Making Time-stepped Applications Tick in the Cloud

Tao Zou[†], Guozhang Wang[†], Marcos Vaz Salles[‡*],
David Bindel[†], Alan Demers[†], Johannes Gehrke[†], Walker White[†]

[†]Cornell University
Ithaca, NY

[‡]University of Copenhagen
Copenhagen, Denmark

{taozou, guoz}@cs.cornell.edu, vmarcos@diku.dk,
{bindel, ademers, johannes, wmwhite}@cs.cornell.edu

ABSTRACT

Scientists are currently evaluating the cloud as a new platform. Many important scientific applications, however, perform poorly in the cloud. These applications proceed in highly parallel discrete time-steps or "ticks," using logical synchronization barriers at tick boundaries. We observe that network jitter in the cloud can severely increase the time required for communication in these applications, significantly increasing overall running time.

In this paper, we propose a general parallel framework to process time-stepped applications in the cloud. Our framework exposes a high-level, data-centric programming model which represents application state as tables and dependencies between states as queries over these tables. We design a jitter-tolerant runtime that uses these data dependencies to absorb latency spikes by (1) carefully scheduling computation and (2) replicating data and computation. Our data-driven approach is transparent to the scientist and requires little additional code. Our experiments show that our methods improve performance up to a factor of three for several typical time-stepped applications.

Categories and Subject Descriptors

H.2.4 [**Information Systems**]: Database Management—*Systems*

General Terms

Algorithms, Languages, Performance

Keywords

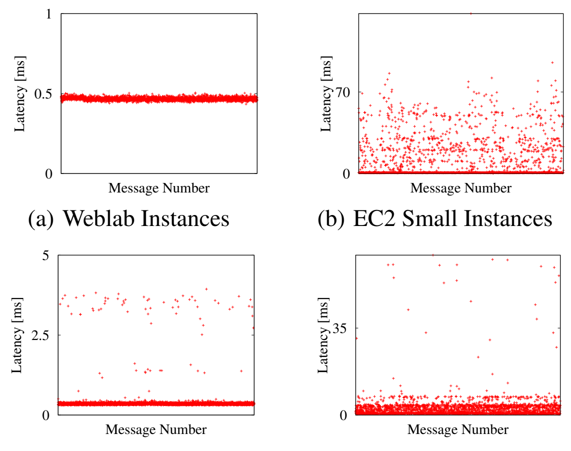
Parallel frameworks, database optimizations, cloud computing

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 [3, 8, 12, 14, 32]. They also include classic iterative methods for solving linear systems and eigenvalue problems [4]. 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 [44].

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 [39]. 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 paper examines what happens when we take these scientific applications 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 [19, 39]. 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 [47, 48, 49]. 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.



(a) Weblab Instances (b) EC2 Small Instances (c) EC2 Cluster Instances (d) EC2 Large Instances

Figure 1: Latency in Our Weblab Cluster and in EC2

*Work performed while author was at Cornell University.

One critical assumption which does not hold in the cloud is that there is a stable, low-latency interconnect among compute nodes. Recent experimental studies have demonstrated that the cloud suffers from high latency jitter [42, 45]. We have confirmed this observation by measuring the TCP round-trip times for 16 KB messages in several environments, as shown in Figure 1. 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. Note that the scales of the y-axes differ significantly. Communication in the Weblab 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 acceptable average latency, but suffer frequent latency "spikes" more than an order of magnitude above the mean. Even the cluster compute instances, advertised for HPC applications, show the same effect.

Time-stepped applications that are programmed in the bulk synchronous model suffer dramatically from this latency jitter. Suppose a message from process P_i to process P_j is delayed by a latency spike during global synchronization. P_j then blocks and cannot start its tick until it gets *unblocked* by the arriving message. If computational load is balanced, P_j will be late sending its own messages for the next tick. This in turn will create a *latency wave* across the cluster, which is hard to compensate for in parallel applications.

The HPC community has invested significant work in optimizing communication for time-stepped applications [1, 6, 29]. 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. [18] 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 Paper. In this paper 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 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 5). A formal description of our model and correctness proofs of our algorithms appear in Appendix A.

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 6).

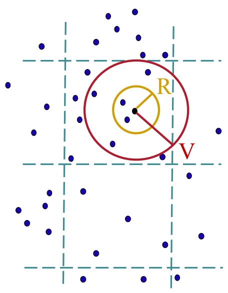
We start our presentation by defining time-stepped applications (Section 2) and summarizing our approach (Section 3).

