

Incremental, Iterative Data Processing with Timely Dataflow

By Derek G. Murray, Frank McSherry, Michael Isard, Rebecca Isaacs, Paul Barham, and Martín Abadi

Abstract

We describe the timely dataflow model for distributed computation and its implementation in the Naiad system. The model supports stateful iterative and incremental computations. It enables both low-latency stream processing and high-throughput batch processing, using a new approach to coordination that combines asynchronous and fine-grained synchronous execution. We describe two of the programming frameworks built on Naiad: GraphLINQ for parallel graph processing, and differential dataflow for nested iterative and incremental computations. We show that a general-purpose system can achieve performance that matches, and sometimes exceeds, that of specialized systems.

1. INTRODUCTION

This paper describes the *timely dataflow* model for iterative and incremental distributed computation, and the Naiad system that we built to demonstrate it. We set out to design a system that could simultaneously satisfy a diverse set of requirements: we wanted efficient high-throughput processing for bulk data-parallel workloads; stateful computations supporting queries and updates with low latency (on the order of milliseconds); and a simple yet expressive programming model with general features like iteration. Systems already exist for batch bulk-data processing,^{6, 13, 27} stream processing,³ graph algorithms,¹¹ machine learning,¹⁵ and interactive ad hoc queries¹⁸; but they are all deeply specialized for their respective domains. Our goal was to find a common low-level abstraction and system design that could be re-used for all of these computational workloads. We were motivated both by the research question of whether such a low-level model could be found, and also by the pragmatic desire to reduce the engineering cost of domain-specific distributed systems by allowing them to share a single highly optimized core codebase.

To understand the difficulty of supporting low-latency, high-throughput, and iterative computations in the same system, we must first think about scheduling and coordination. An easy way to achieve low latency in a distributed system is to use fully decentralized scheduling with no global coordination: workers eagerly process messages sent by other workers and respond based on purely local information. One can write highly complex computations this way—for example using a trigger mechanism²¹—but it is challenging to achieve consistency across the system. Instead we sought a high-level programming model with the abstraction of computing over collections of data using constructs with well-understood semantics, including loops; however, it is hard to translate such a high-level program description into an uncoordinated mass of triggers. At the other extreme,

the easiest way to implement a high-throughput batch system with strong consistency is to use heavyweight central coordination, which has acceptable cost when processing large amounts of data, because each step of the distributed computation may take seconds or even minutes. In such systems it may make sense to insert synchronization barriers between computational steps,⁶ and manually unroll loops and other control flow into explicitly acyclic computation graphs.^{20, 27} The overhead of these mechanisms precludes low-latency responses in cases where only a small amount of data needs to be processed.

Timely dataflow is a computational model that attaches virtual timestamps to events in structured cyclic dataflow graphs. Its key contribution is a new coordination mechanism that allows low-latency asynchronous message processing while efficiently tracking global progress and synchronizing only where necessary to enforce consistency. Our implementation of Naiad demonstrates that a timely dataflow system can achieve performance that matches—and in many cases exceeds—that of specialized systems.

A major theme of recent high-throughput data processing systems^{6, 13, 27} has been their support for transparent fault tolerance when run on large clusters of unreliable computers. Naiad falls back on an older idea and simply checkpoints its state periodically, restoring the entire system state to the most recent checkpoint on failure. While this is not the most sophisticated design, we chose it in part for its low overhead. Faster common-case processing allows more computation to take place in the intervals between checkpointing, and thus often decreases the total time to job completion. Streaming systems are, however, often designed to be highly available³; users of such systems would rightly argue that periodic checkpoints are not sufficient, and that (setting aside the fact that streaming systems generally do not support iteration) a system like MillWheel³ could achieve much higher throughput if it simply dispensed with the complexity and overhead of fault tolerance. In keeping with the philosophy of timely dataflow we believe there is a way to accommodate both lazy batch-oriented and eager high-availability fault tolerance within a single design, and interpolate between them as appropriate within a single system. We have developed a theoretical design for timely dataflow fault tolerance² and are in the process of implementing it.

