Naiad: A Timely Dataflow System

Derek G. Murray   Frank McSherry   Rebecca Isaacs
Michael Isard   Paul Barham   Martín Abadi
Microsoft Research Silicon Valley
{derekmur,mcsherry,risaacs,misard,pbar,abadi}@microsoft.com

Abstract

Naiad is a distributed system for executing data parallel, cyclic dataflow programs. It offers the high throughput of batch processors, the low latency of stream processors, and the ability to perform iterative and incremental computations. Although existing systems offer some of these features, applications that require all three have relied on multiple platforms, at the expense of efficiency, maintainability, and simplicity. Naiad resolves the complexities of combining these features in one framework.

A new computational model, *timely dataflow*, underlies Naiad and captures opportunities for parallelism across a wide class of algorithms. This model enriches dataflow computation with timestamps that represent logical points in the computation and provide the basis for an efficient, lightweight coordination mechanism.

We show that many powerful high-level programming models can be built on Naiad's low-level primitives, enabling such diverse tasks as streaming data analysis, iterative machine learning, and interactive graph mining. Naiad outperforms specialized systems in their target application domains, and its unique features enable the development of new high-performance applications.

1 Introduction

Many data processing tasks require low-latency interactive access to results, iterative sub-computations, and consistent intermediate outputs so that sub-computations can be nested and composed. Figure 1 exemplifies these
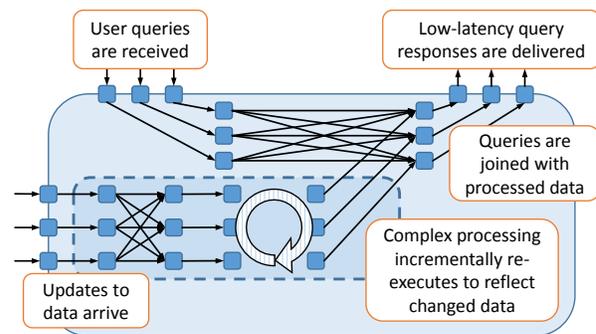
Figure 1: A Naiad application that supports real-time queries on continually updated data. The dashed rectangle represents iterative processing that incrementally updates as new data arrive.

requirements: the application performs iterative processing on a real-time data stream, and supports interactive queries on a fresh, consistent view of the results. However, no existing system satisfies all three requirements: stream processors can produce low-latency results for non-iterative algorithms [3, 5, 9, 38], batch systems can iterate synchronously at the expense of latency [27, 30, 43, 45], and trigger-based approaches support iteration with only weak consistency guarantees [29, 36, 46]. While it might be possible to assemble the application in Figure 1 by combining multiple existing systems, applications built on a single platform are typically more efficient, succinct, and maintainable.

Our goal is to develop a general-purpose system that fulfills all of these requirements and supports a wide variety of high-level programming models, while achieving the same performance as a specialized system. To this end, we have developed a new computational model, *timely dataflow*, that supports the following features:

1. structured loops allowing feedback in the dataflow,

2. stateful dataflow vertices capable of consuming and producing records without global coordination, and

3. notifications for vertices once they have received all records for a given round of input or loop iteration.
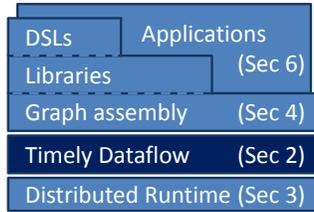
Figure 2: The Naiad software stack exposes a low-level graph assembly interface, upon which high-level libraries, DSLs, and applications can be built.

Figure 3: This simple timely dataflow graph (§2.1) shows how a loop context nests within the top-level streaming context.

Together, the first two features are needed to execute iterative and incremental computations with low latency. The third feature makes it possible to produce consistent results, at both outputs and intermediate stages of computations, in the presence of streaming or iteration.

Timely dataflow exposes a principled set of low-level primitives to the programmer, who can use those primitives to build higher-level programming abstractions. Timely dataflow graphs are directed and may include cycles. Stateful vertices asynchronously receive messages and notifications of global progress. Edges carry records with logical timestamps that enable global progress to be measured. Unlike the timestamps used in previous systems [3, 5, 9], these logical timestamps reflect structure in the graph topology such as loops, and make the model suitable for tracking progress in iterative algorithms. We show that these primitives are sufficient to express existing frameworks as composable and efficient libraries.

Naiad is our prototype implementation of timely dataflow for data parallel computation in a distributed cluster. Like others [16, 42, 43] we target problems for which the working set fits in the aggregate RAM of the cluster, in line with our goal of a low-latency system. Practical challenges arise when supporting applications that demand a mix of high-throughput and low-latency computation. These challenges include coordinating distributed processes with low overhead, and engineering the system to avoid stalls—from diverse sources such as lock contention, dropped packets, and garbage collection—that disproportionately affect computations that coordinate frequently.

We evaluate Naiad against several batch and incremental workloads, and use microbenchmarks to investigate the performance of its underlying mechanisms. Our prototype implementation outperforms general-purpose batch processors, and often outperforms state-of-the-art asynchronous systems which provide few semantic guarantees. To demonstrate the expressiveness of the model and the power of our high-level libraries, we build a complex application based on the dataflow in Figure 1 using tens of lines of code (see §6.4). The resulting application responds to queries with 4–100 ms latency.
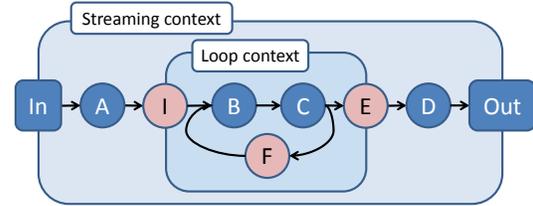
2 Timely dataflow

Timely dataflow is a computational model based on a directed graph in which stateful vertices send and receive logically timestamped messages along directed edges. The dataflow graph may contain nested cycles, and the timestamps reflect this structure in order to distinguish data that arise in different input epochs and loop iterations. The resulting model supports concurrent execution of different epochs and iterations, and explicit vertex notification after all messages with a specified timestamp have been delivered. In this section we define the structure of timely dataflow graphs, introduce the low-level vertex programming model, and explain how to efficiently reason about the delivery of vertex notifications.

2.1 Graph structure

A timely dataflow graph has input vertices and output vertices, where each input receives a sequence of messages from an external producer, and each output emits a sequence of messages back to an external consumer. The external producer labels each message with an integer *epoch*, and notifies the input vertex when it will not receive any more messages with a given epoch label. The producer may also "close" an input vertex to indicate that it will receive no more messages from any epoch. Each output message is labeled with its epoch, and the output vertex signals the external consumer when it will not output any more messages from a given epoch, and when all output is complete.

Timely dataflow graphs are directed graphs with the constraint that the vertices are organized into possibly nested *loop contexts*, with three associated system-provided vertices. Edges entering a loop context must pass through an *ingress vertex* and edges leaving a loop context must pass through an *egress vertex*. Additionally, every cycle in the graph must be contained entirely within some loop context, and include at least one *feedback vertex* that is not nested within any inner loop contexts. Figure 3 shows a single loop context with ingress ('I'), egress ('E'), and feedback ('F') vertices labeled.

This restricted looping structure allows us to design logical timestamps based on the dataflow graph structure. Every message bears a logical timestamp of type

$$\text{Timestamp} : (\overbrace{e \in \mathbb{N}}^{\text{epoch}}, \overbrace{\langle c_1, \ldots, c_k \rangle \in \mathbb{N}^k}^{\text{loop counters}})$$

where there is one loop counter for each of the k loop contexts that contain the associated edge. These loop counters explicitly distinguish different iterations, and allow a system to track forward progress as messages circulate around the dataflow graph.

The ingress, egress, and feedback vertices act only on the timestamps of messages passing through them. The vertices adjust incoming timestamps as follows:

Vertex	Input timestamp	Output timestamp
Ingress	$(e, \langle c_1, \ldots, c_k \rangle)$	$(e, \langle c_1, \ldots, c_k, 0 \rangle)$
Egress	$(e, \langle c_1, \ldots, c_k, c_{k+1} \rangle)$	$(e, \langle c_1, \ldots, c_k \rangle)$
Feedback	$(e, \langle c_1, \ldots, c_k \rangle)$	$(e, \langle c_1, \ldots, c_k + 1 \rangle)$

For two timestamps $t_1 = (x_1, \vec{c}_1)$ and $t_2 = (x_2, \vec{c}_2)$ within the same loop context, we order $t_1 \le t_2$ if and only if both $x_1 \le x_2$ and $\vec{c}_1 \le \vec{c}_2$, where the latter uses the lexicographic ordering on integer sequences. This order corresponds to the constraint on future times at which one message could result in another, a concept that we formalize in the following subsections.

