise deployment decisions to optimize performance for latency-sensitive applications requires knowledge of pairwise communication cost. A natural way to characterize the communication cost is to directly measure round-trip latencies for all instance pairs. To ensure such latencies are a good estimate of communication cost during application execution, two aspects need to be handled. First, the size of the messages being exchanged during application execution is usually non-zero. Therefore, rather than measuring pure round-trip latencies with no data content included, we measure TCP round-trip time

of small messages, where message size depends on the actual application workload. Second, during the application execution, multiple messages are typically being sent and received at the same time. Such temporal correlation affects network performance, especially end-to-end latency. But the exact interference patterns heavily depend on low-level implementation details of applications, and it is impractical to require such detailed information from the tenants. Instead, we focus on estimating the quality of links without interference, as this already gives us guidance on which links are certain to negatively affect actual executions.

Although we observe mean latency heterogeneity is stable in the cloud (Figure 1.2), experimental studies have demonstrated that clouds suffer from high latency jitter [126, 142, 167]. Therefore, to estimate mean latency accurately, multiple round-trip latency measurements have to be obtained for each pair of instances. Since the number of instance pairs is quadratic in the number of instances, such measurement takes substantial time. On the one hand, we want to run mean-latency measurements as fast as possible to minimize the overhead of using ClouDiA. On the other hand, we need to avoid introducing uncontrolled measurement artifacts that may affect the quality of our results. We propose three possible approaches for organizing pairwise mean latency measurements in the following.

1. Token Passing. In this first approach, a unique token is passed between instances. When an instance i receives this token, it selects another instance j and sends out a probe message of given size. Once the entire probe message has been received by j, it replies to i with a message of the same size. Upon receiving the entire reply message, i records the round-trip time and passes the

token on to another instance chosen at random or using a predefined order. By having such a unique token, we ensure that only one message is being transferred at any given time, including the message for token passing itself. We repeat this token passing process a sufficiently large number of times, so multiple round-trip measurements can be collected for each link. We then aggregate these measurements into mean latencies per link.

This approach achieves the goal of obtaining pairwise mean-latency measurements without correlations between links. However, the lack of parallelism restricts its scalability.

2. Uncoordinated. To improve scalability, we would like to avoid excessive coordination among instances, so that they can execute measurements in parallel. We introduce parallelism by the following simple scheme: Each instance picks a destination at random and sends out a probe message. Meanwhile, all instances monitor incoming messages and send reply messages once an entire probe message has been received. After one such round-trip measurement, each instance picks another probe destination and starts over. The process is repeated until we have collected enough round-trip measurements for every link. We then aggregate these measurements into mean latencies per link.

Given n instances, this approach allows up to n messages to be in flight at any given time. Therefore, this approach provides better scalability than the first approach. However, since probe destinations are chosen at random without coordination, it is possible that: 1) one instance needs to send out reply messages while it is sending out a probe message; or 2) multiple probe messages are sent to the same destination from different sources. Such cross-link correlations are undesirable for our measurement goal.

3. Staged. To prevent cross-link correlations while preserving scalability, coordination is required when choosing probe destinations. We add an extra *coordinator* instance and divide the entire measurement process into *stages*. To start a stage for n instances in parallel, the coordinator first picks $\lfloor \frac{n}{2} \rfloor$ pairs of instances $\{(i_1, j_1), (i_2, j_2), ..., (i_{\lfloor \frac{n}{2} \rfloor}, j_{\lfloor \frac{n}{2} \rfloor})\}$ such that $\forall p, q \in \{1..n\}, i_p \neq j_q$ and $i_p \neq i_q$ if $p \neq q$. The coordinator then notifies each $i_p, p \in \{1, .., \lfloor \frac{n}{2} \rfloor\}$, of its corresponding j_p. After receiving a notification, i_p sends probe messages to j_p and measures round-trip latency as described above. Finally, i_p ends its stage by sending a notification back to the coordinator, and the coordinator waits for all pairs to finish before starting a new stage.

This approach allows up to $\frac{n}{2}$ messages between instances in flight at any time at the cost of having a central coordinator. We minimize the cost of per-stage coordination by consecutively measuring round-trip times between the same given pair of instances K_s times within the same stage, where K_s is a parameter. With this optimization, the staged approach can potentially provide scalability comparable to the *uncoordinated* approach. At the same time, by careful implementation, we can guarantee that each instance is always in one of the following three states: 1) sending to one other instance; 2) receiving from one other instance; or 3) idle in networking. This guarantee provides independence among pairwise link measurements similar to that achieved by *token passing*.

Approximations. Even the Staged network latency benchmark above can take non-negligible time to generate mean latency estimates for a large number of instances. Given that our goal is simply to estimate link costs for our solvers, we have experimented with other simple network metrics, such as hop count and IP distance, as proxies for round-trip latency. Surprisingly, these metrics

did *not* turn out to correlate well with round-trip latency. We provide details on these negative results in Appendix B.2.

3.6 Experimental Results

In this section, we present experiments demonstrating the effectiveness of ClouDiA. We begin with a description of the several representative workloads used in our experiments (Section 3.6.1). We then present micro-benchmark results for the network measurement tools and the solver techniques of ClouDiA (Sections 3.6.2 and 3.6.3). Next, we experimentally demonstrate the performance improvements achievable in public clouds by using ClouDiA as the deployment advisor (Section 3.6.4). Finally, we show the effectiveness of lightweight algorithmic approaches compared with solver-based solutions (Section 3.6.5).

3.6.1 Workloads

To benchmark the performance of ClouDiA, we implement three different workloads: a behavioral simulation workload, a query aggregation workload, and a key-value store workload. Each workload illustrates a different communication pattern.

Behavioral Simulation Workload. In behavioral simulations, collections of individuals interact with each other to form complex systems. Examples of behavioral simulations include large-scale traffic simulations and simulation of groups of animals. These simulations organize computation into ticks and

achieve parallelism by partitioning the simulated space into regions. Each region is allocated to a processor and inter-node communication is organized as a 2D or 3D mesh. As synchronization among processors happens every tick, the progress of the entire simulation is limited by the pair of nodes that take longest to synchronize. Longest-Link is thus a natural fit to the deployment cost of such applications. We implement a workload similar to the fish simulation described by Couzin et al [40]. The communication graph is a 2D mesh and the message size per link is 1 KB for each tick. To focus on network effects, we hide CPU intensive computation and study the time to complete 100K ticks over different deployments.

Synthetic Aggregation Query Workload. In a search engine or distributed text database, queries are processed by individual nodes in parallel and the results are then aggregated [13]. To prevent the aggregation node from becoming a bottleneck, a multi-level aggregation tree can be used: each node aggregates some results and forwards the partial aggregate to its parent in the tree for further aggregation. The response time of the query depends on the path from a leaf to the root that has highest total latency. Longest-Path is thus a natural fit for the deployment cost of such applications. We implement a two-level aggregation tree of a top-k query answering workload. The communication graph is a tree and the forwarding message size varies from the leaves to the root, with an average of 4 KB. We hide ranking score computation and study the response time of aggregation query results over different deployments.

Key-Value Store Workload. We also implement a distributed key-value store workload. The key-value store is queried by a set of front-end servers. Keys are randomly partitioned among the storage nodes, and each query

touches a random subset of them. The communication graph therefore is a bipartite graph between front-end servers and storage machines. However, unlike the simulation workload, the average response time of a query is not simply governed by the slowest link. To see this, consider a deployment with mostly equal-cost links, but with a single slower link of cost c, and compare this to a similar deployment with two links of cost $c - \epsilon$. If Longest-Link were used as the deployment cost function, the second deployment would be favored even though the first deployment actually has lower average response time. Indeed, neither Longest-Link nor Longest-Path is the precisely correct objective function for this workload. We evaluate the query response time over different deployments by using Longest-Link, with a hope that it can still help avoid high cost links.

3.6.2 Network Micro-Benchmarks

Setup. The network measurement tools of ClouDiA are implemented in C++ using TCP sockets (SOCK_STREAM). We set all sockets to non-blocking mode and use *select* to process concurrent flows (if any). We disable the Nagle Algorithm.

We ran experiments in the Amazon Elastic Compute Cloud (Amazon EC2). We used large instances (m1.large) in all experiments. Each large instance has 7.5 GB memory and 4 EC2 Compute Units. Each EC2 Compute Unit provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor [4]. Unless otherwise stated, we use 100 large instances for network micro-benchmarks, all allocated by a single *ec2-run-instance* command, and set

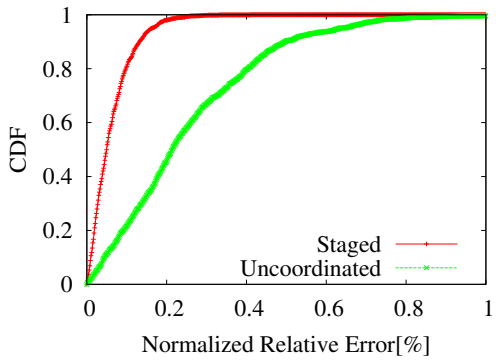


Figure 3.2: Staged vs Uncoordinated (against Token Passing)

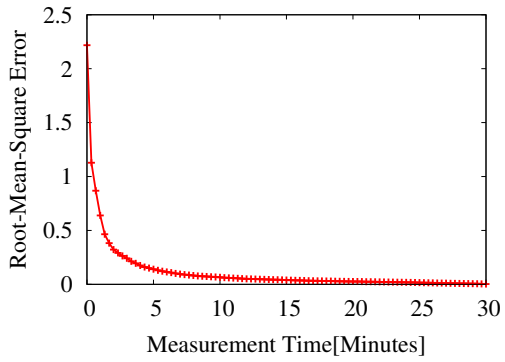


Figure 3.3: Latency Measurement Convergence over Time

the round-trip message size to 1KB. We show the following results from the same allocation so that they are comparable. Similar results are obtained in other allocations.

Round-Trip Latency Measurement. We run latency measurements with each of the three approaches proposed in Section 3.5 and compare their accuracy in Figure 3.2. To make sure *token passing* can observe each link a sufficiently large number of times, we use 50 large instances in this experiment. We consider the mean latencies for 50^2 instance pairs as a 50^2-dimension vector of mean latencies, of which each dimension represents one link. Results are first normalized to the unit vector. Then, *staged* and *uncoordinated* are compared with the baseline *token passing*. The CDF of the relative error of each dimension is shown in Figure 3.2. Using *staged*, we find 90% of links have less than 10% relative error and the maximum error is less than 30%; whereas using *uncoordinated* we find 10% of links have more than 50% relative error. Therefore, as expected, *staged* exhibits higher measurement accuracy than *uncoordinated*.

Figure 3.3 shows convergence over time using the *staged* approach with 100 instances and K_s = 10. Again, the measurement result is considered as a latency

vector. The result of the full 30 minutes observation is used as the *ground truth*. Each stage on average takes 2.75 ms. Therefore, we obtain about 3004 measurements for each instance pair within 30 minutes. We then calculate the root-mean-square error of partial observations between 1 and 30 minutes compared with the ground truth. From Figure 3.3, we observe the root-mean-square error drops quickly within the first 5 minutes and smooths out afterwards. Therefore, we pick 5 minutes as the measurement time for all the following experiments with 100 instances. For experiments with $n \neq 100$ instances, since the *staged* approach tests $\frac{n}{2}$ pairs in parallel whereas there are $O(n^2)$ total pairs, measurement time needs to be adjusted linearly to $5 \cdot \frac{n}{100}$ minutes.

3.6.3 Solver Micro-Benchmarks

Setup. We solve the MIP formulation using the IBM ILOG CPLEX Optimizer, while every iteration of the CP formulation is performed using IBM ILOG CP Optimizer. Solvers are executed on a local machine with 12 GB memory and Intel Core i7-2600 CPU (4 physical cores with hyper-threading). We enable parallel mode to allow both solvers to fully utilize all CPU resources. We use k-means to cluster link costs. Since the link costs are in one dimension, such k-means can be optimally solved in $O(kN)$ time using dynamic programming, where N is the number of distinct values for clustering and k is the number of *cost clusters*. After running k-means clustering, all costs are modified to the mean of the containing cluster and then passed to the solver. For comparison purposes, the same set of network latency measurements as in Section 3.6.2 is used for all solver micro-benchmarks. In each solver micro-benchmark, we set the number of application nodes to be 90% of the number of allocated instances. To find an

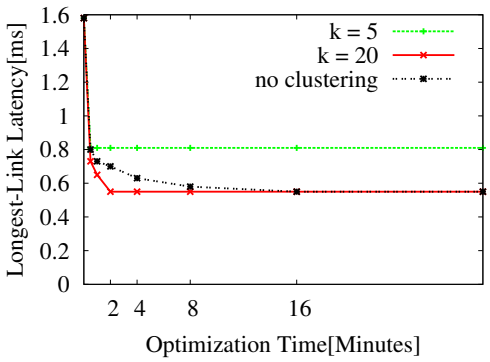


Figure 3.4: CP Convergence with k-means: Longest Link

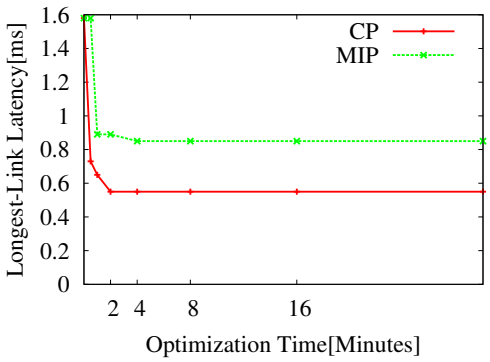


Figure 3.5: CP vs MIP Convergence: Longest Link (k=20)

initial solution to bootstrap the solver's search, we randomly generate 10 node deployment plans and pick the best one among those.

Longest-Link Node Deployment Problem. We provide both a MIP formulation and a CP formulation for LLNDP. Since the parameter of cost clustering may have different impacts on the performance of the two solvers, we first analyze the effect of cost clustering. Figure 3.4 shows the convergence of the CP formulation for 100 instances with different numbers of clusters. The communication graph for Figure 3.4 to Figure 3.6 is a 2D mesh from the simulation workload and the deployment cost is the cost of the longest link. We tested all possible k values from 5 to the number of distinct values (rounded to nearest 0.01 ms), with an increment of 5. We present three representative configurations: $k = 5$, $k = 20$ and *no clustering*. As we decrease the number of clusters, the CP approach converges faster. Indeed, with no clustering, the best solution is found after 16 minutes, whereas it takes 2 minutes and 30 seconds for the CP approach to converge with $k = 20$ and $k = 5$, respectively. This is mainly due to the fact that fewer iterations are needed to reach the optimal value. Such a difference demonstrates the effectiveness of cost clustering in reducing the search

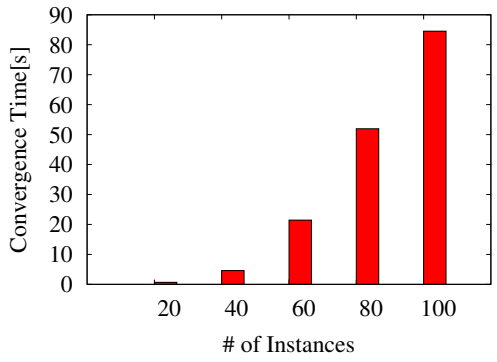


Figure 3.6: CP Solver Scalability: Longest Link

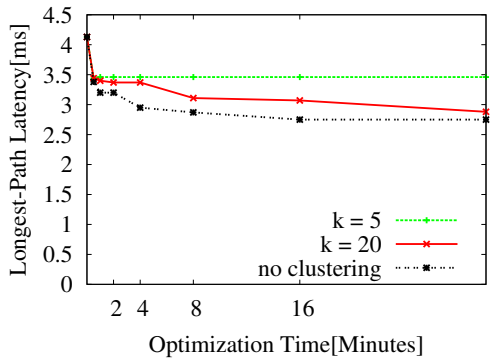


Figure 3.7: MIP Convergence with k-means: Longest Path

time. On the other hand, the smaller the value of k is, the coarser the cost clusters are. As a result, the CP model cannot discriminate among deployments within the same cost cluster, and this might lead to sub-optimal solutions. As shown in Figure 3.4, the solver cannot find a solution with a deployment cost smaller than 0.81 for $k = 5$, while both cases $k = 20$ and *no clustering* lead to a much better deployment cost of 0.55.

Figure 3.5 shows the comparison between the CP and the MIP formulations with $k = 20$. MIP performs poorly with the scale of 100 instances. Also, other clustering configurations do not improve the performance of MIP. One reason is that for LLNDP, the encoding of the MIP is much less compact than CP. Moreover, the MIP formulation suffers from a weak linear relaxation, as x_{ij} and $x_{i'j'}$ should add up to more than one for the relaxed constraint 3.3 to take effect.

Given the above analysis, we pick CP with $k = 20$ for the following scalability experiments as well as LLNDP in Sections 3.6.4 and 3.6.5.

Measuring scalability of a solver such as ours is challenging, as problem size does not necessarily correlate with problem hardness. To observe scalability

behavior with problem size, we generate multiple inputs for each size and measure the average convergence time of the solver over all inputs. The multiple inputs for each size are obtained by randomly choosing 50 subsets of instances out of our initial 100-instance allocation. The convergence time corresponds to the time the solver takes to not be able to improve upon the best found solution within one hour of search. Figure 3.6 shows the scalability of the solver with the CP formulation. We observe that average convergence time increases acceptably with the problem size. At the same time, at every size, the solver is able to devise node deployment plans with similar average deployment cost improvement ratios.

Longest-Path Node Deployment Problem. Figure 3.7 shows the convergence of the MIP formulation for 50 instances with different number of link cost clusters. The communication graph is an aggregation tree with depth less than or equal to 4. Similar to Figure 3.4, the solver performs poorly under the configuration of $k = 5$. Interestingly, clustering costs does not improve performance for LPNDP. This is because the costs are aggregated using summation over the path for LPNDP, and therefore the solver cannot take advantage of having fewer distinct values. We therefore use MIP with *no clustering* for LPNDP in Sections 3.6.4 and 3.6.5.