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 [44]. The conceptual simplicity of this model has led to its widespread adoption by a large number of scientific applications [3, 4, 8, 12, 14, 32].

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 paper.

Running Example: Behavioral Simulations. Behavioral simulations model complex systems of individual, intelligent agents, such as transportation networks and animal swarms [12, 14]. 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 [46]. For instance, Couzin et al. used this type of simulation to study information transfer in schools of fish [14]. 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 [37] and iterative solvers [4], are time-stepped and therefore designed to exploit data parallelism and locality. We develop two additional examples of such applications in Appendixes B.1 and B.2. They correspond to an iterative method, Jacobi iteration [4], and a graph processing application, PageRank [8].

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 [1, 6, 29]. 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 programming 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 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 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 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 5.3) uses redundant data and its respective computation both to communicate less often *and* to unblock even more computation internal to a process. Hence, this technique can be used to complement dependency scheduling by providing additional flexibility at synchronization points.

Table 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.

4. PROGRAMMING MODEL

In this section, we describe the programming model offered by our jitter-tolerant runtime. Table 1 summarizes the functions we require application developers to instantiate. We explain them in detail in the following subsections.

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 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 [36]. Our techniques can be extended to multiple tables and structured attributes.

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. 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 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 [23, 40]. 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 $\text{GSTEP}(S^t) = \bigcup_{i=1}^{n} \text{GSTEP}(Q_i(S^t))$. This is because the correct computation of partition $Q_i(S)$ may require GSTEP to inspect data from other partitions as context.

To address this problem, we introduce two more functions: a *local initialization function* $\text{NEW}(Q)$ and a *local update function* $\text{STEP}(A,B)$. The local initialization function $\text{NEW}(Q)$ 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 5.

The local update function $\text{STEP}(A,B)$ 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 $S_a \cap S_b = \emptyset$,*

$$\pi_{id}(\text{STEP}(S_a,S)) \cap \pi_{id}(\text{STEP}(S_b,S)) = \emptyset \qquad (1)$$

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

$$\text{STEP}(S_a,S) \cup \text{STEP}(S_b,S) = \text{STEP}(S_a \cup S_b, S) \qquad (2)$$

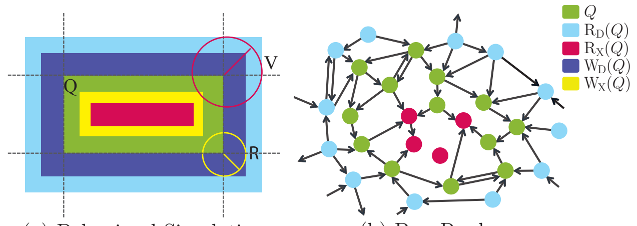
Property 1 guarantees that the outputs of computations on partitions still forms a partition of the global state. Property 2 ensures that independent computations on the subsets of the global state can be recombined simply. These two properties are the key to parallelizing the computation. In practice,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

[1]Monotonic queries maintain the containment relationship between input states [36]. So adding tuples to a state cannot make a selection query over this state return less tuples.



(a) Behavioral Simulation (b) PageRank

Figure 2: R_D, R_X, W_D and W_X Functions

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 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 1).

Figure 2(a) illustrates the implementation of the data dependency functions for the fish simulation. Formal definitions of the properties these functions must respect can be found in Appendix A. As mentioned in Section 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 [28]. Figure 2(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 B.2.

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.