2.2 Vertex computation

Timely dataflow vertices send and receive timestamped messages, and may request and receive notification that they have received all messages bearing a specific timestamp. Each vertex v implements two callbacks:

v.ONRECV(e : Edge, m : Message, t : Timestamp)
v.ONNOTIFY(t : Timestamp).

A vertex may invoke two system-provided methods in the context of these callbacks:

$this$.SENDBY(e : Edge, m : Message, t : Timestamp)
$this$.NOTIFYAT(t : Timestamp).

Each call to u.SENDBY(e, m, t) results in a corresponding invocation of v.ONRECV(e, m, t), where e is an edge from u to v, and each call to v.NOTIFYAT(t) results in a corresponding invocation of v.ONNOTIFY(t). The invocations of ONRECV and ONNOTIFY are queued, and for the most part the model is flexible about the order in which they may be delivered. However, a timely dataflow system must guarantee that v.ONNOTIFY(t) is invoked only after no further invocations of v.ONRECV(e, m, t'), for $t' \le t$, will occur. v.ONNOTIFY(t) is an indication that all v.ONRECV(e, m, t) invocations have been delivered to

```
class DistinctCount<S,T> : Vertex<T>
{
  Dictionary<T, Dictionary<S,int>> counts;
  void OnRecv(Edge e, S msg, T time)
  {
    if (!counts.ContainsKey(time)) {
      counts[time] = new Dictionary<S,int>();
      this.NotifyAt(time);
    }

    if (!counts[time].ContainsKey(msg)) {
      counts[time][msg] = 0;
      this.SendBy(output1, msg, time);
    }

    counts[time][msg]++;
  }

  void OnNotify(T time)
  {
    foreach (var pair in counts[time])
      this.SendBy(output2, pair, time);
    counts.Remove(time);
  }
}
```

Figure 4: An example vertex with one input and two outputs, producing the distinct input records on `output1`, and a count for each one on `output2`. The distinct records may be sent as soon as they are seen, but the counts must wait until all records bearing that time have been received.

the vertex, and is an opportunity for the vertex to finish any work associated with time t.

The ONRECV and ONNOTIFY methods may contain arbitrary code and modify arbitrary per-vertex state, but do have an important constraint on their execution: when invoked with a timestamp t, the methods may only call SENDBY or NOTIFYAT with times $t' \ge t$. This rule guarantees that messages are not sent "backwards in time" and is crucial to support notification as described above.

As an example, Figure 4 contains code for a vertex with one input and two outputs. The first output is the set, at each time, of distinct elements observed in the input, and the second output counts how often each distinct input is observed at that time. The ONRECV method may send elements on the first output as soon as they are first observed, allowing for low latency, but to ensure correctness the vertex must use ONNOTIFY to delay sending the counts until all inputs have been observed.

2.3 Achieving timely dataflow

In order to deliver notifications correctly, a timely dataflow system must reason about the impossibility of future messages bearing a given timestamp. In this subsection we lay a foundation for reasoning about the safe

delivery of notifications, and develop tools for a single-threaded implementation. Section 3 discusses the issues that arise in a distributed implementation.

At any point in an execution, the set of timestamps at which future messages can occur is constrained by the current set of unprocessed *events* (messages and notification requests), and by the graph structure. Messages in a timely dataflow system flow only along edges, and their timestamps are modified by ingress, egress, and feedback vertices. Since events cannot send messages backwards in time, we can use this structure to compute lower bounds on the timestamps of messages an event can cause. By applying this computation to the set of unprocessed events, we can identify the vertex notifications that may be correctly delivered.

Each event has a timestamp and a location (either a vertex or edge), and we refer to these as a *pointstamp*:

$$\text{Pointstamp} : (t \in \text{Timestamp}, \overbrace{l \in \text{Edge} \cup \text{Vertex}}^{\text{location}}).$$

The SENDBY and NOTIFYAT methods generate new events: for $v.\text{SENDBY}(e,m,t)$ the pointstamp of m is (t,e) and for $v.\text{NOTIFYAT}(t)$ the pointstamp of the notification is (t,v).

The structural constraints on timely dataflow graphs induce an order on pointstamps. We say a pointstamp (t_1,l_1) *could-result-in* (t_2,l_2) if and only if there exists a path $\psi = \langle l_1,\ldots,l_2 \rangle$ in the dataflow graph such that the timestamp $\psi(t_1)$ that results from adjusting t_1 according to each ingress, egress, or feedback vertex occurring on that path satisfies $\psi(t_1) \leq t_2$. Each path can be summarized by the loop coordinates that its vertices remove, add, and increment; the resulting *path summary* between l_1 and l_2 is a function that transforms a timestamp at l_1 to a timestamp at l_2. The structure of timely dataflow graphs ensures that, for any locations l_1 and l_2 connected by two paths with different summaries, one of the path summaries always yields adjusted timestamps earlier than the other. For each pair l_1 and l_2, we find the minimal path summary over all paths from l_1 to l_2 using a straightforward graph propagation algorithm, and record it as $\Psi[l_1,l_2]$. To efficiently evaluate the could-result-in relation for two pointstamps (t_1,l_1) and (t_2,l_2), we test whether $\Psi[l_1,l_2](t_1) \leq t_2$.

We now consider how a single-threaded *scheduler* delivers events in a timely dataflow implementation. The scheduler maintains a set of *active pointstamps*, which are those that correspond to at least one unprocessed event. For each active pointstamp the scheduler maintains two counts: an *occurrence count* of how many outstanding events bear the pointstamp, and a *precursor count* of how many active pointstamps precede it in the could-result-in order. As vertices generate and retire events, the occurrence counts are updated as follows:

Operation	Update
$v.\text{SENDBY}(e,m,t)$	$\text{OC}[(t,e)] \leftarrow \text{OC}[(t,e)] + 1$
$v.\text{ONRECV}(e,m,t)$	$\text{OC}[(t,e)] \leftarrow \text{OC}[(t,e)] - 1$
$v.\text{NOTIFYAT}(t)$	$\text{OC}[(t,v)] \leftarrow \text{OC}[(t,v)] + 1$
$v.\text{ONNOTIFY}(t)$	$\text{OC}[(t,v)] \leftarrow \text{OC}[(t,v)] - 1$

The scheduler applies updates at the start of calls to SENDBY and NOTIFYAT, and as calls to ONRECV and ONNOTIFY complete. When a pointstamp p becomes active, the scheduler initializes its precursor count to the number of existing active pointstamps that could-result-in p. At the same time, the scheduler increments the precursor count of any pointstamp that p could-result-in. A pointstamp p leaves the active set when its occurrence count drops to zero, at which point the scheduler decrements the precursor count for any pointstamp that p could-result-in. When an active pointstamp p's precursor count is zero, there is no other pointstamp in the active set that could-result-in p, and we say that p is in the *frontier* of active pointstamps. The scheduler may deliver any notification in the frontier.

When a computation begins the system initializes an active pointstamp at the location of each input vertex, timestamped with the first epoch, with an occurrence count of one and a precursor count of zero. When an epoch e is marked complete the input vertex adds a new active pointstamp for epoch $e + 1$, then removes the pointstamp for e, permitting downstream notifications to be delivered for epoch e. When the input vertex is closed it removes any active pointstamps at its location, allowing all events downstream of the input to eventually drain from the computation.

2.4 Discussion

Although the timestamps in timely dataflow are more complicated than traditional integer-valued timestamps [22, 38], the vertex programming model supports many advanced use cases that motivate other systems.