3.6.4 ClouDiA Effectiveness

Setup. We evaluate the overall ClouDiA system in EC2 with 100 to 150 instances over different allocations. Other settings are the same as in Section 3.6.2 (network measurement) and Section 3.6.3 (solver). We use a 10% over-allocation

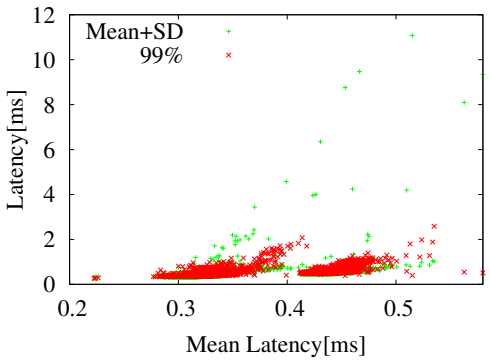


Figure 3.8: Correlation between Different Cost Metrics

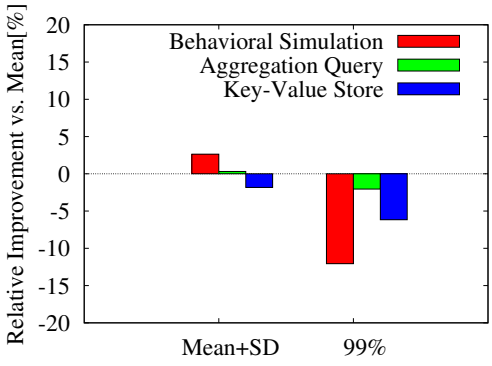


Figure 3.9: Performance Difference under Different Cost Metrics

ratio in all experiments except the last one (Figure 3.10), in which we vary this ratio. The deployment decision made by ClouDiA is compared with the default deployment, which uses the instance ordering returned by the EC2 allocation command. Note that EC2 does not offer us any control on how to place instances, so the over-allocated instances we obtain are just the ones returned by the *ec2-run-instance* command.

Cost Metrics. In Figure 3.8, we study the correlation between three communication cost metrics under one representative allocation of 110 instances. Each point represents the link between a pair of nodes: The x axis shows its mean latency (Mean) and the y axis shows its mean latency plus standard deviation (Mean+SD) or 99th percentile latency (99%). While links with larger mean latencies tend to have larger Mean+SD or 99% values, they are not perfectly correlated. This result motivates us to study the actual application performance under deployments generated by ClouDiA with different cost metrics. Figure 3.9 shows the relative improvement of using Mean+SD or 99% compared with using Mean. Using 99th percentile latency reduces actual performance for all three applications, suggesting that the performance of these applications is not well-

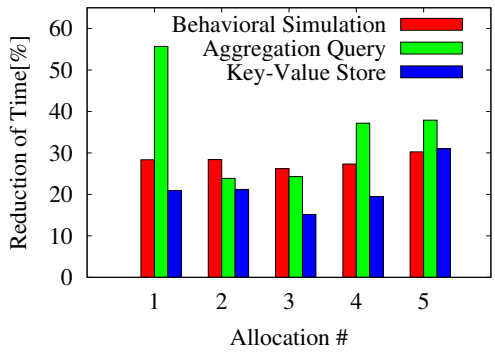


Figure 3.10: Overall Improvement in EC2

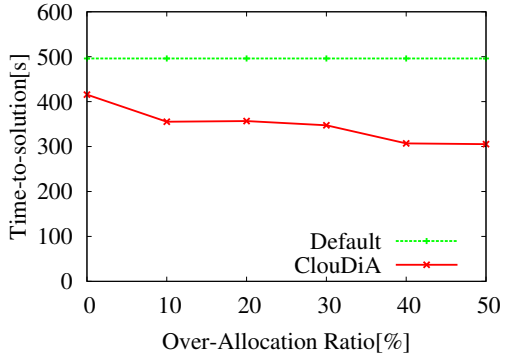


Figure 3.11: Effect of Over-Allocation in EC2: Behavioral Simulation

captured solely by this metric. While using Mean+SD improves performance for behavioral simulation and aggregation query workloads, it reduces performance for the key-value store workload. However, the observed differences in performance with respect to using Mean are not dramatic. These results suggest that although different applications favor different cost metrics, both Mean and Mean+SD are reasonable metrics to use under these workloads.

Overall Effectiveness. We show the overall percentage of improvement over 5 different allocations in EC2 in Figure 3.10. The behavioral simulation and key-value store workloads use 100 application nodes, whereas the aggregation query workload uses 50 nodes. For the simulation workload, we report the reduction in time-to-solution. For the aggregation query and key-value store workloads, we report the reduction in response time. Each of these is averaged based on an at least 10 minutes of observation for both. We compare the performance of ClouDiA optimized deployment to the default deployment. ClouDiA achieves 15% to 55% reduction in time-to-solution or response time over 5 allocations for three workloads. The reduction ratio varies for different allocations. Among three workloads, we observe the largest reduction ratio on aver-

age in the aggregation query workload, while the key-value store workload gets less improvement than the others. There are two reasons for this effect. First, the communication graph of the aggregation query workload has the fewest edges, which increases the probability that the solver can find a high quality deployment. Second, the Longest-Link deployment cost function does not exactly match the mean response time measurement of the key-value store workload, and therefore deployment decisions are made less accurately.

Effect of Over-allocation. In Figure 3.11, we study the benefit of over-allocating instances for increasing the probability of finding a good deployment plan. Note that although ClouDiA terminates extra instances once the deployment plan is determined, these instances will still be charged for at least one hour usage due to the round-up pricing model used by major cloud service providers [4, 148, 113]. Therefore, a trade-off must be made between performance and initial allocation cost. In this experiment, we use an application workload similar to Figure 3.10, but with 150 EC2 instances allocated at once by a single *ec2-run-instance command*. To study the case with over-allocation ratio x, we use the first $(1 + x) \cdot 100$ instances out of the 150 instances by the EC2 default ordering. Figure 3.11 shows the improvement in time-to-solution for the simulation workload. The default deployment always uses the first 100 instances, whereas ClouDiA searches deployment plans with the $100x$ extra instances. We report 38% performance improvement with 50% extra instances over-allocated. Without any over allocation, 16% improvement is already achieved by finding a good injection of application nodes to instances. Interestingly, with only 10% instance over-allocation, 28% improvement is achieved. Similar observations are found on other allocations as well.

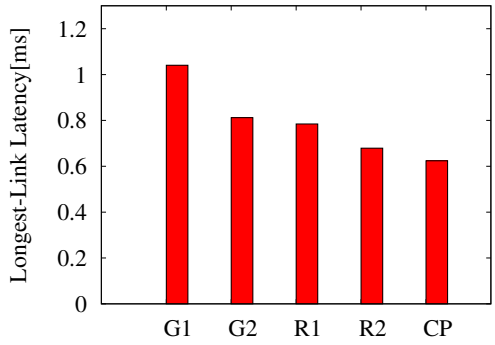


Figure 3.12: Lightweight Approaches: Longest Link

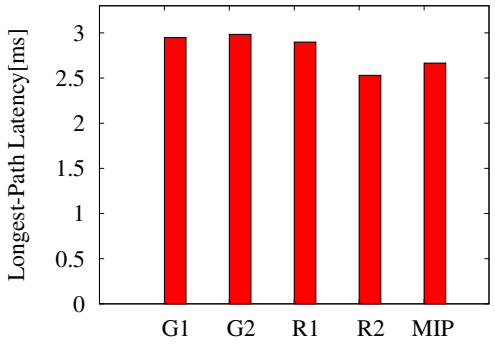


Figure 3.13: Lightweight Approaches: Longest Path

3.6.5 Lightweight Approaches

Setup. In Figures 3.12 and 3.13, we compare the effectiveness of lightweight approaches against the CP and MIP formulations. Results are averaged over 20 different allocations of 50 instances with 10% over-allocation. G1 is the simple greedy algorithm, which adds a node following a lowest-cost-edge criterion at each step. G2 is the refined greedy algorithm, which iteratively adds a node such that partial deployment cost is minimal after addition. R1 is the lowest deployment cost obtained by generating 1000 random deployments. R2 is the lowest deployment cost obtained by generating random deployments in parallel using the same amount of wall-clock time as well as the same hardware given to the CP or MIP solvers. The solver setup and hardware configuration are the same as in Section 3.6.3.

Longest-Link Node Deployment Problem. In Figure 3.12, both CP and R2 run for two minutes and all other methods finish in less than one second. G1 provides the worst solution overall, with a cost 66.7% higher than CP. We examine the top-5 implicit links that are not explicitly chosen by G1, but are nevertheless

included in the final solution. These links are on average 31.6% more expensive than the worst link picked by CP. G2 improves G1 significantly by taking implicit links into consideration during each step of solution expansion. Interestingly, R1 is able to generate deployments with average cost 3.39% lower than G2. R2 is able to generate deployments with cost only 8.65% higher than CP. Such results suggest that simply generating a large number of random deployments and picking the best one can provide reasonable effectiveness with minimal development overhead.

Longest-Path Node Deployment Problem. In Figure 3.13, both MIP and R2 run for 15 minutes and all other methods finish in less than one second. Although G1 and G2 are designed for LLNDP, they are still able to generate deployments with cost comparable to R1. Surprisingly, R2 is able to find deployments with cost on average 5.10% lower than MIP. We conjecture that even though the MIP solver can exploit the search space in a much more intelligent way than R2, the distribution of good solutions in this particular problem makes such intelligent searching less important. Meanwhile, R2's simplicity and efficiency enable it to explore a larger portion of the search space than MIP within the same amount of time.

To verify the effectiveness of R2, we also ran an additional experiment where the total number of instances was decreased to 15. In this scenario, MIP was always able to find optimal solutions within 15 minutes over 20 different allocations. Meanwhile, R2 found suboptimal solutions for 40% of the allocations given the same amount of time as MIP. So we argue that MIP is still a complete solution which guarantees optimality when the search finishes. R2, however, cannot provide any guarantee, even when the search space is small.

3.7 Conclusions

We have shown how ClouDiA makes intelligent deployment decisions for latency-sensitive applications under heterogeneous latencies, which naturally occur in public clouds. We formulated the deployment of applications into public clouds as optimization problems and proposed techniques to speed up the search for high-quality deployment plans. We also presented how to efficiently obtain latency measurements without interference. Finally, we evaluated ClouDiA in Amazon EC2 with realistic workloads. ClouDiA is able to reduce the time-to-solution or response time of latency-sensitive applications by 15% to 55%, without any changes to application code.

As future work, we plan to extend our formulation to support weighted communication graphs. Another direction of practical importance is quantifying over-allocation cost and analyzing its impact on total cost-to-solution for scientific applications. Finally, we will investigate the deployment problem under other criteria, such as bandwidth, for additional classes of cloud applications.

CHAPTER 4

CONTROLLING RESPONSE TIME FOR INTERACTIVE DATA ANALYTICS

4.1 Introduction

As we have discussed in Section 1.1.2, the flexible resource management that the cloud offers matches the workloads of interactive data analytics. However, current data analytics engines typically do not support elastic resource management. Instead, they deal with a predefined amount of resources, regardless of changes in the number of users, the size of the active datasets, and the computational demand of queries. When multiple users share the same engine, the resource manager typically maintains weighted fairness between users, or guarantees each user a minimum resource share. The amount of resources each user can consume depends on the activity of other users, resulting in unpredictable speed and possible loss of interactivity. Also, the resource managers are not aware of the flexibility and diversity of resource allocation in the cloud; users have to allocate resources manually before starting the engine, which can be both unintuitive and suboptimal.

This chapter is a first step towards building a cloud-aware elastic resource manager for interactive data analysis. Beyond maintaining the active part of the dataset in distributed memory cost effectively, our approach gives users control over the speed of the queries by allowing them to tune the amount of CPU resources allocated together with memory.

An interactive data analysis workload has two special characteristics that

can be exploited when multiple users share the same engine. One is the existence of think time between tasks issued by the same user [147, 32, 27], which implies that over-committing resources can be beneficial when a slightly relaxed query performance guarantee is acceptable. The second characteristic is that different users may work on overlapping datasets. In this case, the overlapping part needs to be stored only once and additional savings can be achieved.

Contributions of this Chapter. In this chapter, we make the following contributions:

1) We define a new resource allocation workflow for users to run interactive data analysis in the cloud. This workflow does not only simplify the existing manual resource allocation workflow, but also enables users to dynamically control the interactivity, or *speed*, of their interactive data analysis queries in the cloud. We introduce Smart, an elastic resource manager that supports this new resource allocation workflow, and transparently optimizes cost efficiency.

2) We propose SmartShare, a new mechanism for sharing resources among users running interactive data analysis. SmartShare provides exactly the same query performance guarantees as allocating isolated instances of Smart, but achieves significant cost savings.

3) To take advantage of think time and overlapping datasets we extend SmartShare to SmartShare+. SmartShare+ further improves cost-effectiveness, possibly at a slight sacrifice of isolation between users.

4) We have implemented speed control using Smart, SmartShare, and SmartShare+. Through an extensive experimental analysis we show that significant improvements in cost reduction, session start time, and speed change

reaction time can be achieved with our techniques.

4.2 Overview

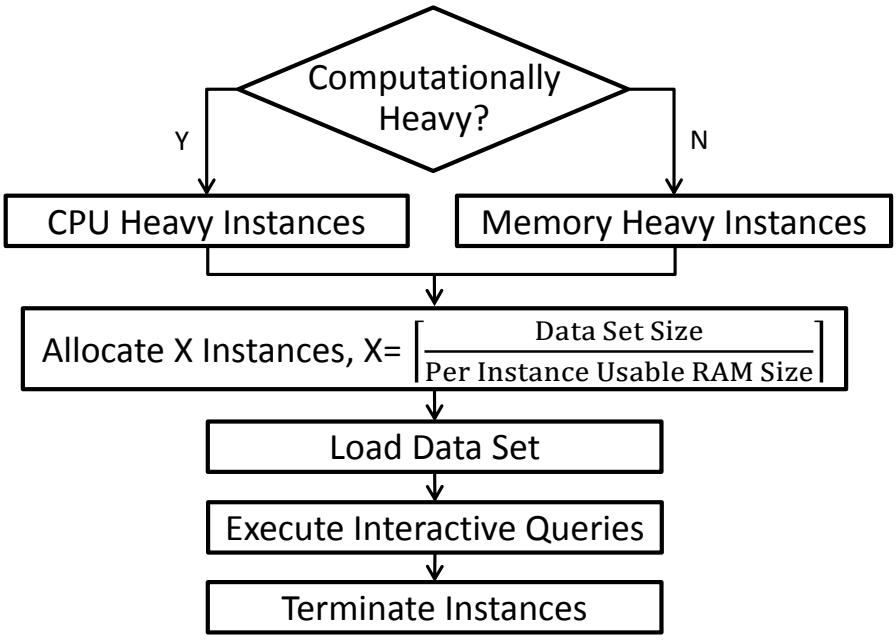


Figure 4.1: A Typical Resource Allocation Workflow

Consider data scientists (users) who want to run queries interactively against a large dataset in public or private cloud. Existing interactive data analytics engines require users to pre-allocate a cluster of cloud instances. As shown in Figure 4.1, users have to decide the type of instances (CPU-heavy or memory-heavy) before running any queries, based on an estimate of computational heaviness for future queries and the desired level of interactivity. Then they allocate enough instances to hold the entire dataset in RAM, load the dataset and run interactive queries against it. Finally, after finishing all the queries, they deallocate resources.

We observe two drawbacks to this workflow. First, it is overly restrictive to force users to estimate the computational heaviness of queries up front, without knowing the data characteristics. After observing the results of the first few queries, a user may wish to try another set of queries with different computational heaviness than originally planned. Second, although for simplicity we show only two instance types in Figure 4.1, most cloud platforms provide tens of different types of instances [4, 62, 148]. It is nontrivial for a user to find the instance type best suited for a particular workload.

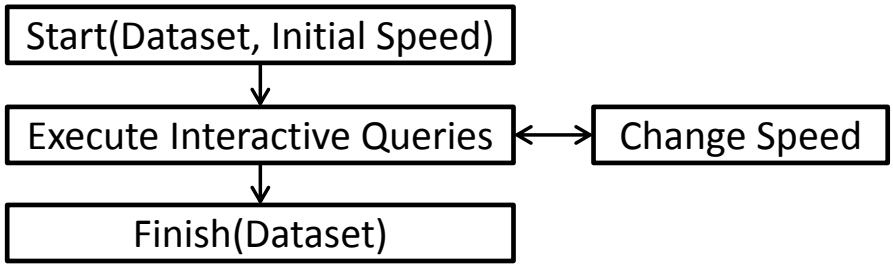


Figure 4.2: A Simplified Resource Allocation Workflow

Figure 4.2 shows a revised resource allocation workflow addressing these issues. In the revised workflow, users do not need to pick an instance type. Instead, they identify the dataset they will be processing and a desired initial *speed*. The speed is defined as the ratio between compute power and in-memory data size and can be changed on the fly during *interactive sessions*. By adjusting speed according to the computational heaviness of queries, users can control the level of interactivity. We formally define speed, interactive sessions and other concepts and discuss their implications in Section 4.3. In Section 4.4, we introduce Smart, an elastic resource manager that supports this revised resource allocation workflow for interactive data analysis. Smart automatically allocates a cluster of instances best suited to the interactive workload in the cloud, and

adjusts the cluster size, instance type, and data placement in response to users' speed change requests.

When multiple users want to run interactive queries at the same time, we could run multiple copies of Smart, each managing a cluster of instances for a single user. Alternatively, we could share instances among users. In Section 4.5, we describe SmartShare, a model for sharing resources among users running interactive data analysis, which significantly improves resource utilization without weakening query performance guarantees to individual users. The better resource utilization is achieved mainly by collocating users with complementary requirements. SmartShare exposes the same resource allocation workflow as shown in Figure 4.2, and thus is transparent to users. We also propose a billing model that ensures that all users enjoy the same cost saving ratio compared with running interactive sessions in Smart.

Recall the special workload characteristics of interactive data analysis: think time between queries and overlapping datasets. In Section 4.6 we describe SmartShare+, which exploits these characteristics to further reduce cost. SmartShare+ can over-commit resources to users, reducing resource idle time while maintaining only slightly relaxed query performance guarantees. It can also save memory usage when users work on overlapping datasets, without weakening query performance guarantees.

4.3 Problem Formulation

In this section, we formally define the problem setting and describe the assumptions we make.

Instances. We assume users run their interactive data analytic engine in the cloud. Virtual machine *instances* can be allocated in the cloud as needed, and the total cost is charged per instance per unit billing time period.

Instance Types. Commercial Cloud services, such as Amazon EC2, Google Computer Engine, and Windows Azure, all provide multiple types of instances. Different instance types provide different amounts of computation power, memory, network bandwidth and local storage, with different pricing. We assume instance type cannot be changed after allocation.

Cloud Storage. We also assume the data to be analyzed is stored in durable *cloud storage*, accessible from all instances within the cloud. After instance allocation, data is copied from cloud storage into the memory of instances. Examples of cloud storage include Amazon Simple Storage Service (S3), Google Cloud Storage and Windows Azure Storage. These cloud storage systems are typically highly scalable blob stores. In our experience, Amazon S3 provides enough read throughput to fully utilize the aggregated network bandwidth of up to 100 m1.xlarge instances. Therefore, we assume the cloud storage itself is not a performance bottleneck.

Interactive Speed. In the workflow of Figure 4.2, users can specify initial speed and dynamically change it between interactive queries. We formally define the speed as follows:

$$\text{Interactive Speed} = \frac{\text{Local Compute Units}}{\text{GB of Data}} \tag{4.1}$$

In this definition, a compute unit is a unit of processing power. This is typically defined by cloud providers to normalize the performance of different CPU types

("EC2 Compute Unit" for Amazon EC2, and "virtual core" for Google Compute Engine and Windows Azure). Such compute units must be collocated with the in-memory data to be effective for interactive query processing, as moving data across the network and processing remotely is orders of magnitude slower than processing it locally.