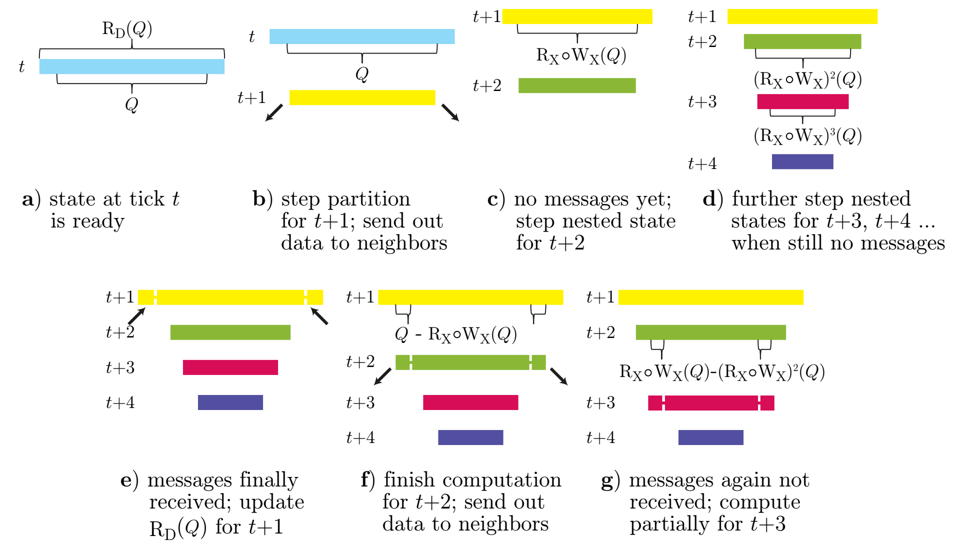
5. JITTER-TOLERANT RUNTIME

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 reordered. 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.

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



$R_D(Q)$
t
Q
$t+1$
$t+2$
$R_X \circ W_X(Q)$
$(R_X \circ W_X)^2(Q)$
$t+3$
$(R_X \circ W_X)^3(Q)$
$t+4$
a) state at tick t is ready
b) step partition for $t+1$; send out data to neighbors
c) no messages yet; step nested state for $t+2$
d) further step nested states for $t+3$, $t+4$ ... when still no messages
Q - $R_X \circ W_X(Q)$
$R_X \circ W_X(Q)$-$(R_X \circ W_X)^2(Q)$
e) messages finally received; update $R_D(Q)$ for $t+1$
f) finish computation for $t+2$; send out data to neighbors
g) messages again not received; compute partially for $t+3$

Figure 3: Dependency Scheduling.

Algorithm 1 Local Synchronization at Process P_i

```
Input: User-defined $R_D$, $W_D$, DISJOINT
Input: $Q_i$ and $S_i^0 = \mathrm{NEW}(R_D(Q_i))$
Input: Number of timesteps $T$ , Number of processors $N$
 1: for $t = 1$ to $T$ do
 2:     $S_i^t = \mathrm{STEP}(Q_i(S_i^{t-1}), S_i^{t-1})$;
 3:     for $j = 1$ to $N$ do
 4:         if $\neg\mathrm{DISJOINT}(Q_i, W_D \circ R_D(Q_j))$ then
 5:             $\mathrm{SEND}\Big((W_D \circ R_D(Q_j))(S_i^t), j\Big)$
 6:         end if
 7:     end for
 8:     for $j = 1$ to $N$ do
 9:         if $\neg\mathrm{DISJOINT}(Q_j, W_D \circ R_D(Q_i))$ then
10:             $S' = \mathrm{RECEIVE}(j)$
11:             $S_i^t = S_i^t \cup S'$
12:         end if
13:     end for
14: end for
```

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]

$$\neg\mathrm{DISJOINT}\big(Q_i, W_D \circ R_D(Q_j)\big)$$

The idea is that $R_D(Q_j)$ is complete upon having all tuples generated by correctly stepping $W_D \circ R_D(Q_j)$. 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 $W_D \circ R_D(Q_j)$. 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 R_D and 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.2 states the correctness 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.

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 3(a). We can safely advance Q to the next tick $t+1$ (Figure 3(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 3(c) and (d). Note that it is possible that for some $d' < d$, $(R_X \circ W_X)^{d'}(Q)$ is already an empty set. If this is the case, we can stop further applying $R_X \circ W_X$.

When the messages from neighbors finally arrive, we can use them to update the tuples in $R_D(Q)$ to the next tick $t+1$ (Figure 3(e)). Now, we can finish the remaining computation in Q for $t+2$ (Figure 3(f)). Intuitively, finishing computation for earlier