The requirement that a vertex explicitly request notifications (rather than passively receive notifications for all times) allows a programmer to make performance trade-offs by choosing when to use coordination. For example, the monotonic aggregation operators in Bloom^L [13] may continually revise their output without coordination; in Naiad a vertex can achieve this by sending outputs from ONRECV. Such an implementation can improve performance inside a loop by allowing fast uncoordinated iteration, at the possible expense of sending multiple messages before the output reaches its final value. On the other hand an implementation that sends only once, in ONNOTIFY, may be more useful at the boundary of a sub-computation that will be composed with other processing, since the guarantee that only a single value will be produced simplifies the downstream
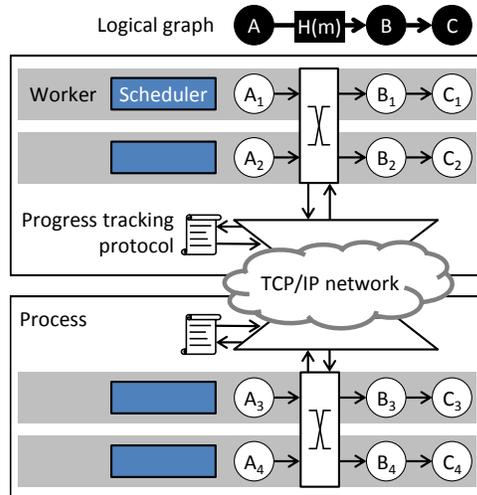
Figure 5: The mapping of a logical dataflow graph onto the distributed Naiad system architecture.

sub-computation. Timely dataflow makes it easy to combine both styles of implementation in a single program.

As described, a notification in timely dataflow is guaranteed not to be delivered before a time t, and has the capability to send messages at times greater or equal to t. We can decouple these two properties of a notification into a *guarantee time* t_g and *capability time* t_c, which may be distinct. This generalization for example allows "state purging" notifications [22] that free resources associated with t_g, but do not generate other events and so can set t_c to $\top$ (i.e., after all processing). Since $t_c = \top$, the notification does not prevent other notifications from being delivered, and need not introduce any coordination. Notifications with $t_g < t_c$ can also be useful to constrain otherwise asynchronous execution, for example by providing "bounded staleness" [11], which guarantees that the system does not proceed more than a defined number of iterations beyond any incomplete iteration.

3 Distributed implementation

Naiad is our high-performance distributed implementation of timely dataflow. Figure 5 shows the schematic architecture of a Naiad cluster: a group of processes hosting *workers* that manage a partition of the timely dataflow vertices. Workers exchange messages locally using shared memory, and remotely using TCP connections between each pair of processes. Each process participates in a distributed progress tracking protocol, in order to coordinate the delivery of notifications. We implemented the core Naiad runtime as a C# library, in 22,700 lines of code. In this section, we describe the techniques that Naiad uses to achieve high performance.

3.1 Data parallelism

Like other dataflow systems [15, 41, 42] Naiad relies on data parallelism to increase the aggregate computation, memory, and bandwidth available to applications. A program specifies its timely dataflow graph as a *logical graph* of *stages* linked by typed *connectors*. Each connector optionally has a partitioning function to control the exchange of data between stages. At execution time, Naiad expands the logical graph into a *physical graph* where each stage is replaced by a set of vertices and each connector by a set of edges. Figure 5 shows a logical graph and a corresponding physical graph, where the connector from A to B has partitioning function H(m) on typed messages m.

The regular structure of data parallel dataflow graphs simplifies vertex implementations, which can be agnostic to the degree of parallelism in a stage. When a vertex sends a message on a connector, the system automatically routes the message to the appropriate destination vertex using the partitioning function. Specifically, the partitioning function maps a message to an integer, and the system routes all messages that map to the same integer to the same downstream vertex. A programmer can use partitioning functions to hash or range partition incoming messages by a key, in order to implement "group by" or "reduce" functionality [15, 41]. When no partitioning function is supplied, the system delivers messages to a local vertex (e.g., B_i to C_i in Figure 5).

Regular structure also allows Naiad to simplify its reasoning about the could-result-in relation. Naiad projects each pointstamp p from the physical graph to a pointstamp $\hat{p}$ in the logical graph, and evaluates the could-result-in relation on the projected pointstamps. This projection leads to a loss of resolution, since there are cases where p_1 cannot-result-in p_2 but $\hat{p}_1$ could-result-in $\hat{p}_2$. However, using the logical graph ensures that the size of the data structures used to compute the relation depends only on the logical graph and not the much larger physical graph. As we explain in §3.3, using projected pointstamps also reduces the amount of communication needed for coordination between workers.

3.2 Workers

Each Naiad worker is responsible for delivering messages and notifications to vertices in its partition of the timely dataflow graph. When faced with multiple runnable actions (messages and notifications to deliver) workers break ties by delivering messages before notifications, in order to reduce the amount of queued data. Different policies could be used, such as prioritizing the delivery of messages and notifications with the earliest pointstamp to reduce end-to-end latency.

Workers communicate using shared queues and have no other shared state. This isolation ensures that only a single thread of control ever executes within a vertex, and allows much simpler vertex implementations. Each call to SENDBY implicitly causes the calling vertex to yield if the destination vertex is managed by the same worker. Thus, the worker may deliver the message immediately by invoking the appropriate ONRECV callback, rather than queuing it for later delivery. At this point the worker may also deliver queued messages that have been received from other workers. The ability of a worker to move between vertices and deliver incoming remote messages enables Naiad to keep system queues small and to lower the latency of message delivery.

The existence of cycles in the dataflow graph raises the possibility of re-entrancy: a vertex interrupted when it calls SENDBY may be re-entered by one of its ONRECV callbacks. By default vertices are not re-entrant, and the vertex's worker must enqueue the message for later delivery; however, a vertex implementation may optionally specify a bounded depth for re-entrant calls. Without support for re-entrancy, the implementations of many iterative patterns would overload the system queues. Re-entrancy allows the vertex implementation to coalesce incoming messages in ONRECV, and thereby reduce overall memory consumption.

3.3 Distributed progress tracking

Before delivering a notification, a Naiad worker must know that there are no outstanding events at any worker in the system with a pointstamp that could-result-in the pointstamp of the notification. We adapt the approach for progress tracking based on a single global frontier (§2.3) to a distributed setting in which multiple workers coordinate independent sets of events using a local view of the global state. We base our initial protocol on broadcasting occurrence count updates, and then refine it with two optimizations.

For each active pointstamp each worker maintains a *local occurrence count*, representing its local view of the global occurrence counts, a *local precursor count* computed from its local occurrence counts, and a *local frontier* defined using the could-result-in relation on the local active pointstamps. The worker does not immediately update its local occurrence counts as it dispatches events, but instead broadcasts (to all workers, including itself) *progress updates*, which are pairs ($p \in$ Pointstamp, $\delta \in \mathbb{Z}$) with δ chosen according to the update rules in §2.3. The broadcasts from a given worker to another must be delivered in FIFO order, but there is no constraint on ordering between two workers' broadcasts. When a worker receives a progress update (p, δ), it adds δ to the local occurrence count for p.

This protocol has an important safety property: no local frontier ever moves ahead of the global frontier, taken across all outstanding events in the system. Therefore, if some worker has a pending notification at $p = (t, v)$ and p is in the local frontier, p must also be in the global frontier and the worker can safely deliver the notification to v. A formal specification of the protocol and a safety proof are presented in a separate paper [4].

Optimizing broadcast updates A naive implementation of the protocol would broadcast every progress update and result in impractical communication demands. We implement two optimizations that, taken together, reduce the volume of communication.

The first optimization uses projected pointstamps in the progress tracking protocol. The protocol thus keeps track of occurrence and precursor counts for each stage and connector, as opposed to each vertex and edge. Although, as noted in §3.1, this representation can reduce opportunities for concurrency, it substantially reduces the volume of updates and the size of the state maintained for progress tracking.

The second optimization accumulates updates in a local buffer before broadcasting them. Updates with the same pointstamp are combined into a single entry in the buffer by summing their deltas. Updates may be accumulated as long as every buffered pointstamp p satisfies one of two properties: either some other element of the local frontier could-result-in p, or p corresponds to a vertex whose net update (the sum of the local occurrence count, the buffered update count, and any updates that the worker has broadcast but not yet received) is strictly positive. When the accumulator receives new progress updates (either from local workers or other accumulators), it must test whether the accumulated pointstamps still satisfy this condition: if not, the accumulator broadcasts all updates in the buffer. When updates are broadcast, positive values must be sent before negative values.