For a given query and dataset, it is possible to use workload history as well as other techniques from query completion time prediction [91] to estimate the interactive speed parameter needed to achieve a desired completion time. Allowing users to specify a desired completion time and automatically adjusting interactive speed before running each query might seem attractive. However, adjusting interactive speed is an expensive operation and can take longer than the query completion time itself, making such adjustment ineffective. Thus, we do not support users directly specifying the expected completion time of their queries. Instead, they exercise control of query completion time by increasing or decreasing the interactive speed parameter.

Interactive Sessions. To make data movement overhead explicit, we organize users' interactive queries into *interactive sessions*. An interactive session is a sequence of operations issued by the same user on some predeclared datasets. As shown in Figure 4.2, users start interactive sessions by specifying speed and datasets. Then users query the datasets interactively using different operators, optionally changing speed between queries. Finally, users finish the interactive session, releasing resources. Within an interactive session, users can process only the specified datasets, without shuffling the data. Multiple interactive sessions can be started when data shuffling is necessary, with shuffling results materialized to cloud storage between the sessions. By default, datasets are *par-*

titioned datasets, which are partitioned across instances with possibly different sizes per partition. Users can specify datasets to be *replicated datasets*, which are replicated in *every* instance in which part of any partitioned dataset is stored. An optional hash function ID = $Hash$(Key) can be provided by users for data collocation. Data with the same hash ID is guarantee to be collocated at the same instance.

We assume the input datasets remain unchanged during an interactive session. Updates to query states, such as partial aggregation results, can be stored in limited temporary memory space of each interactive session. Updates to the datasets can be reflected upon starting a new interactive session.

Interactive Operators. Analytical SQL queries are one key type of operations that can be executed during an interactive session. In this case the replicated datasets are the dimension tables, the partitioned datasets are the fact tables, and the hash function can be used to define the co-partitioning columns between multiple fact tables. Within an interactive session, the user can perform selection, projection, aggregation and local joins. Since non-local joins between fact tables result in costly shuffling, it is impractical to guarantee interactivity due to their excessive use of network resources. Users are required to explicitly start separate sessions with different collocation hash functions to run arbitrary joins between multiple fact tables. Beyond SQL, parallel machine learning algorithms that do not rely on shuffling are also supported, such as random forest, regression analysis, and k-means clustering.

In principle, any operator that is data-parallel and does not require shuffling can be implemented as an interactive operator and used in interactive sessions.

Interactivity SLA. With a *strict* SLA of speed = x, we guarantee that for each GB of in-memory data, x compute units are available locally to the data at any time during the interactive sessions.

With a *relaxed* SLA of speed = x, we guarantee that for each GB of in-memory data, x compute units are available locally to the data within on average (or some specified percent of the time) y seconds of delay after a user issues an operation and remain available until the operation finishes.

Master Node. We use a master node to manage the instance allocation and data placement in the virtual machine cluster. User queries are dispatched to the appropriate instances by the master node. The master node also detects instance failures using heartbeat messages. After an instance failure, we can always reconstruct the lost data from cloud storage, where data is stored persistently. More sophisticated failure recovery could be provided, e.g. by adopting ZooKeeper [73], but is orthogonal to this dissertation.

4.4 Smart

In this section, we describe Smart, an elastic resource manager that implements a strict interactive SLA and simplifies the resource allocation workflow for users running interactive data analysis in the cloud, as shown in Figure 4.2. In addition to managing traditional types of resources, such CPU and memory, Smart also manages the placement and movement of the actual data to guarantee fast query response time. Smart provides transparent optimization of data loading

speed and cost efficiency. Smart is designed to manage resources for a single user within a single interactive session.

4.4.1 Instance Management

In this subsection, we discuss the following two problems: 1) when to allocate or terminate instances; 2) which type of instance to allocate.

Given a speed s, ideally we would like always to use the type of instance with minimum cost per unit memory that can meet the local compute unit requirement. We define a function BESTTYPE(s) to return the best instance type for a given speed as follows. Let m_t be the usable memory size in GB of an instance of type t, c_t be the number of compute units of t and $\texttt{cost}_t$ be the price for t per unit billing time period. Then

$$\text{BESTTYPE}(s) = \arg\min_t \frac{\texttt{cost}_t}{\min(m_t, c_t/s)} \tag{4.2}$$

where $\min(m_t, c_t/s)$ is the available memory size of a type t instance, under speed s. Intuitively, BESTTYPE finds the instance type that achieves lowest cost per GB of memory, while providing enough local compute units to fulfill a given speed requirement. By purely using instances of BESTTYPE, apart from the last instance which might not be fully used, the highest cost-effectiveness can be achieved.

Each instance, after allocation and before termination, can have two labels, `active` and `optimal`. Instances that are currently used by interactive sessions are labeled `active`.

An `optimal` label on an instance indicates that the type of this instance is

the same as BESTTYPE($s_{current}$), where $s_{current}$ is the current speed. Intuitively, to achieve minimal cost, we should always use instances of label `optimal` and always terminate instances when they are not `active`. However, there are reasons for not doing so: The current billing time period of an instance may not be over yet. Depending on the provider, the billing time period can be as long as one hour [4]. No cost can be saved by early termination of suboptimal or inactive instances, and these instances may still be useful now or when another speed change occurs.

Termination. With each instance that is not both `active` and `optimal` we associate a termination time. We set the termination time to be shortly before the end of the current billing time period to allow time for the data to migrate away if the instance is still `active`. For providers whose billing time period is very short, we can enforce a pre-defined *delay time* before termination. We do not terminate instances in other circumstances.

Allocation. Instances are always allocated with type BESTTYPE($s_{current}$). Since BESTTYPE($s_{current}$) can change over time due to users' speed change requests and we keep suboptimal instances around until their termination time, instances of different type can be `active` at the same time. New instances are allocated in response to the following three events: 1) the start of an interactive session; 2) a speed increase is requested and current instances are insufficient to serve the new speed; and 3) an `active` instance that is not `optimal` approaches its termination time. We discuss the data movement required for instance allocation and speed changes in the following two subsections.

By following the above instance allocation and termination protocols, we ensure the following property:

Property 3 (Convergence)**.** *If a user maintains steady speed for at least one billing time period plus delay time, all running instances are of type* BESTTYPE($s_{current}$)*.*

This is easy to see, since we keep suboptimal instances around only until the end of their billing time period or the pre-defined delay time, after which they are replaced by instances of type BESTTYPE($s_{current}$).

4.4.2 Initial Data Loading

At the beginning of an interactive session, the datasets need to be loaded from cloud storage into the instances. The loading is done in parallel using all instances. Depending on the properties of the datasets, there are three possible cases: 1) if the datasets are partitioned without a collocation hash function, data is stored locally after loading; 2) if the datasets are partitioned with a collocation hash function, data is forwarded to the specific instance according to $Hash$(Key); and 3) if the datasets are replicated, data is stored locally and forwarded to the other instances [1].

Cloud storage can supply data as fast as instances can read. To maximize loading throughput, we deploy multiple reader threads in each instance. Simple multithreading is not enough to minimize loading time because of *stragglers*: Some data reads take significantly longer to finish than the others. To mitigate the slowdown caused by stragglers, we implement two techniques: *work sharing* and *work stealing*. In work sharing, Smart assigns a data block to each reader thread initially. When a reader thread finishes reading, it fetches the location

[1] When the replicated data is large, such forwarding can be done by pushing data linearly along a chain, as described in [59].

of another unread data block from Smart. In work stealing, Smart assigns all data blocks to reader threads initially. When a reader thread finishes reading its assigned data blocks, it contacts other threads, possibly in other instances, to "steal" work from them. Theoretically, work stealing involves fewer control message exchanges than work sharing [22]. We experimentally compare the effectiveness of these two techniques in Section 4.7.2.

4.4.3 Data Rebalancing for Speed Changes

Speed change requests from users are processed in the background. A user can run interactive queries immediately after a speed change; the requests run at the old speed until the concurrent speed change has completed.

For a speed change, we first adjust the cluster size based on the discussion in Section 4.4.1. Then we rebalance the data to fit the new speed. In this subsection, we discuss how the rebalancing is done through an example. To simplify the presentation, only partitioned datasets are used in the example. Other cases can be solved similarly.

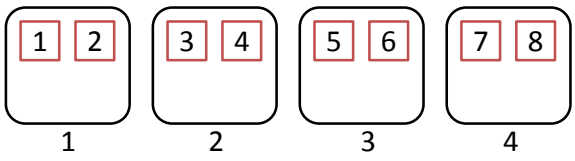


Figure 4.3: Initial Speed = 1

Suppose a user starts a session with a partitioned dataset of 8GB and initial speed 1 CU/GB, where CU stands for compute units. Assume BESTTYPE(1) returns an instance type with <2CU, 4GB> per instance. Although each instance

has 4GB of memory, only 2GB can be used since each instance has only 2CU and the speed is 1CU/GB. Thus, we need to allocate four instances of this type. Data is loaded to the four instances evenly, as shown in Figure 4.3. Each solid square is a data chunk of 1GB. All the instances now are both `active` and `optimal`.

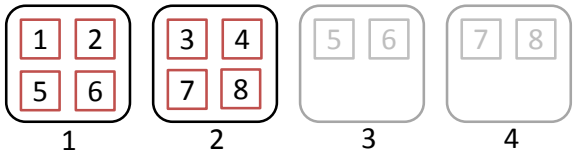


Figure 4.4: Decreased Speed = 0.5

Then, the user reduces the speed to 0.5 CU/GB and BESTTYPE(0.5) = BESTTYPE(1). Now each instance can serve 4GB of data, so two instances are enough to fulfill the required speed. Data chunks 5 to 8 are migrated into instances 1 and 2 in the background. When data migration finishes, the label `active` is removed from instances 3 and 4, and they turn gray in Figure 4.4. We do not terminate instances 3 and 4 immediately, because they have not reached their termination times. Copies of data chunks 5 to 8 remain in instances 3 and 4.

Before instances 3 and 4 reach their termination times, the user increases the speed to 2 CU/GB and BESTTYPE(2) returns a new type of instance with <4CU, 2GB> per instance. To perform the speed increase, we first reactivate instances 3 and 4. Since each of instances 1-4 has only enough compute units to serve one data chunk, we need to allocate two new instances, instance 5 and 6, with <4CU, 2GB> per instance to serve the other four data chunks.

Ideally, we want to rebalance the data between all instances with minimal finishing time. However, this requires solving a non-linear integer program with the number of variables proportional to the product of the number of data

chunks and the number of instances, which is unrealistic to finish in real time. Instead, we approximate the goal in two steps. First, we minimize the amount of data transferred by maximizing the number of data chunks that can be served at the current instances after speed increases. Second, we choose the source and destination pairs for each data chunk to be migrated using a greedy algorithm.

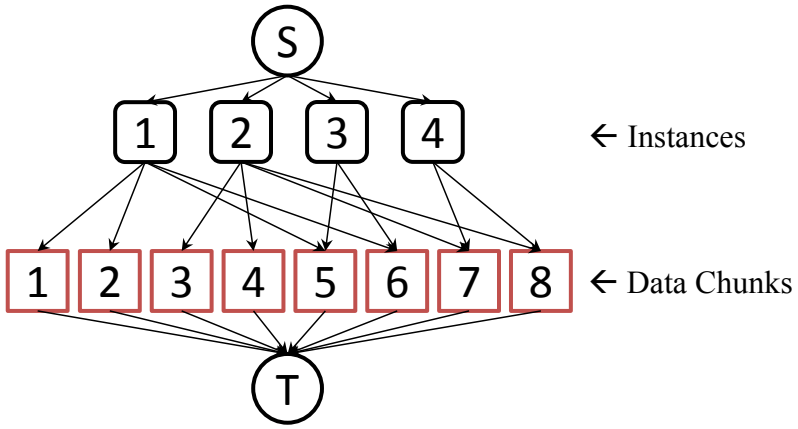


Figure 4.5: Max-Flow Formulation for Minimizing Data Movement

In the first step, we formulate the data movement minimization as a max-flow problem, as shown in Figure 4.5. The source S sends flow into the four instances that currently store data chunks. The capacity of the flow from S to instance i equals to the number of data chunks instance i can serve after the speed increase. In this example, all flows from S to instances 1-4 have unit capacity. Each instance sends a unit capacity flow to any data chunk it stores. For example, instance 1 stores data chunks 1,2,5 and 6. Therefore it sends a unit capacity flow to each of data chunks 1,2,5 and 6. Finally, each data chunk sends a unit capacity flow to the sink T. A unit capacity flow from S to T indicates that one data chunk can be served locally after the speed increase. By maximizing the network flow from S to T, we maximize the number of distinct

data chunks that do not need to migrate to achieve the new speed. One max-flow in this example is the aggregation of four unit capacity flows $< S, 1, 1, T >$, $< S, 2, 3, T >$, $< S, 3, 5, T >$, $< S, 4, 7, T >$, with total capacity 4. Therefore, as shown in Figure 4.6 and 4.7, data chunks 1, 3, 5 and 7 (solid red square) can be served without migrating after the speed increase. An alternative way to take advantage of data locality is to assign each data chunk randomly to one instance that currently stores it. Indeed, this simple approach is equivalent to generating a source to sink flow in the above flow formulation. This flow, however, may not be maximal.

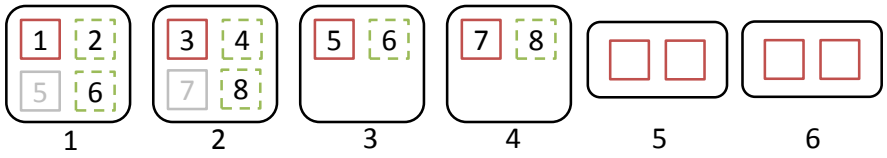


Figure 4.6: Increased Speed = 2, before Rebalancing

The next step is to decide how to transfer data chunks that cannot be served locally (dashed green squares in Figure 4.6) to instances with unused capacities. We greedily allow instances that are reactivated after the speed increase (instances 3 and 4) to pick distinct data chunks to be sent out first. In the example, instance 3 can send out data chunk 6 and instance 4 can send out data chunk 8. Then instances that were active before the speed change pick the remaining distinct data chunks. In the example, data chunk 2 is picked by instance 1, as data chunk 6 has already been selected by instance 3. Similarly, data chunk 4 is picked by instance 2. We start all the data migration simultaneously, with the destination picked among instances with unused capacity, in arbitrary order. In the example, instance 5 receives data chunk 2 from instance 1 and data chunk 6 from instance 3; instance 6 receives data chunk 4 from instance 2 and data chunk

8 from instance 4. The data placement after rebalancing is shown in Figure 4.7.

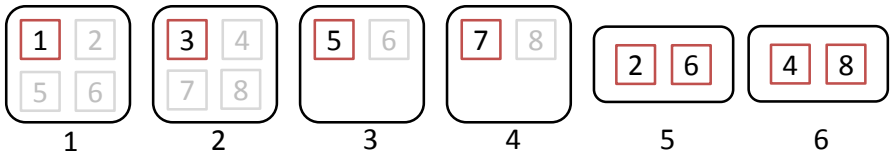


Figure 4.7: Increased Speed = 2, after Rebalancing

Before instances 1-4 reach their termination times, two new instances with type BESTTYPE(2) are allocated. Data chunks 1, 3, 5 and 7 are moved to the two new instances in the background. Now all instances have converged to type BESTTYPE(2), as shown in Figure 4.8.

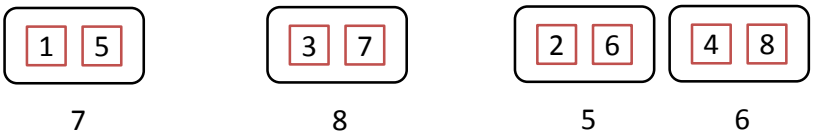


Figure 4.8: Increased Speed = 2, after terminating instance 1-4

4.5 SmartShare

In this section we introduce SmartShare, which enables resource sharing between multiple interactive sessions from different users. Using SmartShare, multiple interactive sessions can share the same set of instances. Rather than maintaining fairness between the users, SmartShare takes advantage of the dynamic instance allocation in the cloud and provides a strict interactive SLA as if each interactive session were being managed by an independent Smart. While providing strong performance isolation, SmartShare can better utilize resources and obtain significant cost saving compared with non-sharing solutions.

4.5.1 Instance Management

In Smart, we obtain the most cost effective instance type for a given speed using the BESTTYPE function. However, when there are concurrent interactive sessions with different speeds, a combination of instance types may be required to maximize cost effectiveness, so an analogous BESTTYPE function that selects a single instance type does not exist.

We can still leverage the BESTTYPE function for instance allocation by defining the *aggregated speed* s_{aggr} as the ratio between the total compute units in use and the total memory in use. We allocate new instances with type BESTTYPE(s_{aggr}), because this is the best fit for the current aggregated workload. As before, because the workload is changing over time, multiple instances types can coexist.

Consolidation. Terminating suboptimal instances and switching to the most cost effective instance type works in Smart but not in SmartShare. Even after an interactive session finishes, there is a high likelihood that other users are still using all the instances and we cannot remove the `active` label from any of them. To scale down the total size of allocated resources, underutilized instances must be consolidated.

Before an instance i reaches the end of its current billing time period, we determine whether there is enough unused capacity in other running instances to absorb the remaining data from i and there is enough time before the end of the billing time period to perform data migration. If so, all data is migrated away from instance i in the background and i is terminated. Otherwise, instance i remains unchanged.

For cloud providers with very short billing time periods, focus can be shifted to finding the instances that can be consolidated with the least migration overhead. This can be done by periodically scanning the instance list and picking an instance i such that there is enough unused capacity in other running instances to absorb its remaining data, and the amount of data migration is as small as possible.

4.5.2 Initial Data Placement

When a user starts a new interactive session, its data needs to be loaded from cloud storage. Unlike Smart, which uses only empty instances when interactive sessions start, SmartShare first tries to place the data in existing shared instances. New instances of type BESTTYPE(s_{aggr}) are allocated only when existing instances do not have enough unused capacity to host the dataset.

The data placement problem here is related to the traditional online vector bin packing problem. However, it is more complicated. SmartShare can freely place the partitioned dataset into arbitrarily many instances with different sizes for each instance, as long as the speed requirement is met. Each additional partition incurs extra overhead for storing replicated data but allows more flexible placement.

Based on traditional techniques used in online vector bin packing, we present two greedy algorithms for initial data placement. In Section 4.7.1, we experimentally study the effectiveness of each algorithm.