In the remainder of this paper we first introduce timely dataflow and describe how its distributed implementation

The original version of this paper was entitled “Naiad: A Timely Dataflow System” and was published in the *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (Farmington, PA, Nov. 3-6, 2013), 439–455.

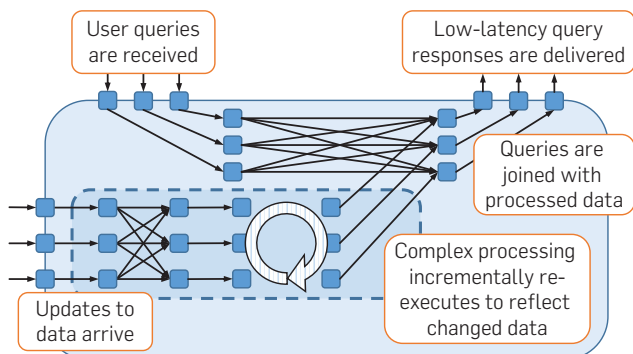
achieves our desiderata (Section 2). We then discuss some applications that we have built on Naiad, including graph computation (Section 3) and differential dataflow (Section 4). Finally we discuss lessons learned and open questions (Section 5). Some of the material in this article was previously published at SOSP 2013 in a paper that describes Naiad in more detail.¹⁹

2. SYSTEM DESIGN AND IMPLEMENTATION

Figure 1 illustrates one type of application that motivated timely dataflow, since it mixes high-throughput iterative processing on large volumes of data with fine-grained, low-latency reads and updates of distributed state. Updates continually arrive at the left, reflecting activity in a social network. The dashed rectangle surrounds an iterative clustering algorithm that incrementally maintains a view of conversation topics, aggregated by the dynamic community structure that the recent activity implies. At the top, incoming queries request topic recommendations that are tailored to particular users and their community interests: these queries are joined with the freshest available clustering to provide high quality and up-to-date results. Before Naiad, no existing system could implement all of these features with acceptable performance. A standard solution might be to write the clustering algorithm in the language of a batch system like MapReduce⁶ or Spark²⁷ and re-run it from scratch every few hours, storing the output in a distributed datastore like Bigtable.⁵ A separate program might target a low-latency streaming system like MillWheel³ and perform a simpler non-iterative categorization of recent updates, saving fresh but approximate recommendations to another table of the distributed store. A third program would accept user queries, perform lookups against the batch and fresh data tables, combine them and return results. While this kind of hybrid approach has been widely deployed, a single program on a single system would be simpler to write and maintain, and it would be much easier to reason about the consistency of its outputs.

Combining these disparate requirements in a high-performance system is challenging, and a crucial first step was to design suitable abstractions to structure the necessary computation. This section starts by explaining the

Figure 1. An application that supports real-time queries on continually updated data. The dashed rectangle represents iterative processing that incrementally updates as new data arrive.



computational model we arrived at, the abstractions we chose, and the reasoning behind them.

2.1. Dataflow

Our first choice was to represent every program as a dataflow graph. Dataflow is a common approach for distributed data processing^{6, 13, 27} because it explicitly encapsulates the boundaries between computations: the nodes of a dataflow graph represent subcomputations, and the directed edges represent the paths along which data is communicated between them. As a result, a system that represents its programs using dataflow can automatically determine subcomputations that can be executed in parallel. It then has a large degree of flexibility in scheduling them, and it can—at least in principle—place, move, and restart nodes independently without changing the semantics of the overall computation.

We based our design on *stateful dataflow*, in which every node can maintain mutable state, and edges carry a potentially unbounded stream of messages. Although statefulness complicates fault tolerance, we believe that it is essential for low-latency computation. Incremental or iterative computations may hold very large indexed data structures in memory and it is essential that an application be able to rapidly query and update these data structures in response to dataflow messages, without the overhead of saving and restoring state between invocations. We chose to require state to be private to a node to simplify distributed placement and parallel execution. One consequence of adopting stateful dataflow is that loops can be implemented efficiently using cycles in the dataflow graph (with messages returning around a loop to the node that stores the state). In contrast, stateless systems^{20, 27} implement iteration using acyclic dataflow graphs by dynamically unrolling loops and other control flow as they execute.