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

Algorithm 2 Dependency Scheduling at Process P_i

```
Input: User-defined $W_X$, $R_X$, DISJOINT
Input: $Q_i$ and $S_i^0 = \text{NEW}(R_D(Q_i))$
Input: Number of timesteps $T$, Scheduling depth $d$
 1: Initialize $t_c = t_w = 1$, DEPTH$[1] = 0$, DEPTH$[2..T] = -1$
 2: while $t_w \le T$ do
 3:   if $t_c \le T$ then
 4:     /* schedule next computation to execute */
 5:     $Q_i^1 = (R_X \circ W_X)^{\text{DEPTH}[t_c]}(Q_i)$
 6:     if DEPTH$[t_c+1] = -1$ then /* not initialized */
 7:       $S_i^{t_c} = \text{STEP}(Q_i^1(S_i^{t_c-1}), S_i^{t_c-1})$
 8:     else
 9:       $Q_i^2 = (R_X \circ W_X)^{\text{DEPTH}[t_c+1]-1}(Q_i)$
10:       $S_i^{t_c} = S_i^{t_c} \cup \text{STEP}\big((Q_i^1 \setminus Q_i^2)(S_i^{t_c-1}), S_i^{t_c-1}\big)$
11:     end if
12:     /* send data if $t_c$'s computation finished */
13:     if DEPTH$[t_c] = 0$ then
14:       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 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$[t_c] \le d$
24: end while
```

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 $Q - R_X \circ W_X(Q)$ to $t+2$, however, we may need to inspect data in $R_X \circ W_X(Q)$ at $t+1$ as context. The maintenance of these multiple versions is illustrated in Figure 3 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 $R_X \circ W_X(Q) - (R_X \circ W_X)^2(Q)$, advancing the contained tuples to tick $t+3$ (Figure 3(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 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 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 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.

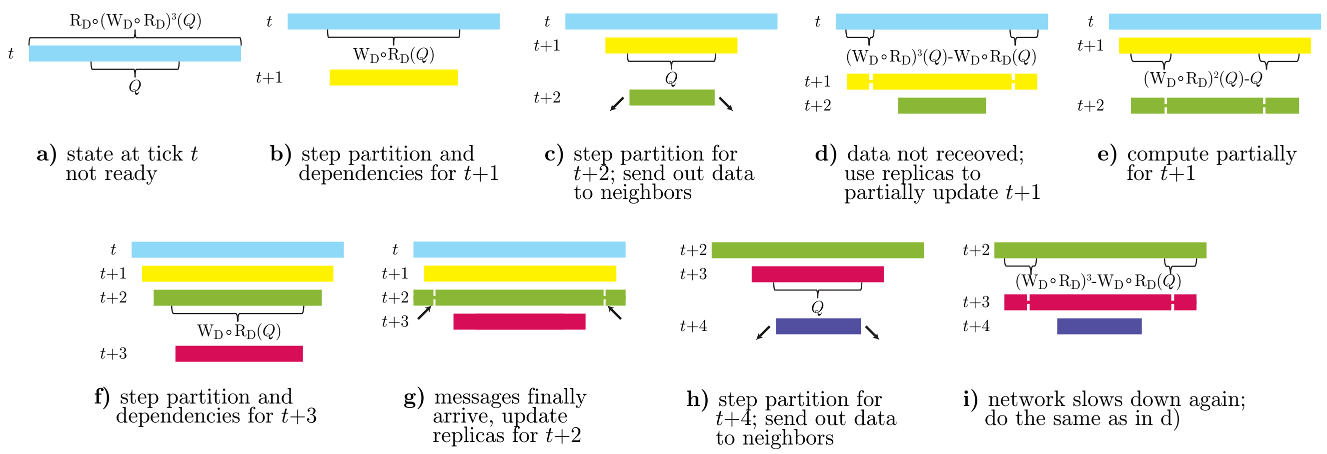
5.3 Computational Replication

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 7). However, we observe that sometimes it may be more profitable to have $k \le m$ 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 4(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 4(b) and (c)). At this point, we send messages to our neighbors. Suppose now the incoming messages from our neighbors are



a) state at tick t not ready
b) step partition and dependencies for $t+1$
c) step partition for $t+2$; send out data to neighbors
d) data not receoved; use replicas to partially update $t+1$
e) compute partially for $t+1$
f) step partition and dependencies for $t+3$
g) messages finally arrive, update replicas for $t+2$
h) step partition for $t+4$; send out data to neighbors
i) network slows down again; do the same as in d)

Figure 4: Computational Replication.

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 4(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 4(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 4(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 4(h)). The above procedure can be repeated if another latency spike again delays incoming messages (Figure 4(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, 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

Algorithm 3 Computational Replication at Process P_i

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

two techniques can work together and we show their combined effect in our experiments (Section 6).