1. First Fit. We say an instance is a *qualifying* instance for given datasets when it

has enough CU and memory to hold a portion of the partitioned dataset along with the entire replicated dataset. Similar to First Fit in the bin packing problem, we search for the first qualifying instance in an arbitrary order. For a qualifying instance, we always place as much data as possible, until either all the memory or all the compute units are fully consumed. If there is still data left unassigned after enumerating all existing instances, new instances of type BESTTYPE(*aggr*) are allocated to host the remaining data.

2. Best Fit. Rather than placing data into the first qualifying instance enumerated in an arbitrary order, Best Fit finds the qualifying instance of highest *fitness* first and places as much data as possible there. This process is repeated until all datasets are assigned or there is no qualifying instances in the cluster. In the latter case, new instances of type BESTTYPE(*aggr*) are allocated and added to the cluster.

How to define the fitness between an instance and the given datasets remains a question. In traditional problems, bins with the least unused capacity after placing a given item are considered to have the highest fitness. Similar approaches cannot be applied to this problem for two reasons. First, unused capacities in our case comprise two different resource types, compute units and memory, and are not naturally totally ordered. Second, the datasets can be partitioned arbitrarily and, because of replicated data, the total unused capacity is affected by how we partition the datasets.

We have tested different definitions of fitness, and it turns out that a simple definition works best under various workloads. For a given qualifying instance, assume we can place at most size = m_p of partitioned datasets without violating

resource constraints, we define

$$fitness = m_p \tag{4.3}$$

Under this definition, we try to use as few instances as possible to store given datasets, and therefore minimize the number of copies of replicated datasets. We show the effectiveness of Best Fit under this definition in Section 4.7.1.

After the placement for all data of a new interactive session has been determined, the data is loaded from cloud storage to these instances using the method described in Section 4.4.2 for Smart.

4.5.3 Data Rebalancing for Speed Changes

Like Smart, SmartShare migrates data in the background to implement speed changes.

To deal with a speed reduction for an interactive session, we simply reduce the compute units allocation for the session at all instances where its data is stored. Queries from the interactive session are executed at the reduced speed immediately after the speed reduction.

Speed increase has two cases. Assume interactive session s has increased its speed. For any instance i that stores data for session s, we check whether the unused compute units are sufficient to implement the speed increase. If so, we simply increase the compute unit allocation for session s at instance i. Otherwise, we must reassign the data for interactive session s at this instance to other instances.

The reassignment process can reuse the greedy algorithms for initial data placement. We run the above greedy algorithms in three steps, with a distinct set of instances being considered as migration destinations in each step. In the first step, the dataset is assigned to the current instance, in which case no data needs to be actually moved. Since the current instances do not have enough compute units to host the entire dataset, we must assign a portion of the dataset to other instances in the following two steps. In the second step, we assign the dataset to other instances that also store data for interactive session s. In this case, replicated data does not need to be moved. If any data remains unassigned after the second step, we proceed to the third step, in which all the other instances are considered by the greedy algorithms. If there is still data left unassigned after enumerating all the instances currently in the cluster, new instances of type BESTTYPE($saggr$) are allocated.

4.5.4 Network Bandwidth Sharing

In SmartShare, data is loaded only once, when an interactive session starts, and queries are executed against the same dataset many times before termination. Also, users will tend to change speed infrequently. These observations suggest that if aggregated network bandwidth of instances can be shared among interactive sessions, the data loading speed and the reaction time to speed changes can be improved, since multiple interactive sessions will rarely start or change speed concurrently.

Ideally, to fully use the network bandwidth when an interactive session starts or changes speed, we would want every dataset to be partitioned across

all instances. However, this might not be acceptable because of the memory consumed by replicated data. To control the trade-off between full sharing and the amount of memory devoted to replicated data, we cap the maximum memory an interactive session can use at any instance. This *cap size* must be respected both at initial loading and later migration. By tuning the cap size, we change the level of parallelism for initial loading and speed changes. The lower the cap size, the more instances are used to store each dataset, and the faster both initial loading and speed changes become, but at the cost of devoting more memory to replicated data. We experimentally study the cost and effectiveness of changing the cap size in Sections 4.7.1 and 4.7.2.

4.5.5 Billing

We claim SmartShare can bring significant overall cost saving to users in aggregate, through intelligent resource allocation and data placement. But one question remains: Does the cost saving apply to users individually? We would like to know whether it is possible for use of SmartShare to be more costly for some interactive sessions than just running Smart alone.

To answer this question, we need a billing model for interactive sessions under SmartShare. One might expect such a billing model to compute the actual cost each interactive session is "responsible" for in SmartShare. However, this cost would depend not only on the interactive session's own usage, but also on the workload of other concurrent sessions. For example, a compute heavy interactive session should be "cheaper" when co-existing with memory heavy interactive sessions, compared to being standalone, because the memory heavy

sessions would share some of the cost for resources that would otherwise be idle.

Given the difficulty of devising a fair notion of "responsibility," we propose instead a billing model that charges sessions in proportion to the cost they *would have incurred* running under Smart. Let c_{share} be the total cost of a set of interactive sessions I, c_i be the cost for interactive session $i \in I$ running under Smart without resource sharing, and b_i be the bill for interactive session i. We charge

$$b_i = c_{share} \cdot \frac{c_i}{\sum_{x \in I} c_x} \tag{4.4}$$

Under this billing model, the cost saving ratio for interactive session i is

$$1 - \frac{b_i}{c_i} = 1 - \frac{c_{share}}{\sum_{x \in I} c_x} \tag{4.5}$$

This ratio is the same for all interactive sessions $i \in I$. Thus, every interactive session benefits from SmartShare if Smartshare is able to achieve overall cost savings in the aggregate, that is, if

$$c_{share} \leq \sum_{x \in I} c_x. \tag{4.6}$$

In the experimental results presented in Section 4.7.1, the above inequality always holds, even when there is no concurrency between interactive sessions.

To implement this billing method, we would like to calculate c_i without actually running interactive sessions i in Smart. This can be achieved by logging the non-query operations of each interactive session in SmartShare, and replaying the log to simulate the resource allocation and data placement in Smart following the procedures described in Section 4.4 to obtain the cost under Smart.

4.6 SmartShare+

In this section, we propose SmartShare+, which includes two extensions to SmartShare. Each extension takes advantage of one key characteristic of interactive data analysis workloads.

4.6.1 Resource Over-Allocation

When users interact with large datasets by issuing queries, there is typically a *think time* between seeing the result of a query and submitting the next query [147, 32, 27]. This suggests that resources can be idle for a significant fraction of time during interactive sessions.

One way to improve resource utilization during idle time is to over-allocate computing resources. Over-allocation of compute units by $x\%$ can be achieved by allowing the resource manager to allocate $(1 + x\%)$ of the actual compute units to users, with a restriction that no user can allocate more than the actual compute units at any instance. When $x > 0$, it is possible for concurrent queries to demand more than 100% of the available compute units. If this happens, later queries have to wait, in violation of the strict interactive SLA. A relaxed SLA, as we have defined in Section 4.3, can be used to measure this side-effect of the over-allocation of compute units experimentally. However, it is unrealistic to devise the exact relaxed SLA based on x and the workload directly, because this problem translates to the general $G/G/k$ model in queuing theory, which is intractable [99]. Interestingly, even if we set $x \to \infty$, SmartShare+ still provides stronger guarantees than simply ensuring that all datasets are in memory with-

out considering computation power constraints: SmartShare+ guarantees that when there is only one query running in the system, the strict interactive SLA is always met. Over-allocation of memory space to users requires the ability to evict datasets to larger but slower storage, and read them back quickly. Such storage could be local disks, network attached drives, or cloud storage. Requiring more sophisticated scheduling and performance trade-offs, memory space over-allocation is beyond the scope of this dissertation.

4.6.2 Dataset Sharing

Different users may work on partially or fully overlapping datasets. This is likely if multiple users are from the same organization, or when public datasets draw attention from multiple organizations. Ideally, we want to store each dataset only once. One approach would be to let multiple users share the same interactive session. However, this solution fails if the data is only partially overlapping and not all users have access to all the data. Moreover, sharing interactive sessions would break performance isolation between users and prevent users from specifying different speed requirements.

Therefore, in SmartShare+ users still run separate interactive sessions with exactly the same resource allocation workflow and query performance guarantees. Upon observing dataset overlapping, the system tries to collocate overlapping data for different sessions. When two or more sessions with overlapping data are placed in the same instance, their overlapping data needs to be allocated only once. Of course, the corresponding compute units cannot be saved by collocation since these interactive sessions may run queries at the same time.

The initial data loading process needs to be slightly modified to support dataset sharing. It now runs with two steps. In the first step, only instances that store overlapping datasets are considered. In this case, only compute units and memory for non-overlapping datasets are allocated. All the other instances are considered as normal in the second step.

Data rebalancing for speed changes requires modifications to support dataset sharing as well. In the three-step reassignment process, we alter the second step slightly. Rather than considering only instances storing data from the same session, all instances that contain overlapping data are considered.

With the above changes, SmartShare+ supports data sharing between interactive sessions without sacrificing query performance guarantees.

4.7 Experiments

In this section, we present experiments demonstrating the effectiveness of the techniques we proposed for Smart, SmartShare and SmartShare+. Testing our resource allocation techniques under a variety of workload parameters on a realistic scale would require a large number of runs with thousands of instance hours per run. These experiments would exceed our budget if run in the cloud. Since the decision of instance allocation and data placement is mostly independent of query execution, we first use simulation to learn the effectiveness of each technique. We then demonstrate the applicability of these techniques through deployments in the cloud.

4.7.1 Resource Allocation Simulations

Setup

The resource allocation and data placement logic of Smart, SmartShare, and SmartShare+ is deployed on a single machine, and instance allocation or deallocation calls are replaced by no-ops. The simulator records network queuing time and CPU wait time for each instance at the precision of one second.

The instance types used in the simulation are obtained from Amazon EC2 [5]. We use prices for the US East region. Instances are always started with images stored in Amazon Elastic Block Store. In our 50 trials, the time between instance allocation and successful login is on average 19.9 seconds, with a standard deviation of 5.1 seconds. In EC2, instances are billed for an integer number of hours since allocation, with partial hours rounded up. These numbers are used in the simulator to calculate costs and the time needed to complete speed changes.

To the best of our knowledge, there is no public workload data for in-memory interactive data analysis systems. Therefore, we built our own workload generator. The workload generator generates four types of actions for each user: `start session`, `change speed`, `query data`, and `finish session`. For `start session` and `change speed`, a speed is generated randomly. By default, the speed is chosen from an exponential distribution with mean set to 1 CU/GB, and with cutoffs at 0.1 CU/GB as the minimum and 5 CU/GB as the maximum; the completion time of an interactive operator is chosen from 1 second to 5 seconds uniformly at random; the interval between consecutive speed changes requests and between consecutive queries

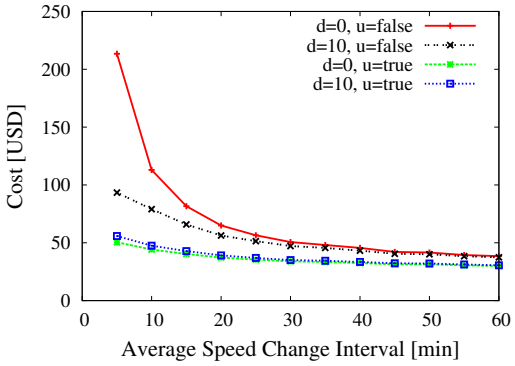


Figure 4.9: Cost: Smart

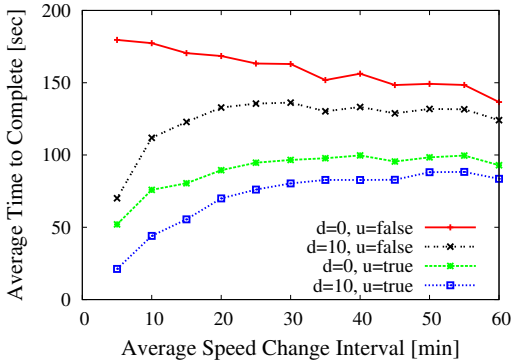


Figure 4.10: Time to Complete Speed Increases: Smart

from the same interactive session are both binomially distributed with configurable means.

Smart

In Figures 4.9 and 4.10, we study the impact of using different termination methods in Smart by simulating interactive sessions of 120 minutes, with a 200GB partitioned dataset. We vary the average speed change interval from 5 minutes to 60 minutes. 100 such interactive sessions are simulated for each average speed change interval. The average cost and the average time to complete speed changes among these 100 interactive sessions are reported. With $u = true$, instances are only terminated when their current billing time period is over, which allows us to make use of additional instances at no extra cost. With $d = 10$, we delay instance termination for 10 minutes after an instance loses either the `active` or `optimal` label, which may incur extra costs but can be helpful if there is a speed increase in the 10-minute time period. A combination of the above two parameters, $d = 10, u = true$, means we first delay for 10 minutes and then wait until the billing time period is over before terminating

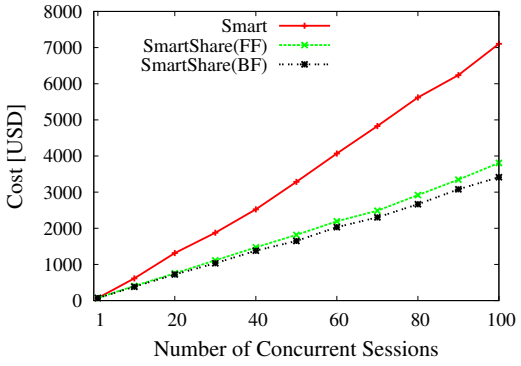


Figure 4.11: Cost: Smart vs. SmartShare

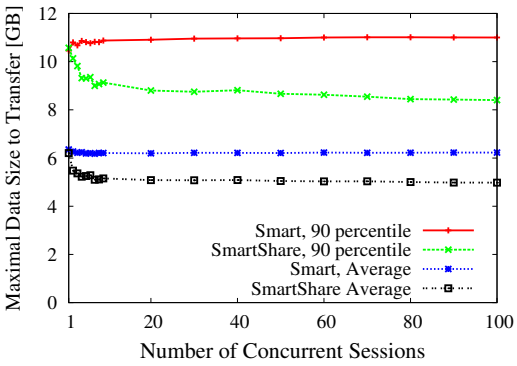


Figure 4.12: Speed Increase Overhead

an instance. With $d = 0, u = false$, suboptimal or inactive instances are terminated immediately and only optimal instances are used. Figures 4.9 and 4.10 show that $d = 0, u = false$ is the most expensive strategy and also takes the longest time to complete speed changes. While being slightly more expensive than $d = 0, u = true$ when average speed change interval is small, $d = 10, u = true$ takes less time to complete speed changes because the extra 10 minutes of waiting time reduces the possibility of having to allocate new instances and wait 19.9 ± 5.1 seconds to access them.

SmartShare

In Figure 4.11 and Figure 4.12 , we compare the cost, as well as the speed increase overhead of running the same set of interactive sessions using Smart with $d = 0, u = true$ and SmartShare. Each interactive session processes a partitioned dataset with size chosen from 10GB to 100GB uniformly at random. The replicated dataset is set to 1% of the total size of the partitioned dataset. The length of a single interactive session is chosen between 30 minutes and 120 minutes uniformly at random, with the overall time span of all interactive sessions being 12

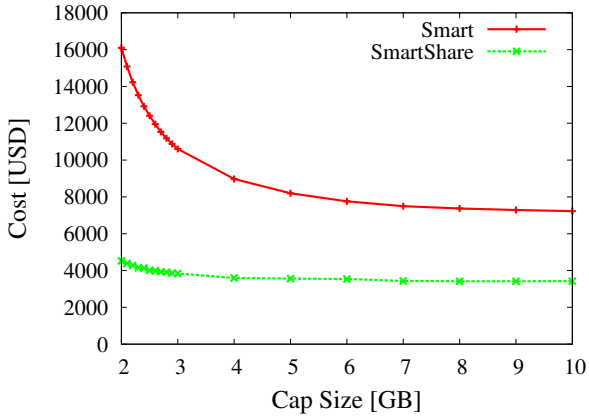


Figure 4.13: Cost: Different Cap Sizes

hours. The average speed change interval is 20 minutes. We vary the number of concurrent sessions from 1 to 100 in SmartShare. As shown in Figure 4.11, by sharing resources between users, significant cost savings can be achieved. With 100 concurrent interactive sessions, SmartShare with First Fit achieves 46% cost saving compared to Smart. SmartShare with Best Fit saves another 10% beyond SmartShare with First Fit.

Figure 4.12 shows the maximum data size that has to be sent or received over the network by any instance to complete a speed increase. Such maximum data sizes are recorded for all speed increases. Speed decreases are always completed instantaneously in SmartShare and therefore are not included in this experiment. Averages and 90 percentiles are then calculated. SmartShare, though it makes no attempt to minimize data movement when implementing speed changes, requires no more data to be transferred than Smart in both average and 90 percentile. This is because resource allocation is more flexible in SmartShare, as there are multiple interactive sessions sharing the resources. Therefore, SmartShare is more likely to complete speed increases locally than Smart.

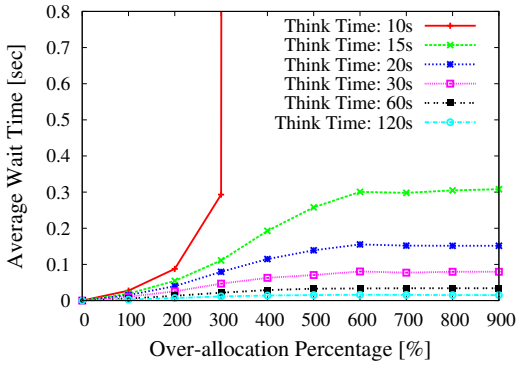


Figure 4.14: Average Wait Time

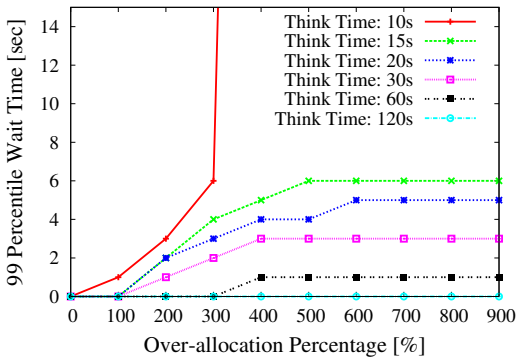


Figure 4.15: 99 Percentile Waiting Time

As discussed in Section 4.5.4, through tuning the maximum memory each interactive session can use at any instance (cap size), we can control the data loading time and the time to complete speed increases. Figure 4.13 shows that with 100 concurrent sessions, the cost of SmartShare is less sensitive to changing cap sizes, compared with the cost of Smart. The reason is that for an instance in SmartShare, while only limited memory size can be used by one interactive session, it is likely that other concurrent interactive sessions can utilize the resources. In contrast, Smart has only one interactive session, so more resources are wasted with a smaller cap size. We show the effectiveness of capping through experiments in the cloud, in Section 4.7.2.