Any fixed group of workers can perform this accumulation, and it may be performed hierarchically. By default Naiad accumulates updates at the process level and at the cluster level: each process sends accumulated updates to a central accumulator, which broadcasts their net effect to all workers. Although this accumulation introduces an additional message delay over direct broadcasts, it substantially reduces the total number of update messages, as we evaluate in §5.3.

Our implementation includes two further optimizations to decrease the expected latency of broadcasting updates. The central cluster-level accumulator optimistically broadcasts a UDP packet containing each update before re-sending updates on the TCP connections between the accumulator and other processes. Messages contain a sequence number to ensure that delivery is or-

dered and idempotent. In addition, Naiad uses a modification of the eventcount synchronization primitive [37] that allows threads to be woken by either a broadcast or unicast notification. Without this optimization, waking workers sequentially adds significant overhead to the critical path of low-latency iterative computations.

3.4 Fault tolerance and availability

Naiad has a simple but extensible implementation of fault tolerance: each stateful vertex implements a CHECKPOINT and RESTORE interface, and the system invokes these as appropriate to produce a consistent checkpoint across all workers. Each vertex may either log data as computation proceeds, and thus respond to checkpoint requests with low latency; or write a full, and potentially more compact, checkpoint when requested. The stateful components of the progress tracking protocol implement the same interface; because they are small and frequently updated, they produce full checkpoints.

When the system periodically checkpoints, all processes first pause worker and message delivery threads, flush message queues by delivering outstanding ON-RECV events (buffering and logging any messages that are sent by doing so), and finally invoke CHECKPOINT on each stateful vertex. The system then resumes worker and message delivery threads and flushes buffered messages. Once the desired level of durability is achieved—e.g., the checkpoint files are flushed to disk, or replicated to other computers—the checkpoint is complete.

To recover from a failed process, all live processes revert to the last durable checkpoint, and the vertices from the failed process are reassigned to the remaining processes. The RESTORE method reconstructs the state of each stateful vertex using its respective checkpoint file.

There is an inherent design tension between allowing a system to make fine-grained updates to mutable state and reliably logging enough information to permit consistent recovery when a local scheduler fails. Our current design favors performance in the common case that there are no failures, at the expense of availability in the event of a failure. Naiad can consume inputs from a reliable message queue and write its outputs to a distributed key-value store [8]. This approach allows the overall system to satisfy reads and writes while Naiad recovers from failures, at some cost to freshness. Other tradeoffs may be more suitable for some applications: for example, MillWheel [5], a non-iterative streaming system with a programming model similar to the one defined in §2.2, writes per-key checkpoints for each batch of messages processed. This policy increases the latency of each message, but enables faster resumption after a failure. We discuss the throughput and latency impact of logging and checkpointing in §6.3.

3.5 Preventing micro-stragglers

Many Naiad computations are sensitive to latency: transient stalls at a single worker can have a disproportionate effect on overall performance. For example, in iterative computations a phase of execution between notifications can last as little as one millisecond [28], whereas events such as packet loss, contention on concurrent data structures, and garbage collection can result in delays ranging from tens of milliseconds to tens of seconds. The probability of such an event occurring in a single phase of execution increases with the size of a cluster, and therefore we view the resulting *micro-stragglers* as the main obstacle to scalability for low-latency workloads.

Micro-stragglers bear some similarity to the well-known stragglers in coarse-grained batch-processing systems, but different mitigation techniques apply. Workers in a batch-processing system are stateless, so scheduling duplicate work items can reduce the impact of stragglers [14, 15, 44]. Naiad maintains mutable state to decrease the latency of execution: speculatively executing duplicate work would require the system to coordinate updates to replicated state, and we expect the cost to outweigh the benefits.

Rather than dealing with micro-stragglers reactively, Naiad reduces their impact and avoids them wherever possible. We now describe several sources of micro-stragglers and their effective mitigations.

Networking Naiad uses TCP over Ethernet to deliver remote messages because it offers reliable message delivery, and modern Ethernet network interface cards (NICs) accelerate much of the TCP protocol stack in hardware. However, the throughput of messages between a pair of processes is bursty: many iterative computations begin with a large data exchange, but towards the tail it is common for messages to fit in a single packet. This bursty pattern can lead to micro-stragglers on a best-effort network such as ours, and we have taken several steps to reduce their impact.

The default TCP configuration on Windows penalizes two processes that exchange a small message in each direction with a 200 ms delay. Nagle's algorithm [32] and delayed acknowledgments [12] are responsible for this delay. We therefore disable Nagle's algorithm for Naiad TCP sockets, and reduce the delayed acknowledgment timeout to 10 ms. In the event of packet loss, the default retransmission timeout is 300 ms, which is far longer than many congestion events in Naiad: for example, the cluster-level progress tracking accumulator (§3.3) often aggregates one packet from each process, and near-simultaneous arrival of these packets can cause loss due to incast [6]. We therefore reduce the minimum retransmit timeout to 20 ms. Since Naiad aggre-

gates messages at the application level, it can maintain high throughput despite these options.

Our evaluation cluster has a switched Gigabit Ethernet network with a simple topology: one core switch, and two top-of-rack switches with 32 ports each. Despite over-provisioning the inter-switch links with a 40 Gbps uplink and enabling 802.3x flow control, we observe packet loss at the NIC receive queues during incast traffic patterns [31]. It is likely that Datacenter TCP [6] would be beneficial for our workload, but the rack switches in our cluster lack necessary support for explicit congestion notification.

Since Naiad controls all aspects of data exchange, it is likely that a specialized transport protocol would provide better performance than TCP over Ethernet. We are investigating the use of RDMA over InfiniBand, which has the potential to reduce micro-stragglers using mechanisms such as microsecond message latency, reliable multicast, and user-space access to message buffers. These mechanisms will avoid TCP-related timers in the operating system, but achieving optimal performance will require attention to quality of service [35].

Data structure contention To scale out within a single machine, most data structures in Naiad—in particular the vertex state—are accessed from a single worker thread. Nevertheless, coordination is required to exchange messages between workers, and Naiad uses .NET concurrent queues and lightweight spinlocks for this purpose. These primitives back off by sleeping for 1 ms when contention is detected. Since the default timer granularity on Windows is 15.6 ms, with typical scheduling quanta of 100 ms or more, backing off can result in very high latency for concurrent access to a contended shared data structure. Decreasing the clock granularity to 1 ms reduces the impact of these stalls.

Garbage collection The .NET runtime, on which we implemented Naiad, uses a mark-and-sweep garbage collector (GC) to reclaim memory. While the .NET GC is concurrent, it can suspend thread execution during some allocations and lead to micro-stragglers.

To lower the cost of garbage collection, we engineered the system to trigger the GC less frequently, and shorten pauses due to collection. The Naiad runtime and the libraries that we have built on top of it avoid object allocation wherever possible, using buffer pools to recycle message buffers and transient operator state (such as queues). We use value types extensively, because an array of value-typed objects can be allocated as a single region of memory with a single pointer, and the GC cost is proportional to the number of pointers (rather than objects). The .NET runtime supports structured value types, enabling their use for many Naiad data structures.

4 Writing programs with Naiad

Although one can write Naiad programs directly against its timely dataflow abstraction, many users find simpler, higher-level interfaces easier to use. Examples include SQL, MapReduce [15], LINQ [41], Pregel's vertex-program abstraction [27], and PowerGraph's GAS abstraction [16]. We designed Naiad so that common timely dataflow patterns can be collected into libraries, allowing users to draw from these libraries when they meet their needs and to construct new timely dataflow vertices when they do not, all within the same program. This section first shows a simple Naiad program to highlight the common structure of applications built on Naiad, then discusses some of the libraries we have built, and finally sketches the process of writing libraries and custom vertices using the low-level Naiad API.