Having settled on stateful dataflow we attempted to minimize the number of execution mechanisms, in order to make timely dataflow systems easier to reason about and optimize. For example, we adopted the convention that all computation in nodes occurs in single-threaded event handlers, which the runtime invokes explicitly. With this convention all scheduling decisions are centralized in a common runtime, making CPU usage more predictable and allowing the system builder to aggressively optimize performance and control latency. It also simplifies the implementation of individual nodes: because the system guarantees that all event handlers will run in a single thread, the application programmer can ignore the complexities of concurrent programming. By encouraging single-threaded node implementations we push programmers to obtain parallelism by adding nodes to the dataflow graph, and force the system builder to ensure low overhead when scheduling a node's computation. The resulting system should be able to interleave many short-lived invocations of different nodes, and be well-suited to performing fine-grained updates with low latency.

Data-parallelism is a standard approach for constructing parallel dataflow graphs from operators whose inputs and outputs are collections of records. A data-parallel operator

includes a *key function* that maps each input record to a key, such that records with different keys can be processed independently. As DeWitt and Gray showed more than 20 years ago, such an operator can be implemented in a dataflow graph by splitting it into multiple nodes, each of which takes responsibility for a disjoint subset of the key space.⁷ Data-parallelism is attractive because the results are identical regardless of how one partitions the key space, so the programmer need only specify an appropriate key function, and the system can automatically choose the degree of parallelism. Our framework libraries provide standard data-parallel operators that can be customized for specific applications.

2.2. Timely dataflow

Applications should produce consistent results, and consistency requires coordination, both across dataflow nodes and around loops. We called new model “timely” dataflow because it depends on logical timestamps to provide this coordination. We started with the goal of supporting general-purpose incremental and iterative computations with good performance, and then tried to construct the narrowest possible programming interface between system and application writer that satisfied our requirements. Our desire for a narrow interface, like our desire for few mechanisms, stems from the belief that it makes systems simpler to understand and engineer.

Asynchronous messages. All dataflow models require some means for one node to send a message along an outgoing edge to another node. In a timely dataflow system, each node implements an `OnRecv` event handler that the system can call when a message arrives on an incoming edge, and the system provides a `Send` method that a node can invoke from any of its event handlers to send a message on an outgoing edge. Messages are delivered asynchronously, which gives the system has great latitude in how the messages are delivered. For example, it can buffer messages between a pair of nodes to maximize throughput. At the other extreme the system may deliver messages via “cut-through,” whereby the `OnRecv` handler for the destination node runs on the same callstack as the source’s `Send` call. Cut-through eliminates buffering altogether, which improves cache performance and enables optimizations such as eager data reduction²⁵ that can drastically reduce memory consumption. Unlike the Synchronous Data Flow model,¹⁴ a timely dataflow node may call `Send` a variable number of times in response to an incoming message; as a result, timely dataflow can represent more programs, but it requires dynamic scheduling of the individual nodes.

Each message in a timely dataflow graph is labeled with a logical timestamp. A timestamp can be as simple as an integer attached to an input message to indicate the batch in which it arrived. Timestamps are propagated through computations and, for example, enable an application programmer to associate input and output data. Timely dataflow also supports more complex multi-dimensional timestamps, which can be used to enforce consistency when dataflow graphs contain cycles, as outlined below.

Consistency. Many computations include subroutines that must accumulate all of their input before generating an

output: consider for example reduction functions like `Count` or `Average`. At the same time, distributed applications commonly split input into small asynchronous messages to reduce latency and buffering as described above. For timely dataflow to support incremental computations on unbounded streams of input as well as iteration, it a mechanism to signal when a node (or data-parallel set of nodes) has seen a consistent subset of the input for which to produce a result.