SmartShare+

In Section 4.6.1, we explained the possibility of over-allocating compute units to better utilize resources during users' think time, at the cost of relaxing the SLA. Figures 4.14 and 4.15 show simulation results for the amount of SLA relaxation resulting from over-allocation percentages ranging from 0% to 900% and aver-

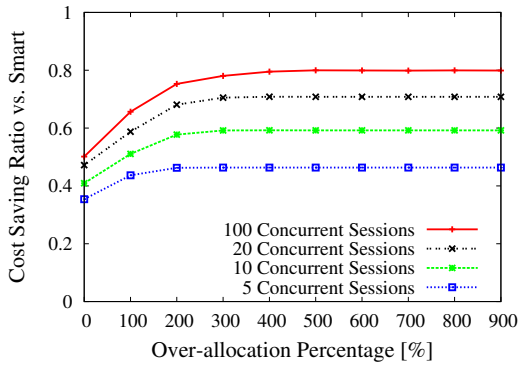


Figure 4.16: Cost Saving: Over-allocation

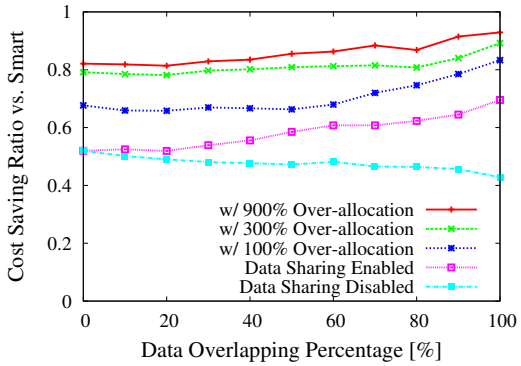


Figure 4.17: Cost Saving: Data Sharing and Over-allocation

age think time ranging from 10 seconds to 120 seconds. Other workload parameters are the same as in Figure 4.13. Since the precision of the simulation is set to one second, 99 percentile wait times in Figure 4.15 are all integers, but the average wait times in Figure 4.14 need not be integers. In both figures, the wait time increases sharply when the average think time is 10 seconds and the over-allocation percentage exceeds 300%. This is because in theses cases, incoming queries arrive at a higher rate than they can be processed and the wait queue for compute units grows without bound. In all other scenarios, the average wait time is below 0.35 seconds and the 99 percentile is no more than 6 seconds. When the average think time is 120 seconds, the 99 percentile wait time is 0 seconds, even with 900% over-allocation. These results show the acceptability of enabling over-allocation when a slightly relaxed SLA can be tolerated.

Figure 4.16 shows the cost saving that can be achieved by enabling over-allocation. The y axis indicates the cost saving ratio compared with running these interactive sessions using Smart. We vary the over-allocation percentage from 0% to 900% and the number of concurrent interactive sessions from 5 to 100. With 100 concurrent interactive sessions, more than 75% cost saving

can be achieved with 300% over-allocation, which effectively halves the cost of SmartShare without over-allocation. The cost saving ratio does not increase further beyond 400% over-allocation. This is because, with such high over-allocation percentages, memory becomes the bottleneck under our workload.

As discussed in Section 4.6.2, when multiple interactive sessions work on overlapping datasets, memory can be saved by collocating these sessions. We study this effect in Figure 4.17 by creating 10 partitioned datasets shared among interactive sessions, with all the other settings the same as in Figure 4.13. As in Figure 4.16, the y axis indicates the cost saving ratio compared with running these interactive sessions using Smart. We vary the percentage of interactive sessions that only work on the shared datasets, and each of these interactive sessions randomly chooses one partitioned dataset out of the 10 shared ones. With the percentage of interactive sessions working on shared datasets increasing from 0% to 100%, enabling data sharing yields from 0% to 27% higher cost saving ratio compared with disabling data sharing. The cost saving ratio does not increase substantially when more interactive sessions share datasets. This is due to compute units becoming the bottleneck: even if all 100 concurrent interactive sessions work on only 10 different datasets, all the compute units still have to be allocated as normal, so we incur much more than 10% of the cost. While over-allocating compute units makes memory the bottleneck, dataset sharing makes compute units the bottleneck. Thus, a combination of these techniques should achieve an even better cost saving ratio. Figure 4.17 confirms this observation. With 900% over-allocation and 100% of interactive sessions working on 10 shared datasets, we achieve 93% cost saving compared with running these interactive sessions separately in Smart.

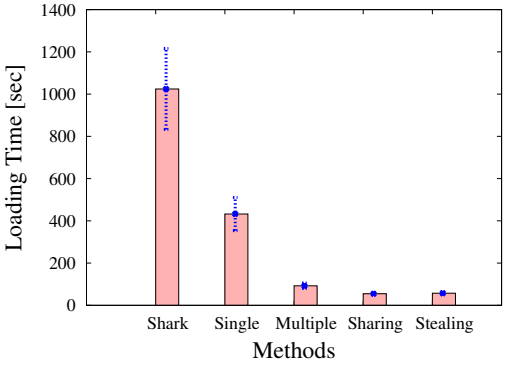


Figure 4.18: Loading Time: Smart

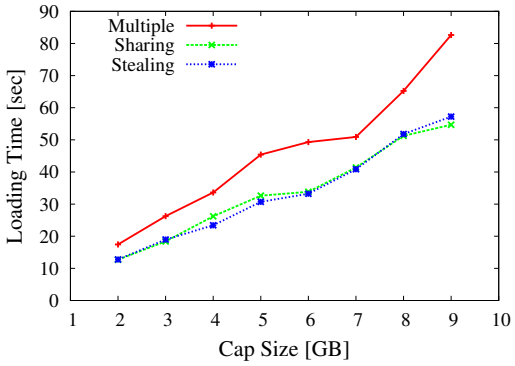


Figure 4.19: Loading Time: SmartShare

4.7.2 Deployments in the Cloud

Figure 4.18 compares the average loading time for 45GB of data between the techniques discussed in Section 4.4.2, using 5 m1.xlarge instances in EC2. The data is loaded to instances from Amazon S3 [6]. The loading time is measured for 5 times with error bars reporting its standard deviation. By using 8 threads per instance (shown as ”Multiple’ in the figure), we reduce the average loading time by 79% compared with using a single thread. Work sharing or work stealing achieves another 38% to 41% time reduction compared with Multiple, making the average loading time less than 1 minute. The performance difference between work sharing and work stealing is insignificant, suggesting that the coordination overhead is dominated by other factors. We also run Shark on the same set of instances following the setup described in [123]. It takes Shark on average over 17 minutes to load 45GB of data, which is slower than our implementation of using a single thread. This shows that without an optimized data loader, data loading from cloud storage can take unacceptably long time for interactive sessions.

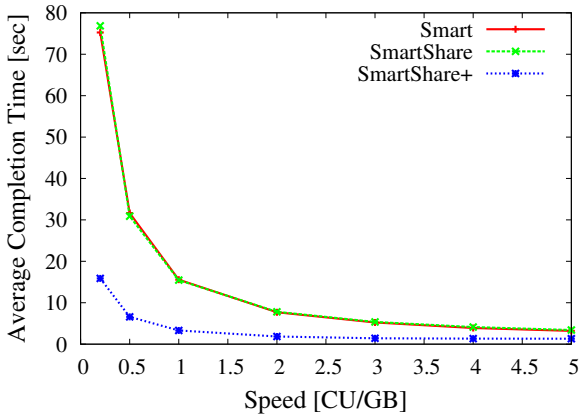


Figure 4.20: Query Completion Time: Logistic Regression

In SmartShare, by reducing the maximum memory each interactive session can use at any instance (cap size), we can finish the initial data loading faster. Figure 4.19 shows this effect. The same S3 dataset is used as in Figure 4.18. By reducing the cap size from 9GB to 2GB, we increase the number of m1.xlarge EC2 instances used from 5 to 23, and the loading time of the 45GB dataset is reduced from more than 54 seconds to less than 13 seconds, under either work stealing or work sharing. Meanwhile, the maximum data size that has to be sent or received over the network by any instance to complete a speed increase is strictly limited by the cap size. Therefore, when the cap size is set to 2GB, the 100 percentile of max data size to transfer is guaranteed to be no larger than than 2GB. This is a much stronger bound than what we can achieve without capping, as shown in Figure 4.12.

We compare the performance of the same user under Smart, SmartShare, and SmartShare+ in EC2. The user runs interactive queries over a 12GB dataset with different speeds. In SmartShare and SmartShare+, this user shares resources with 9 other users, each also using a 12GB dataset. 400% over-allocation is used in SmartShare+. Each other user randomly chooses one interactive operator to

run from scan, group by, or one iteration of logistic regression, with 5 seconds think time. Multiple interactive operators from different tenants may share a physical core in cloud instances; we manage them by giving each operator a slice of running time proportional to its share and alternating between operators. Figure 4.20 shows the query completion time of one iteration of logistic regression for this user, averaged with 100 runs under Smart, SmartShare, and SmartShare+. The average query completion time of SmartShare matches the one of Smart well. Both of them are able to achieve 22 times speedup when speed changes from 0.2 to 5 CU/GB. SmartShare+ with 400% over-allocation, is able to achieve 4.8 to 2.6 times speedup in average query completion time, compared with Smart. The 99 percentile of query completion time of both Smart and SmartShare is at most 10% larger than their averages. However, under SmartShare+, the 99 percentile of query completion time increases when speed is larger than 3. This is because the increase in wait time when using a higher speed, dominates the reduction in actual query running time with the higher speed. With speed = 5, the 99 percentile is 2.5 times the average under SmartShare+, making it worse than the 99 percentile of Smart. Similar observations hold for the query completion time of scan and group by.

4.8 Conclusions

We have proposed a simplified resource allocation workflow for interactive data analysis in the cloud, which enables data scientists to control query execution speed dynamically. We explained how to achieve optimized resource allocation following this new workflow using Smart, a new elastic cloud-aware resource manager. We extended Smart to SmartShare, which enables data scientists to

share cloud resources, achieving more than 50% cost reduction without compromising the query SLA. Finally we described SmartShare+, which can take advantage of think time and overlapped data to further improve memory and computing resource utilization and achieve even greater cost reduction.

As future work, we plan to integrate our techniques into existing widely used interactive data analysis engines, such as Shark. Another direction of work is to implement more sophisticated failure recovery, especially for master node failures. Finally, we will investigate the possibility of achieving more cost savings by using a combination of resources from multiple cloud service providers simultaneously to meet users' interactive SLAs.

CHAPTER 5

RELATED WORK

5.1 Programming Frameworks for Distributed Applications

Previous literature has studied programming abstractions for scientific applications as well as techniques to deal with latency in execution environments. But this work has neither taken a general, data-centric view of programming for these applications nor dealt with the specific challenges posed by cloud environments.

There has been significant work on parallel frameworks for writing discrete event simulations. These systems are based on task parallelism, and handle conflicts by either conservative or optimistic protocols. Conservative protocols limit the amount of parallelism, as potentially conflicting events are serialized [29, 54]. Optimistic protocols, on the other hand, use rollbacks to resolve conflicts [76]. Time-stepped applications typically eschew these approaches, because the high frequency of local interactions causes numerous conflicts and rollbacks, limiting scalability.

Since the mid-1990s, the Message Passing Interface (MPI) Standard has dominated distributed-memory high-performance computing due to its portability, performance, and simplicity [63]. Even in its early days MPI was criticized as inelegant and verbose, and in domains where parallel applications evolve rapidly the relatively low level of MPI programming is perceived as a significant drawback [68, 96]. Thus, there have been efforts to move away from MPI. The DARPA High-Productivity Computing Systems (HPCS) initiative [42]

has funded several systems intended to provide attractive alternatives to MPI, mostly based on new parallel languages. In some domains, it has been possible to shield application developers from MPI with high-level application frameworks designed by experts. For example, a recent flurry of work has focused on graph processing without MPI [31, 80, 95, 98]. Unfortunately, this work does not generalize to the wide class of bulk synchronous applications. MPI remains the dominant programming paradigm for this class of applications.

MPI's low-level programming abstraction creates several difficulties for developers wishing to port bulk synchronous applications to the cloud. In particular, dealing with jitter requires a significant rewrite of the communication layer of most of these applications. Unfortunately, there is not yet consensus on the best techniques to use.

The scientific computing literature includes many established techniques for dealing with uniform communication latency. For example, asynchronous communication primitives facilitate communication *hiding*, and many bulk synchronous applications use these primitives to overlap computation and communication. These optimizations work best when communication latency is uniform and predictable, and it can be difficult in practice to characterize their effectiveness [125].

Grid-based MPI systems such as MPICH-G2 give application developers mechanisms to adapt their applications for environments in which communication latencies are nonuniform due to network heterogeneity [81]. Unfortunately, these systems do not address dynamic latency variance within a single point-to-point communication channel, which is common in the cloud.

The most scalable parallel algorithms do not just hide communication overhead; they also *avoid* communication at the expense of performing some redundant computation. This idea has been used for years in large-scale PDE solvers, where each process is responsible for a part of a mesh surrounded by a layer of "ghost cells" used to receive data from neighboring processes. By using multiple layers of ghost cells, processes can effectively communicate not at every tick, but once every several ticks. These ideas have been extended to the more general setting of sparse linear algebra [47]. While communicating less often certainly helps, this technique alone cannot deal with latency spikes. Even if multiple layers of ghost cells are used, when a message is scheduled to be delivered the receiving process must block waiting for it. Intuitively, in order to tolerate a latency spike, whenever possible, the receiving process should run some useful work that it can perform until the delayed message arrives.

Other techniques from the HPC community target bulk synchronous applications, such as balancing the computation and communication load among processes [105], forming subgroups of processes for global synchronization [23], and replacing the global synchronization barriers with local synchronization by dynamically exploiting locality during each time step [2, 83]. In contrast to our approach, all of these methods block at synchronization points if messages are not available. In order to deal with jitter, new techniques need the flexibility to either take incoming messages at synchronization points or proceed with useful work in case these messages are not available. Our scheduling and replication techniques achieve this goal, generalizing and extending the special case of ghost cells described above to enable both reduced communication and jitter-tolerance.

Specific algorithms have been developed to accelerate convergence of iterative methods, effectively reducing the total communication requirements. Examples include methods for graph algorithms, such as fast convergence of PageRank [79, 87], as well as for computation of large-scale linear systems and eigenvalue problems, such as Krylov subspace methods [16, 47]. While many of these techniques change the communication pattern of applications to accelerate execution, they do not generalize across different applications domains.

Data parallel programming languages provide automatic parallelism over regular data structures such as sets and arrays [21, 24, 128, 43]. However, these approaches only support restricted data structures, making it both unnatural and inefficient to express certain time-stepped applications, such as behavior simulations. In addition, there is little support in these programming models for declaring dependencies among subsets of data.

Emerging programming models for the cloud, such as MapReduce [45] or DryadLINQ [157], have limited support for iterative applications; a number of recent proposals target exactly this issue [26, 94, 159]. Most of these optimizations add support for resident or cached state to a MapReduce programming model. The basic assumption is that the dominant factor in computation time is streaming large amounts of data at every iteration. In contrast, this dissertation looks at scientific applications with fast iterations where computation time typically exceeds data access time. In these scenarios, network jitter is a fundamental optimization aspect.

In previous work, we have shown how database techniques can bring both ease of programming and scalability to behavioral simulations [143], but we did not address how to tolerate network jitter. Related is also Bloom, a declarative,

database-style programming environment for the development of distributed applications in the cloud [3]; our work is not as ambitious as it only targets BSP scientific applications and focusses on network jitter. A confluence of BLOOM and CALM and our techniques is an interesting direction for future work.

5.2 Deployment Optimization in Public Clouds

Subgraph Isomorphism. The *subgraph isomorphism problem* is known to be NP-Complete [57]. There is an extensive literature about algorithms for special cases of the subgraph isomorphism problem, e.g., for graphs of bounded genus [101], grids [102], or planar graphs [48]. Algorithms based on searching and pruning [38, 37, 137] as well as constraint programming [86, 161] have been used to solve the general case of the problem. In our Node Deployment Problem, a mapping from application nodes to instances needs to be determined as in the subgraph isomorphism problem, but in addition the mapping must minimize the deployment cost.

Overlay Placement. Another way to look at the Node Deployment Problem is to find a good overlay within the allocated instances. The networking community has invested significant effort in intelligently placing intermediate overlay nodes to optimize Internet routing reliability and TCP performance [67, 122]. This community has also investigated node deployment in other contexts, such as proxy placement [90] and web server/cache placement [84, 112]. In the database community, there have been studies in extensible definition of dissemination overlays [110], as well as operator placement for stream-processing systems [111]. In contrast to all of these previous approaches, ClouDiA focuses

on optimizing the direct end-to-end network performance without changing routing infrastructure in a datacenter setting.

Virtual Network Embedding. Both the virtual network embedding problem [33, 36, 51, 156] and the testbed mapping problem [120] map nodes and links in a virtual network to a substrate network taking into account various resource constraints, including CPU, network bandwidth, and permissible delays. Traditional techniques used in solving these problems cannot be applied to our public cloud scenario simply because treating the entire datacenter as a substrate network would exceed the instance sizes they can handle. Recent work provides more scalable solutions for resource allocation at the datacenter scale by greedily exploiting server locality [14, 64]. However, this work does not take network latency into consideration. CloudDiA focuses on latency-sensitive applications and examines a different angle: We argue network heterogeneity is unavoidable in public clouds and therefore optimize the deployment as a cloud tenant rather than the infrastructure provider. Such role changing enables us to frame the problem as an application tuning problem and better capture optimization goals relevant to latency-sensitive applications. Also, we only need to consider instances allocated by a given tenant, which is a substantially smaller set than the entire data center. Of course, nothing precludes the methodology provided by ClouDiA being incorporated by the cloud provider upon allocation for a given tenant, as long as the provider can obtain latency-sensitivity information from the application.

Auto-tuning in the Cloud. In the database community, there is a long tradition of auto-tuning approaches, with AutoAdmin [30] and COMFORT [144] as some of its seminal projects. Recently, more attention has focused on auto-tuning in

the cloud setting. Babu investigates how to tune the parameters of MapReduce programs automatically [12], while Jahani et al. automatically analyze and optimize MapReduce programs with data-aware techniques [75]. Lee et al. optimizes resource allocation for data-intensive applications using a prediction engine [88]. Conductor [146] assists public cloud tenants in finding the right set of resources to save cost. Both of the above two approaches are similar in spirit to ClouDiA. However, they focus on Map-reduce style computation with high bandwidth consumption. Our work differs in that we focus on latency-sensitive applications in the cloud, and develop appropriate auto-tuning techniques for this different setting.