Correctness and Efficiency. Theorem 3 in Section A 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.

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 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.

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 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 [14], a linear solver using the Jacobi method [4], and a message-passing algorithm that computes PageRank [8]. The fish simulation has already been explained throughout the paper. 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 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 B.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 [31]. In addition, we used the popular METIS graph partitioning toolkit in PART to compute a per-vertex partitioning of the input graph [28]. Pseudocode for PageRank is given in Appendix B.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 [23]. 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), 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 [13].

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 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 5 illustrates this temporal variation effect. We compare the different techniques from Section 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.

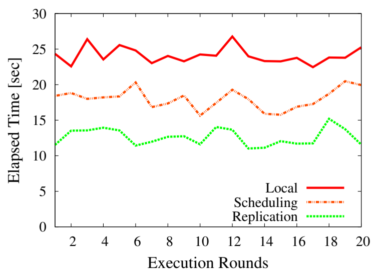


Figure 5: Variance Over Time: Jacobi

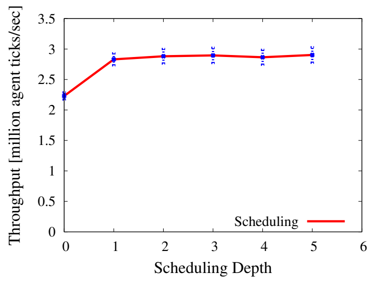


Figure 6: Scheduling: Fish Sim

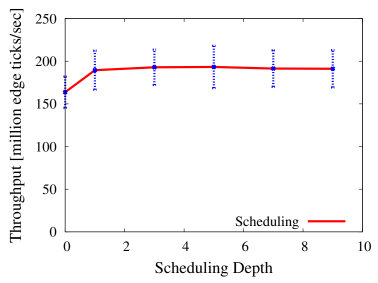


Figure 7: Scheduling: PageRank

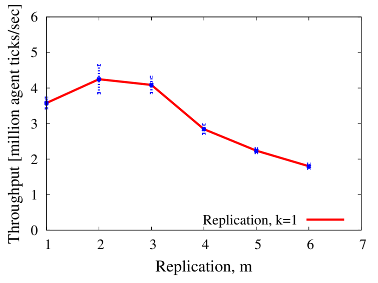


Figure 8: Replication, $k = 1$: Fish Sim

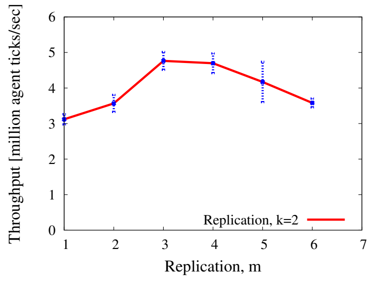


Figure 9: Replication, $k = 2$: Fish Sim

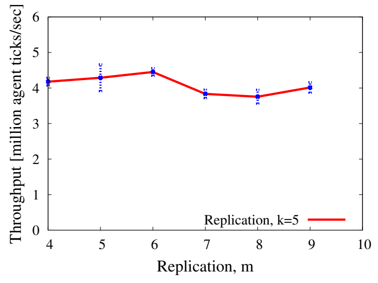


Figure 10: Replication, $k = 5$: Fish Sim

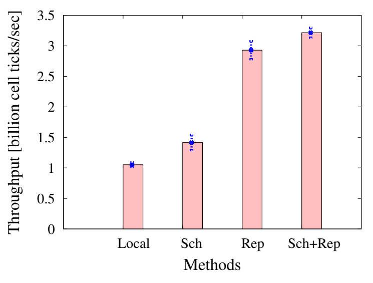


Figure 11: Effect of Combination: Jacobi

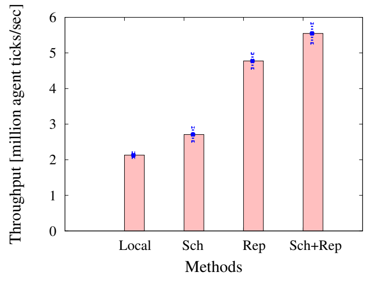


Figure 12: Effect of Combination: Fish Sim

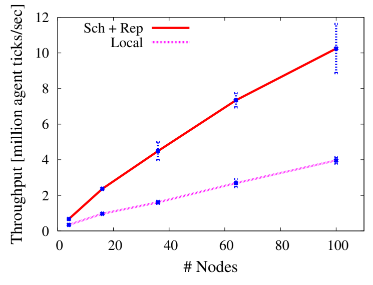


Figure 13: Scalability: Fish Sim

6.3 Results

Effect of Individual Optimizations. Figure 6 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 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 7, we observe similar behavior for PageRank. The results for the Jacobi solver are omitted due to lack of space: 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 8 to 10 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 8, 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 in tolerance to jitter.