4.1 A prototypical Naiad program

All Naiad programs follow a common pattern: first define a dataflow graph, consisting of input stages, computational stages, and output stages; and then repeatedly supply the input stages with data. Input and output stages follow a push-based model, in which the user supplies new data for each input epoch, and Naiad invokes a user-supplied callback for each epoch of output data. The following example fragment uses our library for incremental computation [28], which allows the programmer to use patterns familiar from LINQ to implement an incrementally updatable MapReduce computation:

```
// 1a. Define input stages for the dataflow.
var input = controller.NewInput<string>();

// 1b. Define the timely dataflow graph.
// Here, we use LINQ to implement MapReduce.
var result = input.SelectMany(y => map(y))
                  .GroupBy(y => key(y),
                      (k, vs) => reduce(k, vs));

// 1c. Define output callbacks for each epoch
result.Subscribe(result => { ... });

// 2. Supply input data to the query.
input.OnNext(/* 1st epoch data */);
input.OnNext(/* 2nd epoch data */);
input.OnNext(/* 3rd epoch data */);
input.OnCompleted();
```

Step 1a defines the source of data, and Step 1c defines what to do with output data when produced. Step 1b constructs a timely dataflow graph using `SelectMany` and `GroupBy` library calls, which assemble stages of pre-defined vertices and behave as their LINQ counterparts: `SelectMany` applies its argument function to each message, and `GroupBy` collates the results by a key function before applying its reduction function.

Once the graph is fully assembled, in step 2 `OnNext` supplies the computation with epochs of input data. The `Subscribe` stage applies its callback to each completed epoch of data it observes. Finally, `OnCompleted` indicates that no further epochs of input data exist, allowing Naiad to drain messages and cleanly shut down the computation.

4.2 Data parallel patterns in Naiad

We packaged several higher-level programming patterns into libraries implemented on Naiad's timely dataflow abstractions. This separation of library code from system code makes it easy for users to draw from existing patterns, create their own patterns, and adapt other patterns, all without requiring access to private APIs. Public reusable low-level programming abstractions distinguish Naiad from a number of other data parallel systems [26, 27, 41, 42] that enforce a single high-level programming model, and conceal the boundary between this model and lower-level primitives within private system code. We hope that this distinction will make Naiad appealing as the implementation layer for future data parallel projects.

It was straightforward to implement a library of incremental LINQ-like operators. Most build on unary and binary forms of a generic buffering operator whose ONRECV function adds records to lists indexed by timestamp, and whose ONNOTIFY(t) applies a suitable transformation to the list(s) for timestamp t. To improve performance, we specialized the implementation of operators that do not require coordination: for example `Concat` immediately forwards records from both inputs, `Select` transforms and outputs data without buffering, and `Distinct` outputs a record as soon as it is seen for the first time. The ability to perform this specialization in library code (rather than in the core Naiad runtime) decouples the evolution of the LINQ implementation from improvements to the underlying system.

We implemented a subset of the Bloom framework for asynchronous computation [7]. The LINQ operators `Where`, `Concat`, `Distinct`, and `Join` are sufficient, within a loop, to implement Datalog-style queries. None of these operators invokes NOTIFYAT, and subgraphs using only these will execute asynchronously (without coordination) on Naiad. We also implemented a monotonic `Aggregate` operator that emits records when the aggregate improves, and is suitable for implementing Bloom^L-style aggregation [13]. All of these constructs compose with other LINQ operators and timely dataflow stages, and Naiad introduces coordination only where vertices explicitly require it.

As a final example, we implemented a version of the Pregel bulk synchronous parallel model for graph algorithms [27] as a Naiad library. A Pregel program operates on a data graph in a series of iterations (or "supersteps") in which messages are exchanged, aggregates computed, and the graph mutated. While one can build a Pregel-like implementation using LINQ-style operators [40], such a collection-oriented pattern makes it hard to support Pregel's full semantics including aggregation and graph mutation. Instead we base our Pregel port on a custom vertex with several strongly typed inputs and outputs (for messages, aggregated values, and graph mutations), connected via multiple feedback edges in parallel.

4.3 Constructing timely dataflow graphs

While we expect most uses of Naiad to rely on libraries of graph construction patterns, Naiad provides a simple graph construction interface based on timely dataflow. The interface is the basis for all libraries, but it also makes it easy for applications to include ad hoc vertices that provide specialized functionality.

Graph construction involves two main steps: defining the behavior of dataflow vertices, and defining the dataflow topology (including any loops). A Naiad stage is a collection of vertices defined by a vertex factory, which is invoked by the system to instantiate each independent instance of the vertex. Stages may have multiple inputs and outputs, each of which has an associated C# record type, and which are connected using typed streams whose endpoints must have matching record types. Stage inputs may specify a partitioning requirement, and stage outputs a partitioning guarantee, and the system inserts exchange connectors where necessary to ensure that input partitioning requirements are satisfied. Vertices must provide a typed ONRECV callback for each input, and must provide an ONNOTIFY callback if the stage supports notification.

In general, the inputs of a stage must be connected before its outputs, in order to prevent invalid cycles. System-provided `LoopContext` objects allow the programmer to define multiple ingress, egress, and feedback stages, and connect them to other computation stages. Only feedback stages may have their outputs connected before their inputs, and this ensures that all cycles conform to the constraints of valid timely dataflow graphs.

5 Performance evaluation

Naiad is designed to perform effectively in different modes of operation, supporting both high throughput and low latency as required by the workload. In this section we examine Naiad's behavior in each of these operating regimes using several micro-benchmarks.
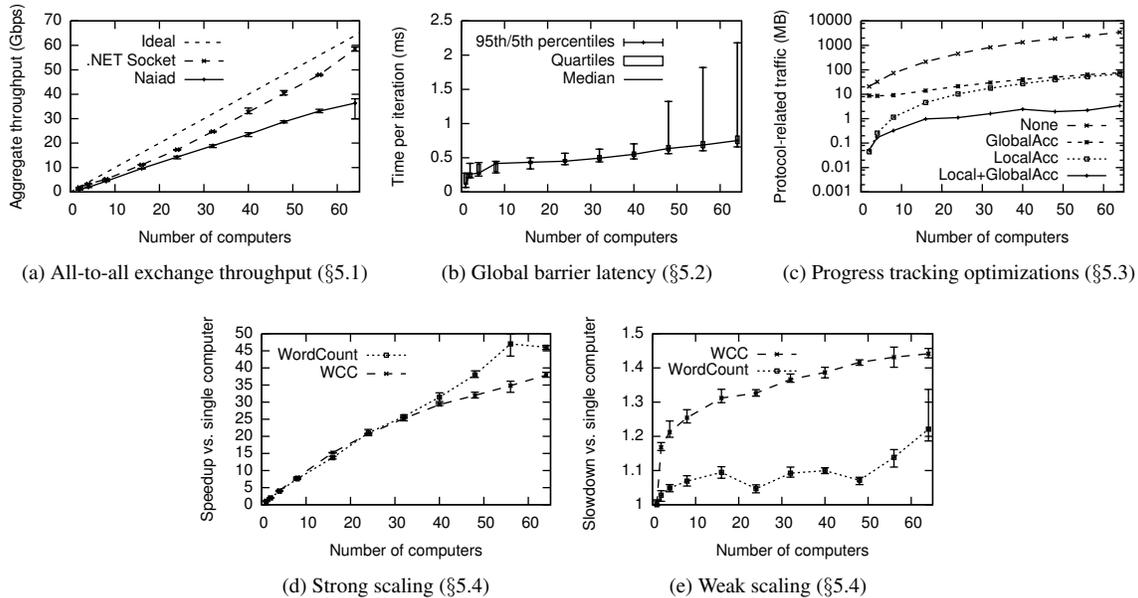
Figure 6: These microbenchmarks evaluate baseline Naiad system performance on synthetic datasets.

The hardware configuration is as follows: two racks of 32 computers, each with two quad-core 2.1 GHz AMD Opteron processors, 16 GB of memory, and an Nvidia NForce Gigabit Ethernet NIC. Each rack switch has a 40 Gbps uplink to the core switch. Unless otherwise stated, graphed points are the average across five trials, with error bars showing minimum and maximum values.

5.1 Throughput

The first micro-benchmark measures the maximum throughput for a distributed computation. The program constructs a cyclic dataflow that repeatedly performs the all-to-all data exchange of a fixed number of records. Figure 6a plots the aggregate throughput against the number of computers.