Cloud Orchestration. AWS CloudFormation [9] allows tenants to provision and manage various cloud resources together using templates. However, interconnection performance requirements cannot be specified. AWS also supports cluster placement groups and guarantees low network latency between instances within the same placement group. Only costly instance types are supported and the number of instances that can be allocated to the same placement group is restricted. HP Intelligent Management Center Virtual Application Network Manager [72] orchestrates virtual machine network connectivity to ease application deployment. Although it allows tenants to specify an 'information rate' for each instance, there is no guarantee on pairwise network performance characteristics, especially network latency. Wrasse [114] provides a generic tool for cloud service providers to solve allocation problems. It does not take network latency into account.

Topology-Aware Distributed Systems. Many recent large-scale distributed systems built for data centers are aware of network topology. Cassandra [85] and

Hadoop DFS [66] both provide policies to prevent rack-correlated failure by spreading replicas across racks. DyradLINQ [157] runs rack-level partial aggregation to reduce cross-rack network traffic. Purlieus [109] explores data locality for MapReduce tasks also to save cross-rack bandwidth. Quincy [74] studies the problem of scheduling with not only locality but also fairness under a fine-grained resource sharing model. The optimizations in these previous approaches are both rack- and application-specific. By contrast, ClouDiA takes into account arbitrary levels of difference in mean latency between instances. In addition, ClouDiA is both more generic and more transparent to applications.

Network Performance in Public Clouds. Public clouds have been demonstrated to suffer from latency jitter by several experimental studies [126, 142]. Our previous work has proposed a general framework to make scientific applications jitter-tolerant in a cloud environment [167], allowing applications to tolerate latency spikes. However, this work does little to deal with stable differences in mean latency. Zaharia et al. observed network bandwidth heterogeneity due to instance colocation in public clouds and has designed a speculative scheduling algorithm to improve response time of MapReduce tasks [160]. Farley et al. also exploit such network bandwidth as well as other types of heterogeneity in public clouds to improve performance [53]. To the best of our knowledge, ClouDiA is the first work that experimentally observes network *latency* heterogeneity in public clouds and optimizes application performance by solving the Node Deployment Problem.

5.3 Speed Control for Interactive Data Analysis in the Cloud

Cluster Resource Manager. There has been much work in the HPC community on efficient management of cluster resources. The TORQUE Resource Manager [134] is a widely-used distributed resource manager which provides control over batch jobs and distributed computing resources. Condor [93] manages large heterogeneous collections of distributed machines and uses their idle time for running computation tasks. The ClassAd matchmaking framework is used in Condor to schedule tasks on machines with enough available resources [117]. LSF [164, 165] enables sharing of compute clusters between organizations for both batch and interactive jobs. Recent cluster resource managers focus on supporting multiple applications or computing frameworks with resource isolation. Mesos [71] introduces a two-level scheduling mechanism to support fine-grained resource sharing between multiple cluster computing frameworks. The resource offering in Mesos is pushed to each framework by allowing them to take a lock on the global resource table and select resources within the table. YARN [139], the resource manager for Hadoop 2.0, uses a pull-based model where YARN receives resource requests from applications and allocate resources based on application-specified constraints. Omega [129] allows different frameworks to access a shared cluster state together under optimal concurrency control to achieve better flexibility and scalability. None of the above resource managers supports elastically changing the overall cluster size, and none of them can compose the most cost-effective cluster using heterogeneous instances.

Task Scheduler. Hadoop uses queue-based schedulers to enable sharing between users while maintaining fairness [136] or guaranteeing minimum capac-

ity [135]. FLEX [149] extends the Hadoop scheduler to optimize for a variety of standard scheduling theory metrics. Quincy [74] is a fair scheduler for Dryad that tackles the conflict between locality and fairness using a min-cost flow formulation. Delay Scheduling [158] suggests that with a small amount of wait time for getting locality, it achieves nearly optimal data locality and fairness in a variety of workloads. Ghodis et al. proposes Dominant Resource Fairness (DRF) for fair resource allocation with multiple resource types [60]. Under DRF, increasing the allocation of any user will decrease the allocation of another user. Hierarchical DRF [20] extends DRF support hierarchies scheduling which ensures each node in the hierarchy tree obtains at least a predefined fair share of resources. Choosy [61] is a fast online scheduler that approximates optimal constrained max-min fairness with on average 2% difference. All these schedulers are mainly designed for batch jobs and do not provide steady and dynamically configurable performance guarantees for interactive in-memory computing.

Performance Isolation in the Cloud. Providing cloud services with performance isolation has recently attracted much attention. Pisces [132] achieves per-tenant performance isolation and fairness for shared key-value storage. Cake [141] allows tenants to set high-level service level objectives to explore the trade-off between latency and throughput in a distributed storage system. SQLVM [104, 44] enables performance isolation between tenants in a single database server without statically assigning resources. These systems target different kinds of cloud services than our focus on distributed in-memory data analytics. Other work improves performance isolation between users sharing the same last level cache [162], CPUs and network devices [65], and data center network [131]. These techniques build the infrastructural foundation to support

application-level performance isolation, complementing our work.

Interactive Data Analytics. Spark [159], built on top of Mesos, supports iterative machine learning algorithms and interactive data analysis by holding the entire dataset in memory and using resilient distributed datasets to retain fault tolerance. Shark [151] extends Spark to support OLAP SQL queries and reoptimizes queries at run-time based on fine-grained data statistics. Both DB2 with BLU Acceleration [118] and SAP HANA [52] achieve interactive querying by optimizing for in-memory processing. Cetintemel et al. proposes a database navigator service for interactive data analysis which helps users find interesting data [27]. Stat! [15] generates query results progressively to interact with data scientists by using a streaming engine. These emerging interactive data analytics engines motivate our work.

Bin Packing and Virtual Machine Placement. For the one-dimensional online bin packing problem, Yao shows that Revised First-Fit has performance ratio 5/3, and also no online algorithm can have performance ratio better than 3/2 [152]. Efforts have been made to extend the original online bin packing problem to more complicated realistic settings. Variable-sized online bin packing problem allows bins of different sizes [41, 130]. Epstein considers the case that each bin is only allowed to store a bounded amount of tasks [49]. In the *vector* bin packing problem, objects are treated as d-dimensional vectors and a set of vectors fits into a bin if and only if the sum of the vectors is no larger than one in each dimension. For online vector bin packing, a simple first fit algorithm provides a (d+0.7) competitive ratio [56] and the lower bound was recently proved to be $\Omega(d^{1-\epsilon})$ [11]. Inspired by the virtual machine placement and tenant allocation problems in data centers, recent work studies the effectiveness of variations

of bin packing algorithms under realistic workloads, independent of worst-case performance [18, 121, 150]. Optimization models for virtual machine allocation problem have also been widely study recently, for multiple objects [163], price uncertainty [28], energy consumption [77], and traffic-awareness [100]. The data placement problem we need to solve, although related to an online vector bin packing and virtual machine placement problem, is strictly harder than the above extensions for two reasons. First, in our systems, interactive sessions can be terminated and consequently cause data removal at *any* time. Also, we are allowed to freely partition the dataset before packing, as long as the speed requirement is met.

User Interaction Model. The importance of taking user's think time into consideration in designing and benchmarking interactive systems has been addressed by many researchers [1, 107, 127, 32, 27]. Agrawal et al. model the user think time between reads and writes within a transaction [1]. Schaffner et al. use a random think time drawn from a exponential distribution with a mean of five seconds for performance measurement [127]. Olston et al. discuss the possibility of using the think time between queries to compute answers of anticipated future queries [107]. Cetintemel et al. also employ user think time to run prefetching and caching for interactive data exploration [27].

APPENDIX A

APPENDIX FOR CHAPTER 3

A.1 Formal Model of Computation

In this section, we provide a formal presentation of the time-stepped model introduced in Section 2.4.

A.1.1 Data Dependencies

We refer the reader to Section 2.4 for the definitions of global state and time-stepped application logic.

Section 2.4 also introduced function STEP. We now formalize the constraints we need to place on the relationship between the two input parameters of STEP. Properties 1 and 2 require that the stepping state be a subset of the context state. In order to express data parallelism and adjust the layers of computation, we want a stronger relationship.

Definition 5 (Read Data Dependency)**.** For a given STEP and query Q, we say $Q \prec_R Q'$ if and only if, for any state S, $Q(S) \subseteq Q'(S)$ and

$$\text{STEP}\Big(Q(S), S'\Big) = \text{STEP}\Big(Q(S), S\Big) \text{ for all } Q'(S) \subseteq S' \subseteq S \tag{A.1}$$

The intuition of Definition 5 is that using $Q'(S)$ as context is sufficient to give us the correct results for $Q(S)$, and adding more tuples does not change this result. The functions $\mathrm{R_D}$ and $\mathrm{R_X}$ in Section 2.4 are declared according to this definition, such that $\forall Q, \mathrm{R_X}(Q) \prec_R Q$ and $Q \prec_R \mathrm{R_D}(Q)$.

In the algorithms presented in Chapter 2, we often break our dependency sets up into several layers. The following proposition is useful for combining these layers back together.

Proposition 1. *For any queries $Q_a \prec_R Q'_a$ and $Q_b \prec_R Q'_b$, we have $Q_a \wedge Q_b \prec_R Q'_a \wedge Q'_b$.*

Proof Sketch. This result follows immediately from Property 2 and Definition 5. □

Another challenge is the representation of write dependencies. In the fish simulation example, the partitions are geometric regions, and a fish may swim from one region to another. Fortunately, time-stepped computation ensures that we need only look at the current state, and not the history of migrations. So we only need to identify the write dependencies at each time step, and use that to guide our interprocess communication. This is the motivation for the following definition.

Definition 6 (Write Data Dependency)**.** For a given STEP and query Q, we say $Q \prec_W Q'$ if and only if, for any state S, $Q(S) \subseteq Q'(S)$ and

$$Q\Big(\text{GSTEP}(S)\Big) = Q\Big(\text{STEP}(S', S)\Big) \text{ for all } Q'(S) \subseteq S' \subseteq S \tag{A.2}$$

Intuitively, if $Q \prec_W Q'$, we are guaranteed that no tuple outside the state specified by Q' will create tuples into the local state $Q(S^t)$ during any time step t. Therefore, by computing $\text{STEP}(Q'(S^t), S^t)$, we can obtain the complete $Q(S^{t+1})$ which already contains all possibly written data. In fact, the functions W_D and W_X in Section 2.4 are declared according to this definition, such that $\forall Q, W_X(Q) \prec_W Q$ and $Q \prec_W W_D(Q)$.

A.1.2 Correctness of Algorithms

We now illustrate how we use this formal model to establish the correctness of the algorithms presented in Chapter 2.

Theorem 1 (Correctness of Algorithm 1)**.** *Let S_i^t be the value for process P_i at line 13. Then $\bigcup_i Q_i(S_i^t) = S^t$.*

Proof. We know from Property 2 and Definition 5 that

$$S^{t+1} = \bigcup_{i=1}^{n} \textsc{Step}(Q_i(S^t), \mathrm{R_D}(Q_i)(S^t)) \tag{A.3}$$

As each query Q_i is monotonic, by Property 1 we only need to prove $Q_i(S^t) \subseteq S_i^t$ and $\mathrm{R_D}(Q_i)(S^t) \subseteq S_i^t$. As Q is properly contained in $\mathrm{R_D}(Q)$, Definition 6 ensures that $Q \prec_W \mathrm{W_D}(\mathrm{R_D}(Q))$. Hence DISJOINT guarantees that we communicate the right information to ensure $\mathrm{R_D}(Q_i)(S^t) \subseteq S_i^t$ by line 13. □

Theorem 2 (Correctness of Algorithm 2)**.** *Let $S_i^{t_w}$ be the value for process P_i at line 20. Then $\bigcup_i Q_i(S_i^{t_w}) = S^{t_w}$.*

Proof Sketch. Because of the nested loop in lines 18 to 23, we need to show that the algorithm halts. In particular, we must guarantee that TRYRECEIVE(t_w) at line 19 eventually succeeds for all t_w. This argument proceeds by induction; assuming that TRYRECEIVE(t) has succeeded for all processes for $t < t_w$, then DEPTH[t_w] = 0 for all these processes and we execute line 14.

The rest of the proof is similar to the one for Algorithm 1, noting that $\mathrm{R_X}$ and $\mathrm{W_X}$ work as the inverses of $\mathrm{R_D}$ and $\mathrm{W_D}$, respectively. The only major difference is handling the difference operations in line 10. This follows from Proposition 1.

□

Theorem 3 (Correctness of Algorithm 3)**.** *Let $S_i^{t_w}$ be the value for process P_i at line 9. Then $\bigcup_i Q_i(S_i^{t_w}) = S^{t_w}$.*

Proof Sketch. The proof uses many of the techniques from the correctness of Algorithms 1 and 2. Again, we can show that the algorithm halts by proving that TRYRECEIVE(t_w) at line 8 will eventually be successful if $t_w \bmod k = 0$ using induction. Assuming that TRYRECEIVE(t) has succeeded for all processes for $t < t_w$ such that $t \bmod k = 0$, then all m layers of replicated are updated to $t_w - k$. Since $m \geq k - 1$, Q is able to proceed to t_w without receiving *any* messages in between. So line 5 is guaranteed to execute.

The rest of the proof is also similar to the one for Algorithm 1. In particular, we again make use of Proposition 1 to combine the replicated layers. □

A.2 Application Pseudocode

A.2.1 Jacobi Pseudocode

As mentioned previously in Section 2.6.1, we consider the prototypical problem of solving a steady-state heat diffusion problem using a regular 2D mesh. To solve this problem by Jacobi iteration, each grid point needs to communicate its heat values to its four spatial neighbors at each step.

It is easy to abstract this style of computation in our programming model. Function PART creates a block partitioning of the original matrix:

```
List<Query> PART(int n) {
```

```
  File globalState = getGlobalStateFromCkpt();
  List<BlockBoundary> bbs =
        blockPartitionMatrix(globalState,n);
  List<BlockQuery> queries = getBlockQuery(bbs);
  return queries;
}
```

Each block query only needs to represent the ranges of indexes that define the block. Applying proper dependencies of such block query to the NEW function yields a submatrix for the corresponding block, which is stored locally to a process.

The computation of a STEP is straightforward and is therefore omitted. We iterate over the cells of the matrix block given as input and execute the standard heat diffusion. Again, the runtime can only generate correct calls to STEP if it can calculate an appropriate context. So the developer must specify dependency functions. As the structure of the matrix does not change during computation, W_D and W_X are just identity. Functions R_D and R_X return queries that obtain the cells in the neighborhood of the query given as input.

```
Query RD(Query q) {
  MatrixRange m = (MatrixRange) q;
  return new MatrixRange(m.lowLeftX() - 1, m.lowLeftY() - 1,
                  m.upperRightX() + 1, m.upperRightY() + 1);
}

Query RX(Query q) {
  MatrixRange m = (MatrixRange) q;
```

```
  return new MatrixRange(m.lowLeftX() + 1, m.lowLeftY() + 1,
                  m.upperRightX() - 1, m.upperRightY() - 1);
}
```

These queries either enlarge or shrink the matrix range by one in each direction.

A.2.2 PageRank Pseudocode

As observed previously in Google's Pregel framework, many graph computations are easily expressible as time-stepped applications [98]. In the following, we show how to express PageRank in our programming model.

We first observe that the graph structure itself does not change during the computation of PageRank. So we can compute a partitioning of the graph at the start, e.g., reusing a well-known graph partitioning toolkit such as METIS [82], and use this partitioning throughout computation. The corresponding PART function is shown as follows:

```
List<Query> PART(int n) {
  File globalState = readGlobalStateFromCkpt();
  PartitionMap pm = callMETIS(globalState,n);
  List<PartitionQuery> queries = getPartitionQueries(pm);
  for (PartitionQuery pq in queries) {
    // precompute labels
    pq.labelPartition(globalState);
  }
  return queries;
```

```
}
```

When we call METIS, we also label each vertex in the state with a special attribute, its partition number. A partition query returns all vertices with a given partition number. PART not only invokes METIS, but also performs some precomputation on the vertices for performance. In particular, we label the boundary vertices of a partition with value 0. Every other vertex inside the partition gets label i if it only has incoming edges from vertices labeled $j \geq i - 1$, and every vertex outside the partition gets label i if it has outgoing edges to vertices labeled $j = i + 1$.

This precomputation allows us to determine dependency relationships more efficiently at runtime by encoding queries on labels and on partition numbers.

The STEP function is the familiar PageRank computation, with context containing all neighbors of vertices in the input set:

```
State STEP(State toStep, State context) {
  State result = getGraph();
  for (Vertex v in toStep) {
    Vertex v' = result.getVertex(v);
    v'.rank = 0.0;
    for (Vertex u in context, u directed to v) {
      // compute contribution of u to v
      v'.rank += u.rank / u.outDegree
  }}
  return result;
}
```

Our runtime needs to ensure only correct applications of function STEP. For this, the developer only needs to provide specifications of the data dependency functions. As in the Jacobi example, the graph structure remains unchanged during computation, and thus functions W_D and W_X are again identity. Queries obtain vertex sets inside and across partitions according to the partition number and the labels inside partitions. Given that these labels are assigned in the precomputation done by function PART, we can express functions R_D and R_X as queries on these labels:

```
Query RD(Query q) {
  // get incoming neighbors in same partition
  Query rdQuery = new LabelQuery(q.label() - 1);
  return rdQuery;
}

Query RX(Query q) {
  Query rxQuery = new LabelQuery(q.label() + 1);
  return rxQuery;
}
```

Function R_D expands the current vertex set by obtaining all vertices with labels smaller by one. As with the Jacobi example, these queries expand or contract the corresponding vertex sets by one hop. Function R_X operates only on partition local data, selecting vertices with label greater by one.

APPENDIX B

APPENDIX FOR CHAPTER 4

B.1 Problem Complexity

Theorem 4. *The Longest-Link Node Deployment Problem (LLNDP) is NP-hard.*

Proof. We reduce the *Subgraph Isomorphism Problem* (SIP), which is known to be NP-Hard [57], to the LLNDP. Consider an instance of SIP, where $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$, and we look for a mapping $\sigma : V_1 \rightarrow V_2$ such that whenever $(i, j) \in E_1$, then $(\sigma(i), \sigma(j)) \in E_2$. We build an instance of the LLNDP as follows. We set $N = V_1$, $S = V_2$, $G = (V_1, E_1)$, and the costs $C_{\mathcal{L}}(i, j)$ to 1 whenever the edge (i, j) belongs to E_2, and to 2 otherwise. By solving LLNDP, we get a deployment plan $\mathcal{D}$. G_2 contains a subgraph isomorphic to G_1 whenever $C^{\mathsf{LL}}_{\mathcal{D}} = 1$ and $\sigma = \mathcal{D}$. □

In order to show hardness of approximation, we will assume in the next two theorems that all communication costs are distinct. This assumption is fairly realistic, even more so as these costs are typically real numbers that are experimentally measured.

Theorem 5. *There is no α-absolute approximation algorithm to the Longest-Link Node Deployment Problem in the case where all communication costs are distinct, unless* P=NP.