A *notification* is an event that fires when all messages at or before a particular logical timestamp have been delivered to a particular node. Since a logical timestamp t identifies a batch of records, a notification event for a node at t indicates that all records in that batch have been delivered to the node, and a result can be produced for that logical timestamp. We exposed notifications in the programming model by adding a system method, `NotifyAt(t)`, that a node can call from an event handler to request a notification. When the system can guarantee that no more messages with that timestamp will ever be delivered to the node, it will call the node’s `OnNotify(t)` handler. This guarantee is a global property of the state of the system and relies on a distributed protocol we describe below. Nodes typically use an `OnNotify` handler to send a message containing the result of a computation on a batch of inputs, and to release any temporary state associated with that batch.

Iteration with cyclic graphs. Support for iteration complicates the delivery of notifications, because in a cyclic dataflow graph the input to a node can depend on its output.³ As a result, we had to invent suitable restrictions on the structure of timely dataflow graphs, and on the timestamps that can be affixed to messages, to make the notification guarantee hold. The general model is described in detail elsewhere^{1,19} but the restrictions that we adopted in the Naiad system are easy to explain informally. A Naiad dataflow graph is acyclic apart from structurally nested cycles that correspond to loops in the program. The logical timestamp associated with each event at a node is a tuple of one or more integers, in which the first integer indicates the batch of input that the event is associated with, and each subsequent integer gives the iteration count of any (nested) loops that contain the node. Every path around a cycle includes a special node that increments the innermost coordinate of the timestamp. Finally, the system enforces the rule that no event handler may send a message with a time earlier than the timestamp for the event it is handling. These conditions ensure that there is a *partial order* on all of the pending events (undelivered messages and notifications) in the system, which enables efficient progress tracking.

2.3. Tracking progress

The ability to deliver notifications promptly and safely is critical to a timely dataflow system’s ability to support low-latency incremental and iterative computation with consistent results. For example, the system can use a global

³ MillWheel has a notification—or “Timer”—interface that is similar to the timely dataflow design,³ but since it does not support iteration, the timestamps are simply integers and the graph is acyclic, greatly simplifying progress tracking.

progress tracker to establish the guarantee that no more messages with a particular timestamp can be sent to a node. By maintaining an aggregated view of the pending events in the system, the progress tracker can use the partial order on these events to determine (for each node) the earliest logical time of any subsequent event; this earliest time is monotonic (i.e., it never goes backwards). Moreover, there is an efficient way—sketched below—to compute this earliest time so that notifications are delivered promptly when they come due.

The progress tracker is an “out-of-band” mechanism for delivering notifications. Previous systems have implemented the equivalent of notifications using “in-band” control messages along dataflow edges: for example by requiring nodes to forward a special “punctuation” message on their outgoing edges to indicate that a batch is complete.²⁴ While in-band punctuations might appear to fit better with our philosophy of keeping things simple, the performance benefits of the out-of-band progress tracker design outweighed the cost of the extra complexity. Punctuations are unattractive for data-parallel dataflow graphs because the number of messages that must be sent to indicate the end of a batch is proportional to the number of edges in the graph rather than the number of nodes (as in the out-of-band design). The simplicity of punctuations breaks down when the dataflow can be cyclic, because (i) a node cannot produce a punctuation until it receives punctuations on all of its inputs, and (ii) in a cyclic graph at least one node must have an input that depends on its output. Although punctuations support a limited class of iterative computations,⁴ they do not generalize to nested iteration or non-monotonic operators, and so do not meet our requirements.

Having established the need for out-of-band coordination, we could still have adopted a simpler centralized scheduling discipline, for example triggering nodes to process events in each iteration after the previous was complete. A subtle but powerful property of incrementally updated iterative computation convinced us to pursue superior performance. Consider for example the problem of computing the connected components of a large graph: it might require 200 iterations and be partitioned over 100 worker computers. Now imagine re-running the computation after deleting a single edge from the graph. It would not be surprising if the work done in the second run were identical to that in the first except for, say, eight distinct loop iterations; and if those iterations differed only at two or three workers each. When incrementally updating the computation, a sophisticated implementation can actually be made to perform work only at those 20 or so times and workers, and this is only possible because the out-of-band notification mechanism can “skip over” workers and iterations where there is nothing to do; a design that required the system to step each node around the loop at every iteration would be much less efficient. This example also illustrates a case in which event handlers send messages and request notifications for a variety of times in the future of the events being processed; again, we could have chosen a simpler design that restricted this generality, but we would have lost substantial performance for useful applications. Space does not permit a full treatment of

the node logic needed for such applications, but Section 4 sketches an explanation and provides further references.