The uppermost line shows the "Ideal" aggregate throughput based on the Ethernet bandwidth, and the middle line shows the sustained all-to-all throughput achieved at the .NET Socket layer using long-running TCP connections with 64 KB messages. The middle line demonstrates the achievable throughput given the network topology, TCP overheads, and .NET API costs.

The final line shows the throughput that Naiad achieves when an application exchanges a large number of 8-byte records (50M per computer) between all processes in the cluster. The small record size leads to near worst-case overheads for serialization and evaluating the partitioning function. The experiment demonstrates that Naiad's throughput capabilities scale linearly, though opportunities exist to improve its absolute performance.

5.2 Latency

The second experiment evaluates the minimal time required for global coordination. Again, we construct a simple cyclic dataflow graph, but in this case the vertices exchange no data and simply request and receive completeness notifications. No iteration can proceed until all notifications from the previous iteration have been delivered. Figure 6b plots the distribution of times for 100K iterations using median, quartiles, and 95th percentile values. The median time per iteration remains small at $753\,\mu$s for 64 computers (512 workers), but the 95th percentile results show the adverse impact of micro-stragglers as the number of computers increases.

For many real programs, the subset of vertices in a stage requesting a completeness notification can be relatively small, and having fewer participants reduces the cost of coordination. For example, in the tail of a fixed-point computation, where latency is crucial, communication patterns typically become sparse and hence the number of participants in coordination is often small.

5.3 Protocol optimizations

In order to evaluate the optimizations to the progress tracking protocol described in §3.3, we run a weakly connected components (WCC) computation on a random graph of 300M edges (about 2.2 GB of raw input). Figure 6c shows the number of bytes of progress protocol traffic generated by 8 workers per computer. In this experiment we plot the results from one run, since the

volume of progress protocol traffic for this computation does not vary significantly between executions.

The optimizations reduce the volume of protocol traffic by one or two orders of magnitude, depending on whether the accumulation is performed at the computer level ("LocalAcc"), at the cluster level ("GlobalAcc"), or both. In practice we find little difference in running times with and without global accumulation; the reduction in messages from local accumulation at the computer level is sufficient to prevent progress traffic from becoming a bottleneck. Although we have not observed it experimentally, we are aware that the current protocol may limit scalability, and anticipate that a deeper accumulation and distribution tree will help to disseminate progress updates more efficiently within larger clusters.

5.4 Scaling

We consider the scaling characteristics of Naiad using two contrasting applications. WordCount is an embarrassingly parallel MapReduce program that computes word frequencies in a Twitter corpus of size 128 GB uncompressed, generated by replicating an initial 12.0 GB corpus. WCC is a weakly connected components computation on a random graph of 200M edges. WCC is a challenging scalability test: it involves numerous synchronization points and is throughput-limited in early iterations of the loop, becoming latency-constrained when nearing convergence.

In order to evaluate strong scaling, we add compute resources while keeping the size of the input fixed, so we expect communication cost to eventually limit further scaling. Figure 6d plots the running times for the two applications. WCC starts to scale more slowly at around 24 computers and reaches a maximum speedup of $38\times$ on 64 computers. WordCount scales fairly linearly, with $46\times$ speedup on 64 computers.

To evaluate weak scaling, we measure the effect of increasing both the number of computers and the size of the input. A computation with perfect weak scaling would have equal running time for each configuration.

Figure 6e shows how WCC performs on a random input graph with a constant number of edges (18.2M) and nodes (9.1M) per computer. The running time degrades by a factor of approximately $1.44\times$ (29.4 s compared to 20.4 s) for a 1.1B edge graph run on 64 computers, relative to the execution in a single computer. Most of the deviation from perfect scaling can be explained by the throughput experiment. For every weak scaling configuration of WCC, the amount of data sent and received by the workers on a given computer is a constant 360 MB. When run on a single computer the destination is always local. However, two computers exchange half of the data across the network, and 64 computers exchange $\frac{63}{64}$ of

the data (355 MB) across the network. From Figure 6a, the cost of exchanging 355 MB between 64 computers is $\sim$7.6 s, accounting for most of the 9 s slowdown.

Figure 6e also shows the weak scaling of WordCount, with 2 GB compressed input per computer. The amount of data exchanged in WordCount is far smaller than in WCC, because of the greater effectiveness of combiners before the data exchange, but it still grows with the number of processes and the computation becomes throughput-limited during the data exchange. As a result, WordCount does not achieve perfect weak scaling (in the worst case $1.23\times$ the single-computer time), but its weak scaling improves over WCC.

6   Real world applications

We now consider several applications drawn from the literature on batch, streaming, and graph computation, and compare Naiad's expressiveness and performance against existing systems. The additional properties and features of other systems complicate a comparison, but we show that Naiad achieves excellent performance relative to both general-purpose frameworks and specialized systems, and that it can express algorithms at a high level with a modest number of lines of code.

Additionally, we develop and evaluate an example in the spirit of Figure 1, maintaining statistics derived from an incremental graph analysis, and serving interactive queries against the results. We are not aware of another system that can implement this computation at interactive timescales, whereas Naiad responds to updates and queries with sub-second latencies.

Unless otherwise specified, the cluster setup and the meaning of error bars are as described in Section 5.

6.1   Batch iterative graph computation

Najork *et al.* [34] compare three different approaches to graph computation on large-scale real world datasets: using a distributed database (PDW [2]), a general-purpose batch processor (DryadLINQ [41]), and a purpose-built distributed graph store (SHS [33]). They measure the performance of standard graph analyses over two ClueWeb09 datasets, including the larger 1B page, 8B edge "Category A" dataset [1]. We compare their performance numbers with those of Naiad in Table 1 on the same problems on equivalent hardware (16 of our cluster computers). The substantial speedups (up to $600\times$) demonstrate the power of being able to maintain application-specific state in memory between iterations. Systems like DryadLINQ incur a large per-iteration cost when serializing local state, and thus favor algorithms that minimize the number of iterations.
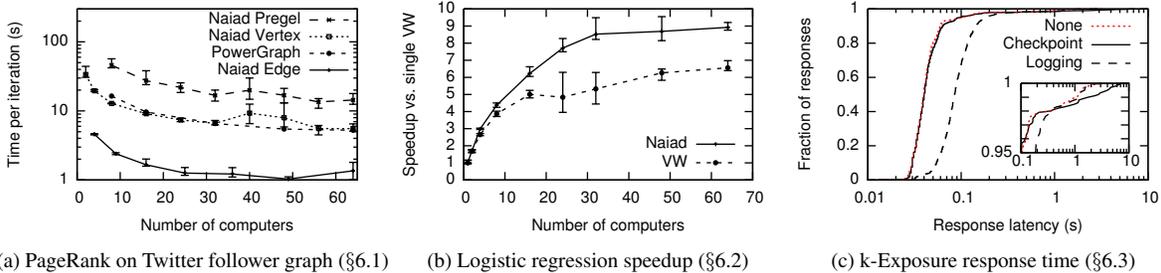
(a) PageRank on Twitter follower graph (§6.1)   (b) Logistic regression speedup (§6.2)   (c) k-Exposure response time (§6.3)

Figure 7: These experiments evaluate Naiad performance on diverse real world applications.

Algorithm	PDW	DryadLINQ	SHS	Naiad
PageRank	156,982	68,791	836,455	**4,656**
SCC	7,306	6,294	15,903	**729**
WCC	214,479	160,168	26,210	**268**
ASP	671,142	749,016	2,381,278	**1,131**

Table 1: Running times in seconds of several graph algorithms on the Category A web graph. Non-Naiad measurements are due to Najork *et al.* [34].

Because Naiad eliminates this per-iteration cost, it can also use algorithms that perform more and sparser iterations. Compared to the published approaches, the incremental algorithms for weakly connected components (WCC) and approximate shortest paths (ASP) do less work and exchange substantially less data over the network, but require many more iterations to converge. Our implementations of PageRank, strongly connected components (SCC), WCC, and ASP in Naiad require 30, 161, 49, and 70 lines of non-library code respectively.