Figures 9 and 10 exhibit similar patterns. However, in Figure 10, 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 $m = 5$.

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 5.2 and 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 11 and 12 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 experiment.

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 13 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.

7. RELATED WORK

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 [10, 20]. Optimistic protocols, on the other hand, use rollbacks to resolve conflicts [24]. 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 [21]. 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 [22, 35]. Thus, there have been efforts to move away from MPI. The DARPA High-Productivity Computing Systems (HPCS) initiative [15] 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 [11, 26, 34, 37]. 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 [41].

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 [27]. 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 [18]. 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 [38], forming subgroups of processes for global synchronization [6], and replacing the global synchronization barriers with local synchronization by dynamically exploiting locality during each time step [1, 29]. 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 [25, 30], as well as for computation of large-scale linear systems and eigenvalue problems, such as Krylov subspace methods [4, 18]. 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 [5, 7, 43, 16]. 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 [17] or DryadLINQ [50], have limited support for iterative applications; a number of recent proposals target exactly this issue [9, 33, 51]. Most of these optimizations add support to 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 paper 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 [46], 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 [2]; our work is not as ambitious as it only targets BSP scientific applications and focuses on network jitter. A confluence of BLOOM and CALM and our techniques is an interesting direction for future work.

8. 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.

9. REFERENCES

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

[2] P. Alvaro, N. Conway, J. M. Hellerstein, and W. R. Marczak. Consistency Analysis in Bloom: a CALM and Collected Approach. In *CIDR*, 2011.

[3] U. ang, C. Tsourakakis, A. P. Appel, C. Faloutsos, and J. Leskovec. HADI: Mining radii of large graphs. *TKDD*, 2010.

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

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

[6] O. Bonorden, B. Juurlink, I. von Otte, and I. Rieping. The paderborn university BSP (PUB) library. *Parallel Computing*, 29(2):187–207, 2003.

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

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

[9] Y. Bu, B. Howe, M. Balazinska, and M. Ernst. HaLoop: Efficient iterative data processing on large clusters. *PVLDB*, 3(1-2):285–296, 2010.

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

[11] R. Chen, X. Weng, B. He, and M. Yang. Large graph processing in the cloud. In *SIGMOD*, 2010.

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

[13] Cornell Database Group Website. http://www.cs.cornell.edu/bigreddata/games.

[14] 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.

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

[16] R. Das, Y. shin Hwang, M. Uysal, J. Saltz, and A. 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.

[17] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[18] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. A. Yelick. Avoiding communication in sparse matrix computations. In *IPDPS*, pages 1–12. IEEE, 2008.

[19] C. Evangelinos and C. N. Hill. Cloud computing for parallel scientific hpc applications. In *CCA*, 2008.

[20] M. J. Feeley and H. M. Levy. Distributed shared memory with versioned objects. In *OOPSLA*, 1992.

[21] W. Gropp. Learning from the success of mpi. In *HiPC*, 2001.

[22] P. B. Hansen. An evaluation of the message-passing interface. *SIGPLAN Not.*, 33:65–72, March 1998.

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

[24] D. R. Jefferson, B. Beckman, F. Wieland, L. Blume, M. D. Loreto, P. Hontalas, P. Laroche, K. Sturdevant, J. Tupman, L. V. Warren, J. J. Wedel, H. Younger, and S. Bellenot. Distributed simulation and the time warp operating system. In *SOSP*, 1987.

[25] S. D. Kamvar, T. H. Haveliwala, C. D. Manning, and G. H. Golub. Extrapolation methods for accelerating the computation of pagerank. In *WWW*, 2003.

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

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

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

[29] J.-S. Kim, S. Ha, and C. S. Jhon. Efficient barrier synchronization mechanism for the BSP model on message-passing architectures. In *IPPS/SPDP*, 1998.