2.4. Implementation

Naiad is our high-performance distributed implementation of timely dataflow. It is written in C#, and runs on Windows, Linux, and Mac OS X.^b A Naiad application developer can use all of the features of C#, including classes, structs, and lambda functions, to build a timely dataflow graph from a system-provided library of generic `Stream` objects. Naiad uses deferred execution: at runtime executing a method like `Max` on a `Stream` actually adds a node to an internal dataflow graph representation. The dataflow computation only starts once the graph has been completely built and data are presented to the input nodes. The same program can run on a single computer with one or more worker threads, or—with a simple change in configuration—as a process that communicates with other instances of the same program in a distributed computation. Workers exchange messages locally using shared memory, and remotely using persistent TCP connections between processes. Each dataflow edge transmits a sequence of objects of a particular C# type, and generics are used extensively so that operators and the edges connecting them can be strongly typed.

Performance considerations. To achieve performance that is competitive with more specialized systems, we heavily optimized Naiad’s few primitive mechanisms. In particular, we found it necessary to reduce overheads from serialization for .NET types by adding run-time code generation, and from garbage collection by using value types extensively in the runtime and standard operators. In order to get low-latency responses to small incremental updates and fast loop iterations, we needed to ensure that progress tracking is efficient: notifications are delivered to a node as soon as possible once it cannot be sent any more messages with a given timestamp.

Naiad’s progress tracking protocol is essentially equivalent to distributed reference counting for termination detection or garbage collection.²³ Each event is associated with a graph location (edge or node): a message with the edge it is sent on, and a notification with the node that requests and receives it. Each worker maintains a count for its local view of the number of outstanding events for each pair of location and timestamp. Whenever an event is delivered the progress tracker decrements the corresponding location’s count at the event’s timestamp and increments any counts for messages sent or notifications requested by the event handler, then broadcasts this information to all other workers.

As stated, this protocol would be wildly inefficient, but we made several optimizations that allow workers to accumulate updates and delay sending them without stalling the global computation. As a simple example, if a worker has a pending notification at a node n and timestamp t , it can safely accumulate updates to later timestamps until that notification is delivered. Accumulated updates frequently cancel each other out, so the global broadcast traffic is much less than a naive analysis would suggest.

^b The full source code is available from <https://github.com/TimelyDataflow/Naiad>.

The delivery of notifications defines the critical path for a Naiad computation, and the protocol as implemented can dispatch notifications across a cluster in a single network round-trip. Figure 2 shows that, using the protocol, a simple microbenchmark of notifications in a tight loop performs a global barrier across 64 servers (connected by Gigabit Ethernet) with a median latency of just 750 μ s.

Layering programming abstractions. We wanted to ensure that Naiad would be easy to use for beginners, while still flexible enough to allow experienced programmers to customize performance-critical node implementations. We therefore adopted a layered model for writing Naiad programs. The lowest layer exposes the raw timely dataflow interfaces for completely custom nodes. Higher layers are structured as framework libraries that hide node implementations behind sets of data-parallel operators with related functionality whose inputs and outputs are distributed collections of C# objects.