Several systems for iterative graph computation have adopted the computation of PageRank on a Twitter follower graph [21] as a standard benchmark. The graph contains 42M nodes and 1.5B edges, and is approximately 6 GB on disk. In Figure 7a we compare an implementation of PageRank using sparse matrix-vector multiplication to the published results for PowerGraph [16], which were measured on more powerful hardware (EC2 cluster-compute instances with 10 Gbps Ethernet). Each data point is the average of 10 successive iterations. We present two "native" Naiad approaches: one partitions edges by source vertex (Naiad Vertex), and the other partitions edges using a space-filling curve (Naiad Edge), similar in spirit to PowerGraph's edge partitioning which optimizes a vertex cut objective. These implementations require 30 and 547 lines of code respectively, where many of the 547 lines could be re-used for other programs in the GAS model [16]. We also present the results of an implementation that uses the Naiad port of the Pregel [27] abstraction, taking 38 lines of code.

While the amount of computation and communication varies slightly according to the variant of the algorithm used, the dominant difference in running times comes from layering the algorithm on different abstractions; for example the Pregel abstraction introduces overhead by supporting features such as graph mutation while the Naiad Edge implementation includes specialized dataflow vertices that use the low-level Naiad API. A strength of Naiad is that most developers, seeking simplicity, can build on high-level libraries, while crucial vertices can be implemented using the low-level API when higher performance is essential.

6.2 Batch iterative machine learning

Vowpal Wabbit (VW) is an open-source distributed machine learning library [17]. It performs an iteration of logistic regression in three phases: each process updates its local state; processes independently perform training on local input data; and finally all processes jointly perform a global average (AllReduce) to combine their local updates. Ideally, for a fixed input, the duration of the first and third phases should be independent of the number of processes, and the duration of the second phase should decrease linearly with the number of processes.

We modify VW so that the first and second phases run inside a Naiad vertex. The third phase uses a Naiad implementation of the AllReduce operation. Figure 7b shows the speedup for an iteration of logistic regression on 312M input records using VW's BFGS optimizer, compared to a single computer running unmodified VW. The reduced vector is 268 MB, and each computer runs three VW processes, which fill 16 GB of RAM.

The constant-time cost of the first and third phases prevents scaling past 32 computers, but Naiad's AllReduce implementation gives an asymptotic performance improvement of 35%. VW uses a binary tree to reduce and broadcast updates, while the Naiad implementation uses a data parallel AllReduce with each of k workers reducing and broadcasting $1/k$ of the vector. VW's algorithm scales better on hierarchical networks, but our data

parallel variant is better suited to small clusters where the switches have full bisection bandwidth. The tree-based algorithm is inherently more susceptible to stragglers, and does not optimize communication between processes on the same computer, adding unnecessary network traffic. We wrote a tree-based AllReduce in Naiad for comparison, and verified that it has the same performance as the native VW implementation.

The experiment shows that Naiad is competitive with a state-of-the-art custom implementation for distributed machine learning, and that it is straightforward to build communication libraries for existing applications using Naiad's API. Our AllReduce implementation requires 300 lines of code, around half as many as VW's AllReduce, and the Naiad code is at a much higher level, abstracting the network sockets and threads being used.

6.3 Streaming acyclic computation

Kineograph ingests continually arriving graph data, takes regular snapshots of the graph for data parallel computation, and produces consistent results as new data arrive [10]. The system is partitioned into ingest nodes and compute nodes, making direct performance comparisons complex. When computing the *k-exposure* metric for identifying controversial topics on Twitter, Kineograph processes up to 185,000 tweets per second (t/s) on 32 computers with comparable hardware to ours, taking an average of 90 s to reflect the input in its output. Reducing the ingestion rate can shrink this delay to 10 s.

We implement k-exposure in 26 lines of code using standard data parallel operators of `Distinct`, `Join`, and `Count`. When run on the same Twitter stream as Kineograph, using 32 computers and ingesting 1,000 tweets per epoch on each machine, the average throughput over five runs is 482,988 t/s with no fault-tolerance, 322,439 t/s with checkpoints every 100 epochs, and 273,741 t/s with continual logging. Figure 7c presents the distributions of latencies for the three approaches: continual logging imposes overhead on each batch, but the overhead of periodic snapshots is visible only in the tail when some batches are delayed by up to 10 s. In each case, all responses return within a few seconds, and the respective median latencies are 40 ms, 40 ms, and 85 ms. The difference in latency arises in part because Kineograph synchronously replicates input data before computation begins, whereas Naiad has the flexibility to report outputs before it has made its state durable.

6.4 Streaming iterative graph analytics

Finally, we bring together several of the programming patterns that Naiad handles well, returning to the analysis task motivated in Figure 1. The goal is to ingest
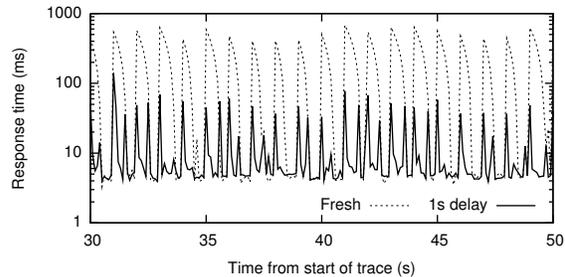

Figure 8: Time series of response times for interactive queries on a streaming iterative graph analysis (§6.4). The computation receives 32,000 tweets/s, and 10 queries/s. "Fresh" shows queries being delayed behind tweet processing; "1 s delay" shows the benefit of querying stale but consistent data.

a continually arriving stream of tweets, extract hashtags and mentions of other users, compute the most popular hashtag in each connected component of the graph of users mentioning other users, and provide interactive access to the top hashtag in a user's connected component.

The dataflow graph follows the outline in Figure 1. There are two input stages: one for the stream of tweets (each containing a user name and the raw tweet text) and the other for requests, specified by a user name and query identifier. The tweets feed in to an incremental connected components computation [28]. To produce the top hashtag for each component, the computation extracts the hashtags from each tweet, joins each hashtag with the component ID (CID) for the user who tweeted it, and groups the results by CID. Incoming queries are joined with the CIDs to get the user's CID, and then with the top hashtags to produce the top hashtag from that component. The logic of the program, not including standard operators and an implementation of connected components [28], requires 27 lines of code.

We add a new query once every 100 ms, and assess the latency before Naiad returns the result to the external program. To generate a constant volume of input data, we introduce 32,000 tweets per second, which is higher than the approximate rate of 10,000 tweets per second in our dataset. We schedule data input according to real time rather than processing the trace as quickly as possible, in order to analyze the effect on latency of updates and queries arriving at different rates.

Figure 8 plots two time series of responses. In the first ("Fresh") all responses are produced in less than one second, but the "shark fin" motif indicates that queries are queued behind the work to update the component structure and popular hashtags, which takes 500–900 ms, because a correct answer cannot be provided until this work completes. We can exploit Naiad's sup-

port for overlapped computation by trading off responsiveness for staleness. The second time series ("1 s delay") shows that response times decrease sharply when the queries refer to computed data that are one second stale, rather than the data that are concurrently being processed. When using one-second-stale data most response times are less than 10 ms, with occasional peaks as high as 100 ms when the CID computation interferes with query execution. A scheduling policy that favors query processing could achieve still lower latencies.

7 Related work

Dataflow Recent systems such as CIEL [30], Spark [42], Spark Streaming [43], and Optimus [19] extend acyclic batch dataflow [15, 18] to allow dynamic modification of the dataflow graph, and thus support iteration and incremental computation without adding cycles to the dataflow. By adopting a batch-computation model, these systems inherit powerful existing techniques including fault tolerance with parallel recovery; in exchange each requires centralized modifications to the dataflow graph, which introduce substantial overhead that Naiad avoids. For example, Spark Streaming can process incremental updates in around one second, while in Section 6 we show that Naiad can iterate and perform incremental updates in tens of milliseconds.

Stream processing systems support low-latency dataflow computations over a static dataflow graph, using *punctuations* in the stream of records [38] to signal completeness. Punctuations can implement blocking operators such as `GROUP BY` [38], but do not support general iteration. MillWheel [5] is a recent example of a streaming system with punctuations (and sophisticated fault-tolerance) that adopts a vertex API very similar to Naiad's, but does not support loops. Chandramouli *et al.* propose the flying fixed-point operator [9] to handle cyclic streams when dataflows do not allow record retraction. In contrast, Naiad can execute algorithms that use retractions, such as sliding-window connected components and strongly connected components.