Proof. Consider an instance I of the LLNDP, consisting of $G = (N, E)$, $C_{\mathcal{L}}(i, j)$ where $i, j \in S$, and $C_{\mathcal{L}}(i, j) = C_{\mathcal{L}}(i', j')$ if and only if $i = i'$ and $j = j'$. We order all the communication links (i_1, j_1), (i_2, j_2) ..., $(i_{|N|^2}, j_{|N|^2})$ in increasing order of

their communication costs. Let (i_w, j_w) be the Longest Link used in the optimal solution for I, with optimal value $C_{\mathcal{L}}(i_w, j_w)$. Notice that any instance of LLNDP that shares the same G and S as well as the same exact ordering of the communication links will also have an optimal value equal to $C_{\mathcal{L}}(i_w, j_w)$. Now, assume that there is an α-absolute approximation algorithm $\mathcal{A}$ to LLNDP. We build an instance I' by changing the communication costs in I to $C_{\mathcal{L}}(i_k, j_k) = (\alpha+1)k$. Note that I and I' share the same ordering of the communication links, and any two links in I' have communication costs that are at least $\alpha+1$ apart. Since $\mathcal{A}$ returns a longest link $(i_{w'}, j_{w'})$ for I' such that $C_{\mathcal{L}}(i_{w'}, j_{w'}) \leq C_{\mathcal{L}}(i_w, j_w)+\alpha$, $\mathcal{A}$ actually solves I' optimally. The fact that LLNDP is NP-hard completes the proof. □

Theorem 6. *There is no ϵ-relative approximation algorithm to the Longest-Link Node Deployment Problem in the case where all costs are distinct, unless* P=NP.

Proof. As in the previous proof, we build an instance I' that differs from I only by the costs of the links. We set these costs to be $C_{\mathcal{L}}(i_k, j_k) = (\epsilon + 1)^k$ for every link (i, j) where $i, j \in S$. In that case, for any two links (i_p, j_p) and (i_q, j_q) where $p < q$, we have $C_{\mathcal{L}}(i_p, j_p) < \epsilon \cdot C_{\mathcal{L}}(i_q, j_q)$. The fact that a ϵ-relative approximation algorithm would return a longest link $(i_{w'}, j_{w'})$ of I' such that $C_{\mathcal{L}}(i_{w'}, j_{w'}) \leq \epsilon \cdot C_{\mathcal{L}}(i_w, j_w)$ completes the proof. □

Theorem 7. *The Longest-Path Node Deployment Problem (LPNDP) is NP-hard.*

Proof. The proof is otherwise identical to the proof of Theorem 4 except when (i, j) does not belong to E_2, we set $C_{\mathcal{L}}(i, j)$ to $|E_1| + 1$ and the final check is now $C^{\mathsf{LP}}_{\mathcal{D}} \leq |E_1|$. □

B.2 Distance Approximations

All techniques in Section 3.5 may require non-negligible measurement time to obtain pairwise mean latencies. We also experimented with the following two approximations to network distance, which are both simple and intuitively related to mean latency.

1. IP Distance. Our first approximation makes use of internal IPv4 addresses in the cloud. The hypothesis is that if two instances share a common /24 address prefix, then these instances are more likely to be located in the same or in a nearby rack than if the two instances only share a common /8 prefix. We can therefore define *IP distance* as a measure of the dissimilarity between two IP addresses: Two instances sharing the same /x address prefix but not /x+1 address prefix have IP distance 32 - x. We can future adjust the sensitive of this measurement by considering $g (1 \leq g < 32)$ consecutive bits of the IP addresses together.

2. Hop Count. A slightly more sophisticated approximation is the hop count between two instances. The hop count is the number of intermediate routers through which data flows from source to destination. Hop count can be obtained by sending packets from source to destination and monitoring the Time To Live (TTL) field of the IP header.

Experimental Evaluation. We evaluated the two approximations above with the same experimental setup described in Section 3.6.2. We compare both IP distance and hop count against the mean round-trip latency measurement results obtained using the *staged* approach described in Section 3.5.

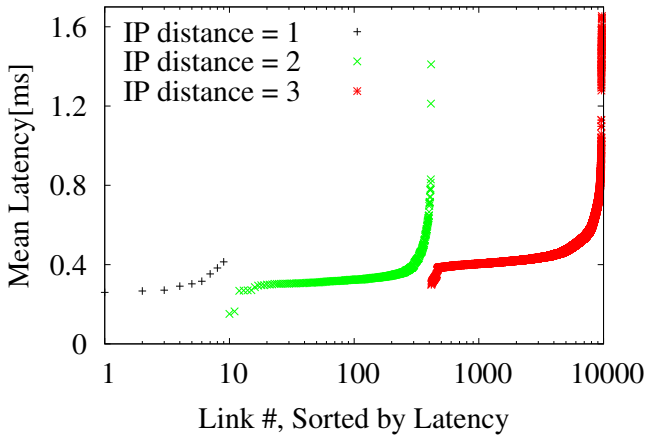


Figure B.1: Latency order by IP distance

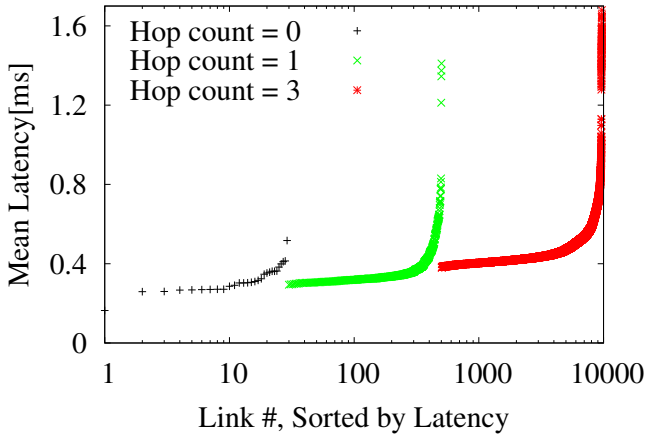


Figure B.2: Latency order by Hop Count

In Figure B.1, we show the effect of using IP distances as an approximation. In this experiment, we consider 8 consecutive bits of the IP address together: two instances sharing a /24 address prefix have IP distance 1; two instances with the same /16 prefix but not /24 prefix have IP distance 2, and so on. We also experimented with other sensitivity configurations and the results are similar. The x axis is in log scale and divided into 3 groups based on the value of IP distance. Since we used the internal IP addresses of EC2, all of which share

the same /8 prefix, we do not observe IP distance of 4. Within each group, links are ordered by round-trip latency measurements. If IP distance were a desirable approximation, we would expect that pairs with higher IP distance would also have higher latencies. Figure B.1 shows that such monotonicity does not always hold. For example, within the group of IP distance = 2, there exist links having lower latency than some links of IP distance = 1, as well as links having higher latency than some links of IP distance= 3. Interestingly, the lowest latencies are observed in pairs with IP distance= 2.

The effect of using hop count as an approximation is shown in Figure B.2. Similarly, the x axis is in log scale and divided into 3 groups based on hop count. We do not observe any pair of instances that are 2 hops apart. Within each group, links are ordered by round-trip latencies. As in Figure B.1, there exists a significant number of link pairs ordered inconsistently by hop count and measured latency.

The above results demonstrate that IP distance and hop count, though easy to obtain, do not effectively predict network latency.

B.3 Public Cloud Service Providers

To demonstrate the applicability of ClouDiA in public clouds other than the one offered by Amazon Web Services, we report latency heterogeneity and mean latency stability measurements in Google Compute Engine and Rackspace Cloud Server.

Figure B.3 shows the CDF of the mean pairwise end-to-end latencies among

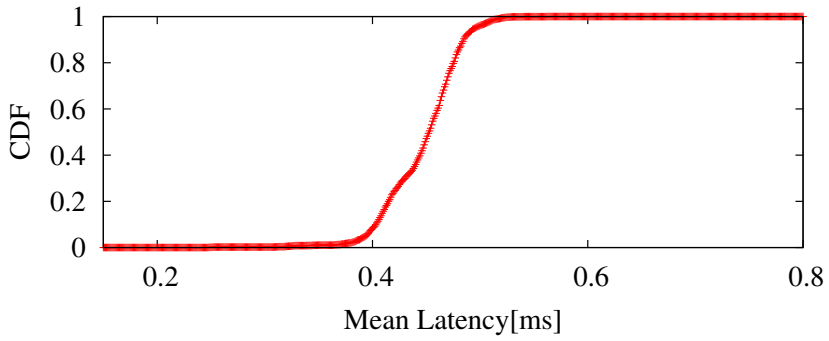


Figure B.3: Latency Heterogeneity in Google Compute Engine

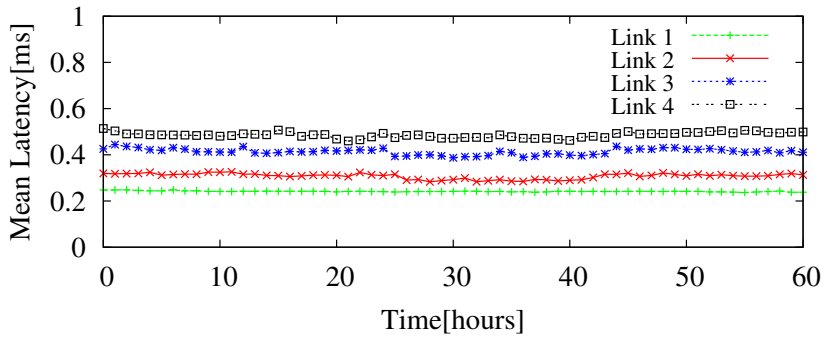


Figure B.4: Mean Latency Stability in Google Compute Engine

50 Google Compute Engine n1-standard-1 instances in the us-central1-a region, obtained by measuring TCP round-trip times of 1KB messages. Around 5% of the instance pairs exhibit mean latency below 0.32 ms, whereas the top 5% are above 0.5 ms. Figure B.4 shows the mean latencies of four representative links over 60 hours, with each latency measurement averaged over an hour. We observe a similar behavior of mean latency stability over time as in Amazon EC2. By contrast, latency heterogeneity is somewhat smaller; however, it is still present.

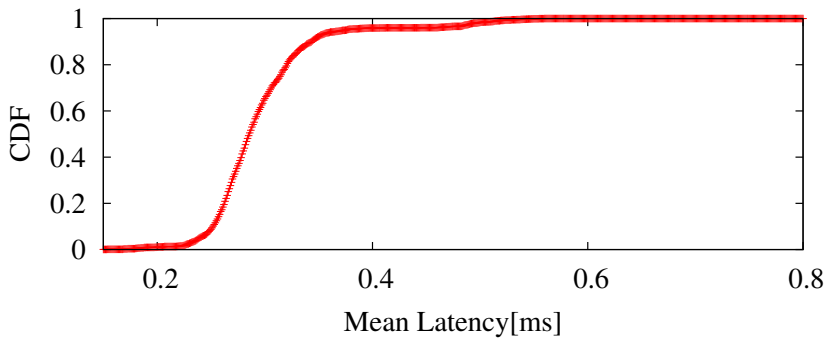


Figure B.5: Latency Heterogeneity in Rackspace Cloud Server

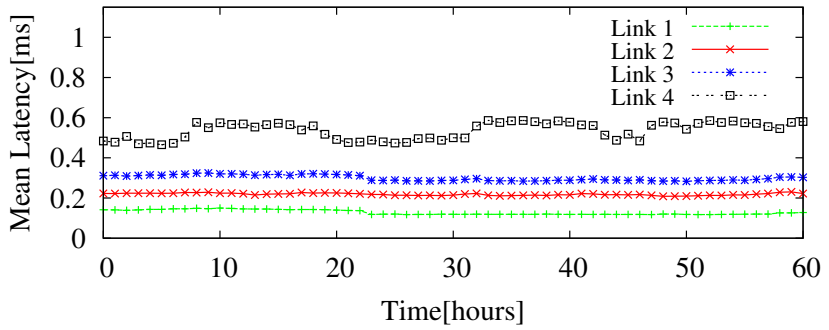


Figure B.6: Mean Latency Stability in Rackspace Cloud Server

Similarly, Figure B.5 shows the CDF of the mean pairwise end-to-end latencies among 50 Rackspace Cloud Server performance1-1 instances in the Northern Virginia (IAD) region, obtained by measuring TCP round-trip times of 1KB messages. Around 5% of the instance pairs have latency below 0.24 ms, whereas the top 5% are above 0.38 ms. Figure B.6 shows the mean latencies of four representative links over 60 hours, with latency measurement averaged over an hour. The effects observed are largely in line with the ones seen in the Google Compute Cloud.

The above results confirm the existence of latency heterogeneity and mean latency stability in the public clouds of both Google Compute Engine and Rackspace Cloud Server. The results suggest that by adopting ClouDiA, cloud tenants can achieve significant reduction in time-to-solution or service response time in these public clouds as well, and not only on Amazon EC2.

BIBLIOGRAPHY

[1] Rakesh Agrawal, Michael J. Carey, and Miron Livny. Concurrency control performance modeling: alternatives and implications. *ACM Trans. Database Syst.*, 12(4):609–654, November 1987.

[2] Richard D. Alpert and James F. Philbin. cBSP: Zero-cost synchronization in a modified BSP model. Technical report, NEC Research Institute, 1997.

[3] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. Consistency Analysis in Bloom: a CALM and Collected Approach. In *CIDR*, 2011.

[4] Amazon Web Services, Elastic Compute Cloud (EC2). http://aws.amazon.com/ec2/.

[5] Amazon Elastic Compute Cloud (EC2) pricing. http://aws.amazon.com/ec2/pricing/.

[6] Amazon Simple Storage Service (S3). http://aws.amazon.com/s3.

[7] U ang, C.E Tsourakakis, Ana Paula Appel, C. Faloutsos, and Jure Leskovec. HADI: Mining radii of large graphs. *TKDD*, 2010.

[8] Amazon web services, case studies. http://aws.amazon.com/solutions/case-studies/.

[9] Amazon web services, cloudformation. http://aws.amazon.com/cloudformation/.

[10] Amazon web services, search engines & web crawlers. http://aws.amazon.com/search-engines/.

[11] Yossi Azar, Ilan Reuven Cohen, Seny Kamara, and Bruce Shepherd. Tight bounds for online vector bin packing. In *STOC*, 2013.

[12] Shivnath Babu. Towards automatic optimization of mapreduce programs. In *SOCC*, 2010.

[13] Claudine Santos Badue, Ricardo A. Baeza-Yates, Berthier A. Ribeiro-Neto, and Nivio Ziviani. Distributed query processing using partitioned inverted files. In *SPIRE*, pages 10–20, 2001.

[14] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Antony I. T. Rowstron. Towards predictable datacenter networks. In *SIGCOMM*, 2011.

[15] Mike Barnett, Badrish Chandramouli, Robert DeLine, Steven M. Drucker, Danyel Fisher, Jonathan Goldstein, Patrick Morrison, and John C. Platt. Stat!: an interactive analytics environment for big data. In *SIGMOD*, 2013.

[16] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[17] Dominic Battré, Natalia Frejnik, Siddhant Goel, Odej Kao, and Daniel Warneke. Evaluation of network topology inference in opaque compute clouds through end-to-end measurements. In *IEEE CLOUD*, 2011.

[18] Doina Bein, Wolfgang W. Bein, and Swathi Venigella. Cloud storage and online bin packing. In *IDC*, volume 382 of *Studies in Computational Intelligence*, pages 63–68, 2011.

[19] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *Internet Measurement Conference*, 2010.

[20] Arka A. Bhattacharya, David Culler, Eric Friedman, Ali Ghodsi, Scott Shenker, and Ion Stoica. Hierarchical scheduling for diverse datacenter workloads. In *SOCC*, 2013.

[21] G. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39:85–97, 1996.

[22] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *FOCS*, pages 356–368, 1994.

[23] Olaf Bonorden, Ben Juurlink, Ingo von Otte, and Ingo Rieping. The paderborn university BSP (PUB) library. *Parallel Computing*, 29(2):187–207, 2003.

[24] T. Brandes. Evaluation of high-performance fortran on some real applications. In *Proc. HPCN*, 1994.

[25] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1-7):107–117, 1998.

[26] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael Ernst. HaLoop: Efficient iterative data processing on large clusters. *PVLDB*, 3(1-2):285–296, 2010.

[27] Ugur Cetintemel, Mitch Cherniack, Justin DeBrabant, Yanlei Diao, Kyriaki Dimitriadou, Alex Kalinin, Olga Papaemmanouil, and Stan Zdonik. Query Steering for Interactive Data Exploration. In *CDIR*, 2013.

[28] Sivadon Chaisiri, Bu-Sung Lee, and Dusit Niyato. Optimization of resource provisioning cost in cloud computing. *IEEE T. Services Computing*, 5(2):164–177, 2012.

[29] K. Mani Chandy and Jayadev Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Commun. ACM*, 24:198–206, 1981.

[30] Surajit Chaudhuri and Vivek Narasayya. An efficient, cost-driven index selection tool for microsoft SQL server. In *VLDB*, 1997.

[31] Rishan Chen, Xuetian Weng, Bingsheng He, and Mao Yang. Large graph processing in the cloud. In *SIGMOD*, 2010.

[32] Yanpei Chen, Sara Alspaugh, and Randy Katz. Interactive analytical processing in big data systems: a cross-industry study of mapreduce workloads. *PVLDB*, 5(12), 2012.

[33] Xiang Cheng, Sen Su, Zhongbao Zhang, Hanchi Wang, Fangchun Yang, Yan Luo, and Jie Wang. Virtual network embedding through topology-aware node ranking. *SIGCOMM CCR*, 41(2):38–47, 2011.

[34] C. Choudhury, T. Toledo, and M. Ben-Akiva. NGSIM freeway lane selection model. Technical report, Federal Highway Administration, 2004. FHWA-HOP-06-103.

[35] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I. Jordan, and Ion Stoica. Managing data transfers in computer clusters with orchestra. In *SIGCOMM*, 2011.

[36] N. M. M. K. Chowdhury, M. R. Rahman, and R. Boutaba. Virtual Network Embedding with Coordinated Node and Link Mapping. In *INFOCOM*, 2009.

[37] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on PAMI*, 26:1367–1372, 2004.

[38] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. Performance evaluation of the VF graph matching algorithm. In *International Conference on Image Analysis and Processing*, 1999.

[39] I. Couzin, J. Krause, N. Franks, and S. Levin. Effective leadership and decision-making in animal groups on the move. 433(7025):513–516, 2005.

[40] I. Couzin, J. Krause, N. Franks, and S. Levin. Effective leadership and decision-making in animal groups on the move. *Nature*, 433(7025):513–516, 2005.

[41] J. Csirik. An on-line algorithm for variable-sized bin packing. *Acta Informatica*, 26(8):697–709, 1989.

[42] DARPA high productivity computing systems project. http://www.highproductivity.org/.