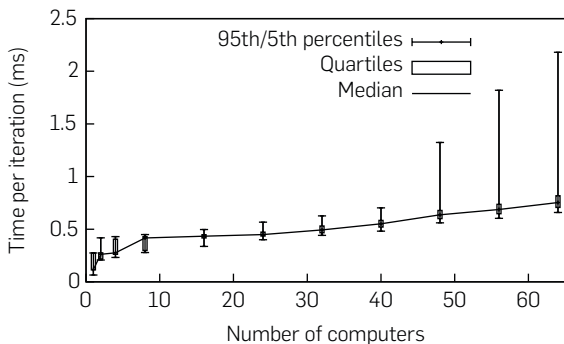
We modeled many of our libraries on the distributed query libraries in DryadLINQ,²⁶ with the added support for graph processing and incremental computation that we discuss in the following sections. Within libraries we can often re-use common implementations; for example most of the LINQ operators in Naiad build on unary and binary forms of a generic buffering operator with an `OnRecv` callback that adds records to a list indexed by timestamp, and an `OnNotify(t)` method that applies the appropriate transformation to the list or lists for time t . In many cases we were able to specialize the implementation of operators that require less coordination: for example `Concat` immediately forwards records from either of its inputs, `Select` transforms and outputs data without buffering, and `Distinct` outputs a record as soon as it is seen for the first time.

The ease of implementing new frameworks as libraries on Naiad enabled us to experiment with various distributed processing patterns. In the following sections, we elaborate on the frameworks that we built for graph processing (Section 3) and differential dataflow (Section 4).

3. GRAPH PROCESSING ON NAIAD

It is challenging to implement high-performance graph algorithms on many data processing systems. Distributed graph

Figure 2. The median latency of a global barrier implemented using notifications in a cycle is just 750 μ s on 64 machines. Error bars show the 95th percentile latencies in each configuration.

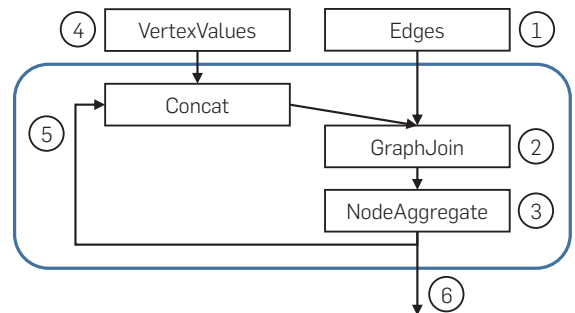


algorithms typically require efficient communication, coordination at fine granularity, and the ability to express iterative algorithms. These challenges have spurred research into specialized distributed graph-processing systems¹¹ and—more recently—attempts to adapt dataflow systems for graph processing.¹² We used a variety of graph algorithms to evaluate both the expressiveness of the timely dataflow programming model and the performance of our Naiad implementation. To avoid confusion in this section we use the term “operator” for dataflow nodes, and “graph,” “node,” and “edge” refer to elements of the graph that is being analyzed by a program running on Naiad unless otherwise qualified.

To understand how we implement graph algorithms on Naiad, it is instructive to consider the Gather-Apply-Scatter (GAS) abstraction of Gonzalez et al.¹¹ In the GAS abstraction, a graph algorithm is expressed as the computation at a node in the graph that (i) gathers values from its neighbors, (ii) applies an update to the node’s state, and (iii) scatters the new value to its neighbors. Figure 3 shows how we express this abstraction as a timely dataflow graph. The first step is to load and partition the edges of the graph (1). This step might use a simple hash of the node ID, or a more advanced partitioning scheme that attempts to reduce the number of edges that cross partition boundaries. The core of the computation is a set of stateful *graph-join* operators (2), which store the graph in an efficient in-memory data structure that is optimized for random node lookup. The graph-join effectively computes the inner join of its two inputs—the static (*src, dst*) edge relation, and the iteratively updating (*src, val*) state relation—and has the effect of scattering the updated state values along the edges of the graph. A set of stateful *node-aggregate* operators (3) perform the gather and apply steps: they store the current state of each node in the graph, gather incoming updates from the neighbors (i.e., the output of the graph-join), apply the final value to each node’s state, and produce it as output. To perform an iterative computation, the node-aggregate operators take the initial value for each node in the first iteration (4), feed updated state values around the back-edge of the loop (5), and produce the final value for each node after the algorithm reaches a fixed point (6).