Previous systems have constructed cyclic dataflow graphs for purposes such as distributed overlays [24] and routing protocols [23, 25], packet processing in software [20], and high-throughput server design [39]. Since none of these applications requires the computation of consistent outputs, they do not contain any mechanism for coordinating progress around cycles.

Asynchronous computation Several recent systems have abandoned synchronous execution in favor of a model that asynchronously updates a distributed shared data structure, in order to achieve low-latency incremen-

tal updates [10, 36] and fine-grained computational dependencies [16, 26]. Percolator [36] structures a web indexing computation as triggers that run when new values are written into a distributed key-value store. Several subsequent systems use a similar computational model, including Kineograph [10], Oolong [29], and Maiter [46]. GraphLab [26] and PowerGraph [16] offer a different asynchronous programming model for graph computations, based on a shared memory abstraction.

These asynchronous systems are not designed to execute dataflow graphs so the notion of completeness of an epoch or iteration is less important, but the lack of completeness notifications makes it hard to compose asynchronous computations. Although GraphLab and PowerGraph provide a global synchronization mechanism that can be used to write a program that performs one computation after another [26, §4.5], they do not achieve task- or pipeline-parallelism between stages of a computation. Naiad allows programs to introduce coordination only where it is required, to support hybrid asynchronous and synchronous computation.

8 Conclusions

Naiad's performance and expressiveness demonstrate that timely dataflow is a powerful general-purpose low-level programming abstraction for iterative and streaming computation. Our approach contrasts with that of many recent data processing projects, which tie new high-level programming patterns to specialized system designs [10, 16, 26, 27]. We have shown that Naiad can implement the features of many of these specialized systems, with equivalent performance, and can serve as a platform for sophisticated applications that no existing system supports.

We believe that separating systems design into a common platform component and a family of libraries or domain-specific languages is good for both users and researchers. Researchers benefit from the ability to differentiate advances in high-level abstractions from advances in the design and implementation of low-level systems, while users benefit from a wider variety of composable programming patterns and fewer, more fully realized systems.

Acknowledgments

References

[1] The ClueWeb09 Dataset.
http://lemurproject.org/clueweb09.

[2] Parallel Data Warehouse.
http://www.microsoft.com/en-us/
sqlserver/solutions-technologies/
data-warehousing/pdw.aspx.

[3] Storm: Distributed and fault-tolerant realtime computation. http://storm-project.net/.

[4] M. Abadi, F. McSherry, D. G. Murray, and T. L. Rodeheffer. Formal analysis of a distributed algorithm for tracking progress. In *Proceedings of the IFIP Joint International Conference on Formal Techniques for Distributed Systems*, June 2013.

[5] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. MillWheel: fault-tolerant stream processing at Internet scale. In *Proceedings of the 39th International Conference on Very Large Data Bases (VLDB)*, Aug. 2013.

[6] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhyey, P. Pately, B. Prabhakarz, S. Senguptay, and M. Sridharany. Data Center TCP (DCTCP). In *Proceedings of the ACM International Conference on Applications, Technologies, Architectures and Protocols for Computer Communications (SIGCOMM)*, Aug. 2010.

[7] P. Alvaro, N. Conway, J. M. Hellerstein, and W. R. Marczak. Consistency analysis in Bloom: a CALM and collected approach. In *Proceedings of the 5th Conference on Innovative Data Systems Research (CIDR)*, Jan. 2011.

[8] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. ul Haq, M. I. ul Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2011.

[9] B. Chandramouli, J. Goldstein, and D. Maier. On-the-fly progress detection in iterative stream queries. *Proceedings of the Very Large Database Endowment (PVLDB)*, 2(1):241–252, Aug. 2009.

[10] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *Proceedings of the EuroSys Conference*, Apr. 2012.

[11] J. Cipar, Q. Ho, J. K. Kim, S. Lee, G. R. Ganger, G. Gibson, K. Keeton, and E. Xing. Solving the straggler problem with bounded staleness. In *Proceedings of the 14th Workshop on Hot Topics in Operating Systems (HotOS)*, May 2013.

[12] D. D. Clark. Window and acknowledgement strategy in TCP. RFC 813, July 1982.

[13] N. Conway, W. R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. Logic and lattices for distributed programming. In *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC)*, Oct. 2012.

[14] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, Feb. 2013.

[15] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2004.

[16] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2012.

[17] D. Hsu, N. Karampatziakis, J. Langford, and A. Smola. Parallel online learning. In R. Bekkerman, M. Bilenko, and J. Langford, editors, *Scaling Up Machine Learning: Parallel and Distributed Approaches*. Cambridge University Press, Dec. 2011.

[18] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the EuroSys Conference*, Mar. 2007.

[19] Q. Ke, M. Isard, and Y. Yu. Optimus: A dynamic rewriting framework for execution plans of data-parallel computation. In *Proceedings of the EuroSys Conference*, Apr. 2013.

[20] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, Aug. 2000.

[21] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *Proceedings of the 19th International World Wide Web Conference (WWW)*, Apr. 2010.

[22] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier. Out-of-order processing: a new architecture for high-performance stream systems. *Proceedings of the Very Large Database Endowment (PVLDB)*, 1(1):274–288, Aug. 2008.

[23] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ranakrishnan, T. Roscoe, and I. Stoica. Declarative networking: language, execution and optimization. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, June 2006.

[24] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2005.

[25] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative routing: extensible routing with declarative queries. In *Proceedings of the ACM International Conference on Applications, Technologies, Architectures and Protocols for Computer Communications (SIGCOMM)*, Aug. 2005.

[26] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new parallel framework for machine learning. In *Proceedings of the 26th Conference on Uncertainty in Artificial Intelligence (UAI)*, July 2010.

[27] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, June 2010.

[28] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard. Differential dataflow. In *Proceedings of the 6th Conference on Innovative Data Systems Research (CIDR)*, Jan. 2013.

[29] C. Mitchell, R. Power, and J. Li. Oolong: asynchronous distributed applications made easy. In *Proceedings of the 3rd Asia-Pacific Workshop on Systems (APSys)*, July 2012.

[30] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. CIEL: a universal execution engine for distributed dataflow computing. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Mar. 2011.

[31] D. Nagle, D. Serenyi, and A. Matthews. The Panasas ActiveScale storage cluster: Delivering scalable high bandwidth storage. In *Proceedings of the ACM/IEEE Supercomputing Conference (SC)*, Nov. 2004.

[32] J. Nagle. Congestion control in IP/TCP internetworks. RFC 896, Jan. 1984.

[33] M. Najork. The scalable hyperlink store. In *Proceedings of the 20th ACM Conference on Hypertext and Hypermedia*, June 2009.

[34] M. Najork, D. Fetterly, A. Halverson, K. Kenthapadi, and S. Gollapudi. Of hammers and nails: an empirical comparison of three paradigms for processing large graphs. In *Proceedings of the 5th ACM International Conference on Web Search and Data Mining (WSDM)*, Feb. 2012.

[35] J. Pelissier. Providing quality of service over InfiniBand™Architecture fabrics. In *Proceedings of the 8th IEEE Symposium on High Performance Interconnects (HOT Interconnects)*, 2000.

[36] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2010.

[37] D. P. Reed and R. K. Kanodia. Synchronization with eventcounts and sequencers. *Communications of the ACM*, 22(2):115–123, Feb. 1979.

[38] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3), May/June 2002.

[39] M. Welsh, D. Culler, and E. Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2001.

[40] R. Xin, J. Gonzalez, M. Franklin, and I. Stoica. GraphX: A resilient distributed graph system on spark. In *Proceedings of the Graph Datamanagement Experiences and Systems (GRADES) Workshop*, June 2013.

[41] 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 *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2008.

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

[43] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized Streams: Fault-tolerant streaming computation at scale. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, Nov. 2013.

[44] M. Zaharia, A. Konwinski, A. D.Joseph, R. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2008.

[45] Y. Zhang, Q. Gao, L. Gao, and C. Wang. PrIter: A distributed framework for prioritized iterative computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SoCC)*, Oct. 2011.

[46] Y. Zhang, Q. Gao, L. Gao, and C. Wang. Accelerate large-scale iterative computation through asynchronous accumulative updates. In *Proceedings of the 3rd ACM Workshop on Scientific Cloud Computing (ScienceCloud)*, June 2012.