[43] Raja Das, Yuan shin Hwang, Mustafa Uysal, Joel Saltz, and Alan Sussman. Applying the CHAOS/PARTI library to irregular problems in computational chemistry and computational aerodynamics. In *Proceedings of the 1993 Scalable Parallel Libraries Conference*, pages 45–56, 1993.

[44] Sudipto Das, Vivek Narasayya, Feng Li, and Manoj Syamala. Cpu sharing techniques for performance isolation in multi-tenant relational database-as-a-service. In *PVLDB*, 2013.

[45] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[46] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

[47] James Demmel, Mark Hoemmen, Marghoob Mohiyuddin, and Katherine A. Yelick. Avoiding communication in sparse matrix computations. In *IPDPS*, pages 1–12. IEEE, 2008.

[48] David Eppstein. Subgraph isomorphism in planar graphs and related problems. In *SODA*, 1995.

[49] Epstein. Online bin packing with cardinality constraints. *SIJDM: SIAM Journal on Discrete Mathematics*, 20, 2006.

[50] Constantinos Evangelinos and Chris N. Hill. Cloud computing for parallel scientific hpc applications. In *CCA*, 2008.

[51] Ilhem Fajjari, Nadjib Aitsaadi, Guy Pujolle, and Hubert Zimmermann. VNE-AC: Virtual network embedding algorithm based on ant colony metaheuristic. In *IEEE International Conference on Communications*, 2011.

[52] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller 0002, Hannes Rauhe, and Jonathan Dees. The SAP HANA database–an architecture overview. *IEEE Data Eng. Bull*, 35(1):28–33, 2012.

[53] Benjamin Farley, Venkatanathan Varadarajan, Kevin Bowers, Ari Juels, Thomas Ristenpart, and Michael Swift. More for your money: Exploiting performance heterogeneity in public clouds. In *SOCC*, 2012.

[54] Michael J. Feeley and Henry M. Levy. Distributed shared memory with versioned objects. In *OOPSLA*, 1992.

[55] Games and Simulations Project, Cornell Database Group Website. http://www.cs.cornell.edu/bigreddata/games.

[56] Michael R. Garey, Ronald L. Graham, David S. Johnson, and A. C. C. Yao. Resource constrained scheduling as generalized bin packing. *J. Combin. Theory Ser. A*, 21:257–298, 1976.

[57] Michael R. Garey and David S. Johnson. *Computers and intractability*. Freeman, 1979.

[58] Roxana Geambasu, Steven D. Gribble, and Henry M. Levy. Cloudviews: Communal data sharing in public clouds. In *HotCloud*, 2009.

[59] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *SOSP*, 2003.

[60] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott

Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, 2011.

[61] Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Choosy: Max-min fair sharing for datacenter jobs with constraints. In *EuroSys*, 2013.

[62] Google compute engine. https://cloud.google.com/products/compute-engine.

[63] William Gropp. Learning from the success of mpi. In *HiPC*, 2001.

[64] Chuanxiong Guo, Guohan Lu, Helen J. Wang, Shuang Yang, Chao Kong, Peng Sun, Wenfei Wu, and Yongguang Zhang. Secondnet: a data center network virtualization architecture with bandwidth guarantees. In *CoNEXT*, 2010.

[65] D. Gupta, L. Cherkasova, R. Gardner, and A.Vahdat. Enforcing performance isolation across virtual machines in xen. In *Middleware*, 2006.

[66] Hadoop. http://hadoop.apache.org/.

[67] Junghee Han, David Watson, and Farnam Jahanian. Topology aware overlay networks. In *INFOCOM*. IEEE, 2005.

[68] Per Brinch Hansen. An evaluation of the message-passing interface. *SIGPLAN Not.*, 33:65–72, March 1998.

[69] Hellerstein, Avnur, Chou, Hidber, Olston, Raman, Roth, and Haas. Interactive data analysis: The control project. *COMPUTER: IEEE Computer*, 32, 1999.

[70] Bruce Hendrickson and Karen Devine. Dynamic load balancing in computational mechanics. *Computer Methods in Applied Mechanics and Engineering*, 184(2-4):485–500, 2000.

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

[72] HP intelligent management center virtual application network

manager. http://h17007.www1.hp.com/us/en/products/network-management/IMC_VANM_Software/index.aspx.

[73] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*, 2010.

[74] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: fair scheduling for distributed computing clusters. In *SOSP*, 2009.

[75] Eaman Jahani, Michael J. Cafarella, and Christopher Ré. Automatic optimization for mapreduce programs. *PVLDB*, 2011.

[76] David R. Jefferson, Brian Beckman, Frederick Wieland, Leo Blume, Mike Di Loreto, Phil Hontalas, Pierre Laroche, Kathy Sturdevant, Jack Tupman, L. Van Warren, John J. Wedel, Herb Younger, and Steve Bellenot. Distributed simulation and the time warp operating system. In *SOSP*, 1987.

[77] Frederico Alvares De Oliveira Jr. and Thomas Ledoux. Self-management of applications qoS for energy optimization in datacenters. In *2nd International Workshop on Green Computing Middleware*, 2011.

[78] Gideon Juve, Ewa Deelman, Karan Vahi, Gaurang Mehta, Bruce Berriman, Benjamin P. Berman, and Phil Maechling. Scientific workflow applications on amazon EC2. In *IEEE International Conference on e-Science.*, 2009.

[79] Sepandar D. Kamvar, Taher H. Haveliwala, Christopher D. Manning, and Gene H. Golub. Extrapolation methods for accelerating the computation of pagerank. In *WWW*, 2003.

[80] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. PEGASUS: A peta-scale graph mining system. In *ICDM*, 2009.

[81] Nicholas T. Karonis, Brian R. Toonen, and Ian T. Foster. MPICH-G2: A grid-enabled implementation of the message passing interface. *J. Parallel Distrib. Comput.*, 63(5):551–563, 2003.

[82] George Karypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *J. Parallel Distrib. Comput.*, 48(1):96–129, 1998.

[83] Jin-Soo Kim, Soonhoi Ha, and Chu Shik Jhon. Efficient barrier synchronization mechanism for the BSP model on message-passing architectures. In *IPPS/SPDP*, 1998.

[84] P. Krishnan, Danny Raz, and Yuval Shavitt. The cache location problem. *IEEE/ACM Transactions on Networking*, 8(5):568–582, 2000.

[85] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS OSR*, 44(2):35–40, 2010.

[86] Javier Larrosa and Gabriel Valiente. Constraint satisfaction algorithms for graph pattern matching. *Mathematical. Structures in Comp. Sci.*, 12(4):403–422, August 2002.

[87] Chris P. Lee, Gene H. Golub, and Stefanos A. Zenios. A two-stage algorithm for computing pagerank and multistage generalizations. *Internet Mathematics*, 4(4):299–327, 2007.

[88] Gunho Lee, Niraj Tolia, Parthasarathy Ranganathan, and Randy H. Katz. Topology-aware resource allocation for data-intensive workloads. *SIGCOMM CCR*, 2011.

[89] Jure Leskovec, Jon M. Kleinberg, and Christos Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *KDD*, pages 177–187, 2005.

[90] Bo Li, M. J. Golin, G. F. Italiano, Xin Deng, and K. Sohraby. On the optimal placement of web proxies in the internet. In *INFOCOM*, 1999.

[91] Jiexing Li, Arnd Christian König, Vivek R. Narasayya, and Surajit Chaudhuri. Robust estimation of resource consumption for SQL queries using statistical techniques. In *VLDB*, 2012.

[92] Frank Lin and William W. Cohen. Semi-supervised classification of network data using very few labels. In *ASONAM*, 2010.

[93] Michael J. Litzkow, Miron Livny, and Matt W. Mutka. Condor - A hunter of idle workstations. In *ICDCS*, pages 104–111, 1988.

[94] Dionysios Logothetis, Christopher Olston, Benjamin Reed, Kevin C. Webb, and Ken Yocum. Stateful bulk processing for incremental analytics. In *SoCC*, pages 51–62, 2010.

[95] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. GraphLab: A new parallel framework for machine learning. In *UAI*, 2010.

[96] Ewing L. Lusk and Katherine A. Yelick. Languages for high-productivity computing: the DARPA HPCS language project. *Parallel Processing Letters*, 17(1):89–102, 2007.

[97] David Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.

[98] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. 2010.

[99] David Meisner and Thomas F. Wenisch. Stochastic queuing simulation for data center workloads. In *Proc. of the Workshop on Exascale Evaluation and Research Techniques*, 2010.

[100] Xiaoqiao Meng, Vasileios Pappas, and Li Zhang. Improving the scalability of data center networks with traffic-aware virtual machine placement. In *INFOCOM*, 2010.

[101] Gary L. Miller. Isomorphism testing for graphs of bounded genus. In *STOC*, 1980.

[102] Z. Miller and James B. Orlin. Np-completeness for minimizing maximum edge length in grid embeddings. *J. Algorithms*, pages 10–16, 1985.

[103] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. Portland: a scalable fault-tolerant layer 2 data center network fabric. In *SIGCOMM*, 2009.

[104] Vivek Narasayya, Sudipto Das, Manoj Syamala, Badrish Chandramouli, and Surajit Chaudhuri. Sqlvm: Performance isolation in multi-tenant relational database-as-a-service. In *CIDR*, 2013.

[105] Lars S. Nyland, Jan Prins, Ru Huai Yun, Jan Hermans, Hye-Chung Kum, and Lei Wang. Modeling dynamic load balancing in molecular dynamics to achieve scalable parallel execution. In *IRREGULAR*, pages 356–365, 1998.

[106] Charlene O'Hanlon. A conversation with werner vogels. *ACM Queue*, 4(4):14–22, May 2006.

[107] Christopher Olston, Edward Bortnikov, Khaled Elmeleegy, Flavio Junqueira, and Benjamin Reed. Interactive analysis of web-scale data. In *CIDR*, 2009.

[108] Arthur O'Sullivan and Steven M. Sheffrin. *Economics: Principles in Action*. Pearson Prentice Hall, 2003.

[109] Balaji Palanisamy, Aameek Singh, Ling Liu, and Bhushan Jain. Purlieus: locality-aware resource allocation for mapreduce in a cloud. In *SC*, 2011.

[110] Olga Papaemmanouil, Yanif Ahmad, Uğur Çetintemel, John Jannotti, and Yenel Yildirim. Extensible optimization in overlay dissemination trees. In *SIGMOD*, 2006.

[111] Peter R. Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo I. Seltzer. Network-aware operator placement for stream-processing systems. In *ICDE*, 2006.

[112] Lili Qiu, Venkata N. Padmanabhan, and Geoffrey M. Voelker. On the placement of web server replicas. In *INFOCOM*, 2001.

[113] Rackspace cloud servers. www.rackspace.com/cloud/servers/.

[114] Anshul Rai, Ranjita Bhagwan, and Saikat Guha. Generalized resource allocation for the cloud. In *SOCC*, 2012.

[115] Lavanya Ramakrishnan, Keith R. Jackson, Shane Canon, Shreyas Cholia, and John Shalf. Defining future platform requirements for e-Science clouds. In *SoCC*, pages 101–106, 2010.

[116] Raghu Ramakrishnan. Data serving in the cloud. In *LADIS*, 2010.

[117] Rajesh Raman, Miron Livny, and Marvin H. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *HPDC*, page 140, 1998.

[118] Vijayshankar Raman, Gopi K. Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, René Müller, Ippokratis

Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam J. Storm, and Liping Zhang. DB2 with BLU acceleration: So much more than just a column store. *PVLDB*, 6(11):1080–1091, 2013.

[119] Venugopalan Ramasubramanian, Dahlia Malkhi, Fabian Kuhn, Mahesh Balakrishnan, Archit Gupta, and Aditya Akella. On the treeness of internet latency and bandwidth. In *SIGMETRICS*, 2009.

[120] Robert Ricci, Chris Alfeld, and Jay Lepreau. A solver for the network testbed mapping problem. *SIGCOMM CCR*, 33(2):65–81, 2003.

[121] Lincoln Uyeda Rina Panigrahy, Kunal Talwar and Udi Wieder. Heuristics for vector bin packing. Unpublished manuscript, 2011.

[122] Sabyasachi Roy, Himabindu Pucha, Zheng Zhang, Y. Charlie Hu, and Lili Qiu. Overlay node placement: Analysis, algorithms and impact on applications. In *ICDCS*, 2007.

[123] Running shark on ec2. https://github.com/amplab/shark/wiki/Running-Shark-on-EC2.

[124] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Comp. Surv.*, 16(2):187–260, 1984.

[125] José Carlos Sancho, Kevin J. Barker, Darren J. Kerbyson, and Kei Davis. Quantifying the potential benefit of overlapping communication and computation in large-scale scientific applications. In *SC*, 2006.

[126] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. Runtime measurements in the cloud: Observing, analyzing, and reducing variance. *PVLDB*, 3(1):460–471, 2010.

[127] Jan Schaffner, Benjamin Eckart, Dean Jacobs, Christian Schwarz, Hasso Plattner, and Alexander Zeier. Predicting in-memory database performance for automating cluster management tasks. In *ICDE*, 2011.

[128] J. T. Schwartz, R. B. Dewar, E. Schonberg, and E. Dubinsky. *Programming with sets; an introduction to SETL*. Springer-Verlag New York, Inc., 1986.

[129] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *EuroSys*, 2013.

[130] Seiden. An optimal online algorithm for bounded space variable-sized bin packing. *SIJDM: SIAM Journal on Discrete Mathematics*, 14, 2001.

[131] Alan Shieh, Srikanth Kandula, Albert Greenberg, and Changhoon Kim. Seawall: Performance isolation for cloud datacenter networks. In *HotCloud*, 2010.

[132] David Shue, Michael J. Freedman, and Anees Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *OSDI*, 2012.

[133] Yee Jiun Song, Marcos Kawazoe Aguilera, Ramakrishna Kotla, and Dahlia Malkhi. RPC chains: Efficient client-server communication in geodistributed systems. In *NSDI*, 2009.

[134] Garrick Staples. TORQUE - TORQUE resource manager. In *SC*, page 8, 2006.

[135] The Capacity Scheduler of Hadoop 1.2.1. http://hadoop.apache.org/docs/r1.2.1/capacity_scheduler.html.

[136] The Fair Scheduler of Hadoop 1.2.1 . http://hadoop.apache.org/docs/r1.2.1/fair_scheduler.html.

[137] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM*, 23, 1976.

[138] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[139] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet another resource negotiator. In *SOCC*, 2013.

[140] Werner Vogels. Data access patterns in the amazon.com technology platform. In *VLDB*, page 1, 2007.

[141] Andrew Wang, Shivaram Venkataraman, Sara Alspaugh, Randy Katz, and Ion Stoica. Cake: Enabling high-level slos on shared storage systems. In *SOCC*, 2012.

[142] Guohui Wang and T. S. Eugene Ng. The impact of virtualization on network performance of Amazon EC2 data center. In *INFOCOM*, pages 1163–1171, 2010.

[143] Guozhang Wang, Marcos Antonio Vaz Salles, Benjamin Sowell, Xun Wang, Tuan Cao, Alan J. Demers, Johannes Gehrke, and Walker M. White. Behavioral simulations in mapreduce. *PVLDB*, 3(1):952–963, 2010.

[144] Gerhard Weikum, Christof Hasse, Alex Moenkeberg, and Peter Zabback. The COMFORT automatic tuning project, invited project review. *Inf. Syst.*, 19(5), 1994.

[145] Y. Wen. *Scalability of Dynamic Traffic Assignment*. PhD thesis, Massachusetts Institute of Technology, 2008.

[146] Alexander Wieder, Pramod Bhatotia, Ansley Post, and Rodrigo Rodrigues. Orchestrating the deployment of computations in the cloud with conductor. In *NSDI*, 2012.

[147] Tom Wilson. Data mining user behavior. In *CMG MeasureIT*, 2010.

[148] Windows azure. http://www.windowsazure.com/.

[149] Joel L. Wolf, Deepak Rajan, Kirsten Hildrum, Rohit Khandekar, Vibhore Kumar, Sujay Parekh, Kun-Lung Wu, and Andrey Balmin. FLEX: A slot allocation scheduling optimizer for mapreduce workloads. In *Middleware*, pages 1–20, 2010.

[150] Petrie Wong, Zhian He, and Eric Lo. Parallel analytics as a service. In *SIGMOD*, 2013.

[151] R. Xin, J. Rosen, M. Zaharia, M. Franklin, S. Shenker, and I. Stoica. Shark: SQL and Rich Analytics at Scale. In *SIGMOD*, 2013.

[152] A. C.-C. Yao. New algorithms for bin packing. *J. ACM*, 27(2):207–227, 1980.

[153] Lamia Youseff, Keith Seymour, Haihang You, Jack Dongarra, and Richard Wolski. The impact of paravirtualized memory hierarchy on linear algebra computational kernels and software. In *HPDC*, pages 141–152, 2008.

[154] Lamia Youseff, Rich Wolski, Brent Gorda, and Chandra Krintz. Evaluating the performance impact of Xen on MPI and process execution for HPC systems. In *VTDC*, 2006.

[155] Lamia Youseff, Rich Wolski, Brent Gorda, and Chandra Krintz. Paravirtualization for HPC systems. In *ISPA Workshops*, pages 474–486, 2006.

[156] Minlan Yu, Yung Yi, Jennifer Rexford, and Mung Chiang. Rethinking virtual network embedding: substrate support for path splitting and migration. *SIGCOMM CCR*, 38(2):17–29, 2008.

[157] 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*, 2008.

[158] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*, pages 265–278, 2010.

[159] Matei Zaharia, Mosharaf Chowdhury, Michael Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing withworking sets. In *HotCloud*, 2010.

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

[161] Stéphane Zampelli, Yves Deville, and Christine Solnon. Solving subgraph isomorphism problems with constraint programming. *Constraints*, 15(3):327–353, 2010.

[162] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multi-core cache management. In *EuroSys*, 2009.

[163] Jianfeng Zhao, Wenhua Zeng, Min Liu, Guangming Li, and Min Liu. Multi-objective optimization model of virtual resources scheduling under cloud computing and it's solution. In *International Conference on Cloud and Service Computing*, 2011.

[164] Songnian Zhou. LSF: load sharing in large-scale heterogeneous dis-

tributed systems. In *Proceedings of the Workshop on Cluster Computing*, December 1992.

[165] Songnian Zhou, Xiaohu Zheng, Jingwen Wang, and Pierre Delisle. Utopia: a load sharing facility for large, heterogeneous distributed computer systems. *Softw. Pract. Exper.*, 23(12):1305–1336, December 1993.

[166] Tao Zou, Ronan Le Bras, Marcos Vaz Salles, Alan Demers, and Johannes Gehrke. Cloudia: A deployment advisor for public clouds. *PVLDB*, 6(2):109–120, 2012.

[167] Tao Zou, Guozhang Wang, Marcos Vaz Salles, David Bindel, Alan Demers, Johannes Gehrke, and Walker White. Making time-stepped applications tick in the cloud. In *SOCC*, 2011.