Depending on the nature of the algorithm, it may be possible to run completely asynchronously, or synchronize after each iteration. In our experience, the most efficient implementation of graph algorithms like PageRank or weakly

Figure 3. Illustration of a graph algorithm as a timely dataflow graph.



connected components uses `OnRecv` to aggregate incoming values to the node-aggregate operator asynchronously, and `OnNotify` to produce new aggregated states for the nodes synchronously in each iteration. Because it is possible to coordinate at timescales as short as a millisecond, more complex graph algorithms benefit from dividing iterations into synchronous sub-iterations, using the prioritization technique that we briefly describe in Section 4.

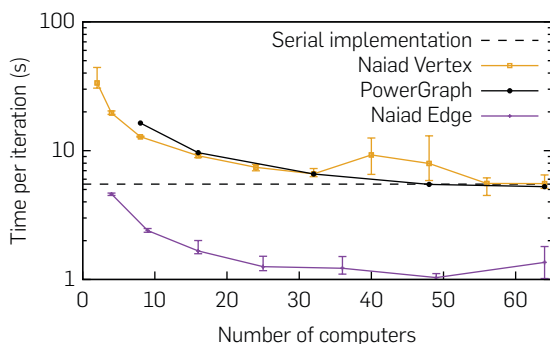
Motivated by the dataflow in Figure 3, we implemented the GraphLINQ framework on Naiad. GraphLINQ extends the LINQ programming model—with its higher-order declarative operators over collections, such as `Select`, `Where`, and `GroupBy`—with `GraphJoin`, `NodeAggregate`, and `Iterate` operators that implement the specialized dataflow nodes depicted in Figure 3. GraphLINQ allows the programmer to use standard LINQ operators to define the dataflow computation that loads, parses, and partitions the input data as a graph, and then specify a graph algorithm declaratively. A simple implementation of PageRank is just nine lines of GraphLINQ code.

When implementing graph algorithms on a dataflow system, a common concern is that the generality of the system will impose a performance penalty over a specialized system. To evaluate this overhead, we measured the performance of several implementations of PageRank on a publicly available crawl of the Twitter follower graph, with 42 million nodes and 1.5 billion edges.^c Figure 4 compares two Naiad implementations of PageRank to the published results for PowerGraph,¹¹ which were measured on comparable hardware.^d We present two different implementations of PageRank on Naiad. The first (“Naiad Vertex”) uses a simple hash function to partition the nodes of the Twitter graph between the workers, and performs all processing for each node on a single worker; this implementation performs similarly to the best PowerGraph implementation, taking approximately 5.55 s per iteration on 64 machines. The more advanced (“Naiad Edge”) implementation uses

^c <http://an.kaist.ac.kr/traces/WWW2010.html>.

^d The Naiad results were computed using two racks of 32 servers, each with two quad-core 2.1 GHz AMD Opteron processors, 16 GB of RAM, and an Nvidia NForce Gigabit Ethernet NIC. The PowerGraph results were computed using 64 Amazon EC2 `cc1.4xlarge` instances, each with two quad-core Intel Xeon X5570 processors, 23 GB of RAM, and 10Gbit/s networking.¹¹

Figure 4. Time per iteration for PageRank on the Twitter follower graph, as the number of machines is varied.



an edge-based partitioning in the spirit of PowerGraph’s edge partitioning with a vertex cut objective, but based on a space-filling curve¹⁶; it outperforms PowerGraph by a factor of 5, taking just 1.03 s per iteration on 49 machines. Figure 4 plots a single-threaded baseline for the PageRank operation, using a late-2014 MacBook Pro with 16 GB of RAM: using a similar data layout to the advanced Naiad implementation, this implementation takes 5.25 s per iteration.