[30] C. P. Lee, G. H. Golub, and S. A. Zenios. A two-stage algorithm for computing pagerank and multistage generalizations. *Internet Mathematics*, 4(4):299–327, 2007.

[31] J. Leskovec, J. M. Kleinberg, and C. Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *KDD*, pages 177–187, 2005.

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

[33] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum. Stateful bulk processing for incremental analytics. In *SoCC*, pages 51–62, 2010.

[34] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new parallel framework for machine learning. In *UAI*, 2010.

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

[36] D. Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.

[37] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD*, 2010.

[38] L. S. Nyland, J. Prins, R. H. Yun, J. Hermans, H.-C. Kum, and L. Wang. Modeling dynamic load balancing in molecular

dynamics to achieve scalable parallel execution. In *IRREGULAR*, pages 356–365, 1998.

[39] L. Ramakrishnan, K. R. Jackson, S. Canon, S. Cholia, and J. Shalf. Defining future platform requirements for e-Science clouds. In *SoCC*, pages 101–106, 2010.

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

[41] J. C. Sancho, K. J. Barker, D. J. Kerbyson, and K. Davis. Quantifying the potential benefit of overlapping communication and computation in large-scale scientific applications. In *SC*, 2006.

[42] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime measurements in the cloud: Observing, analyzing, and reducing variance. *PVLDB*, 3(1):460–471, 2010.

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

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

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

[46] G. Wang, M. A. V. Salles, B. Sowell, X. Wang, T. Cao, A. J. Demers, J. Gehrke, and W. M. White. Behavioral simulations in mapreduce. *PVLDB*, 3(1):952–963, 2010.

[47] L. Youseff, K. Seymour, H. You, J. Dongarra, and R. Wolski. The impact of paravirtualized memory hierarchy on linear algebra computational kernels and software. In *HPDC*, pages 141–152, 2008.

[48] L. Youseff, R. Wolski, B. Gorda, and C. Krintz. Evaluating the performance impact of Xen on MPI and process execution for HPC systems. In *VTDC*, 2006.

[49] L. Youseff, R. Wolski, B. Gorda, and C. Krintz. Paravirtualization for HPC systems. In *ISPA Workshops*, pages 474–486, 2006.

[50] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.

[51] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing withworking sets. In *HotCloud*, 2010.

APPENDIX

A. FORMAL MODEL OF COMPUTATION

In this section, we provide a formal presentation of the time-stepped model introduced in Section 4.

A.1 Data Dependencies

We refer the reader to Section 4 for the definitions of global state and time-stepped application logic.

Section 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 1 (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

$$\textsc{Step}\Big(Q(S), S'\Big) = \textsc{Step}\Big(Q(S), S\Big) \text{ for all } Q'(S) \subseteq S' \subseteq S \quad (3)$$

The intuition of Definition 1 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 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 this paper, 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 1. □

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 2 (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

$$Q\Big(\textsc{GStep}(S)\Big) = Q\Big(\textsc{Step}(S', S)\Big) \text{ for all } Q'(S) \subseteq S' \subseteq S \quad (4)$$

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 $\textsc{Step}(Q'(S^t), S^t)$, we can obtain the complete $Q(S^{t+1})$ which already contains all possibly written data. In fact, the functions $\mathrm{W_D}$ and $\mathrm{W_X}$ in Section 4 are declared according to this definition, such that $\forall Q, \mathrm{W_X}(Q) \prec_W Q$ and $Q \prec_W \mathrm{W_D}(Q)$.

A.2 Correctness of Algorithms

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

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 1 that

$$S^{t+1} = \bigcup_{i=1}^{n} \textsc{Step}(Q_i(S^t), \mathrm{R_D}(Q_i)(S^t)) \quad (5)$$

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 2 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 $\textsc{TryReceive}(t_w)$ at line 19 eventually succeeds for all t_w. This argument proceeds by induction; assuming that $\textsc{TryReceive}(t)$ has succeeded for all processes for $t < t_w$, then $\textsc{depth}[t_w] = 0$ 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. □

B. APPLICATION PSEUDOCODE

B.1 Jacobi Pseudocode

As mentioned previously in Section 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.

B.2 PageRank Pseudocode

As observed previously in Google's Pregel framework, many graph computations are easily expressible as time-stepped applications [37]. 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 [28], 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.