4. DIFFERENTIAL DATAFLOW

Differential dataflow is a computational framework that we developed to efficiently execute and incrementally update iterative data-parallel computations. The framework comprises algorithms, data structures, and dataflow graph constructs layered atop a timely dataflow system.¹⁷

4.1. Incremental view maintenance

Differential dataflow is a generalization of *incremental view maintenance*, a useful technique from database systems. Incremental view maintenance can be implemented as a dataflow graph of data-parallel nodes. Each node continually receives records and maintains the correct output for their accumulation. Because the node implementations are data-parallel, they only need to revisit previously received input records with the same keys as newly arriving inputs. Looking at only these records, the node can determine how the output must be corrected (if at all) to reflect the new input. By producing and communicating only changed output records, the node informs downstream nodes of the relatively few keys they must reconsider. The system as a whole performs work only when and where actual changes occur.

Incremental view maintenance is the basis for many successful stream processing systems³ and graph processing systems.⁸ In a stream processing system, a small per-record update time means that the system can execute with very low latency compared to batch systems. In an incremental graph processing system, the time to perform a round of message exchanges depends only on the number of messages exchanged rather than the total number of nodes or edges. Despite its value for both stream and graph processing systems, incremental view maintenance is not suitable for combining the two.

4.2. From incremental to differential dataflow

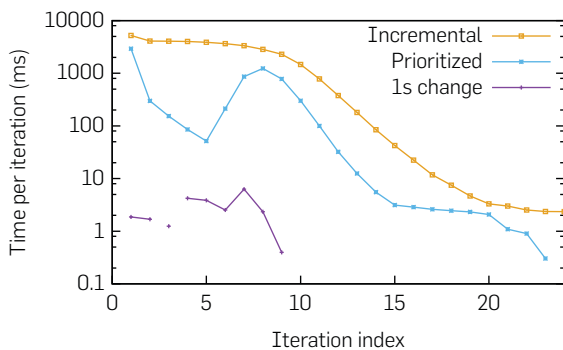
Differential dataflow provides the ability to combine incremental and iterative updates by removing the implicit assumption that time is totally ordered; instead it indexes and accumulates records according to partially ordered timestamps. Consider a graph processing system that accepts incremental updates to its node and edge sets, and correctly updates the output of an iterative computation. This system must deal with multiple types of updates, due to both iterations progressing and inputs changing; differential dataflow distinguishes these types of updates using multi-dimensional logical timestamps. When a new record arrives, the implementation constructs the accumulation needed to determine the new output from all records with timestamps less than or equal to that of the new record. Concretely, consider the example

of timestamps (epoch, iteration) for multiple rounds of an iterative computation that receives multiple epochs of updated input. Using the partial order $(a, b) \leq (x, y)$ iff $a \leq x \wedge b \leq y$ we can get both the standard streaming and graph processing behavior at once: a timestamp (epoch, 0) collects all updates $(i, 0)$ with $i \leq \text{epoch}$, and a timestamp $(0, \text{round})$ collects all updates $(0, j)$ with $j \leq \text{round}$. Further, a timestamp (epoch, round) can take advantage of exactly those records that are useful for it: those at timestamp (i, j) where $i \leq \text{epoch}$ and $j \leq \text{round}$. Records at later epochs or rounds can be ignored.

Figure 5 shows, for different implementation strategies, the execution time for each iteration of a graph processing computation: namely, weakly connected components (via label propagation) on a graph derived from a 24-h window of Twitter mentions. Each vertex represents a user, and it repeatedly exchanges the smallest user ID it has seen so far (including its own) with its neighbors. As the computation proceeds, labels eventually stop changing and converge to the smallest user ID in each connected component. The implementation strategies are as follows:

- **Stateless** batch execution (not shown) repeatedly recomputes all labels in each iteration, and does a constant number of updates as the computation progresses. This is the baseline version that could be implemented on top of MapReduce.
- **Incremental** dataflow uses incremental view maintenance to improve on the stateless version. The amount of work decreases as the computation starts to converge and unchanged labels are neither re-communicated nor re-computed.
- **Prioritized** differential dataflow improves on this further by incrementally introducing the labels to propagate, starting with the smallest values (those most likely to be retained at each vertex) and adding larger values only once the small labels have fully propagated. The advantage of introducing small labels earlier is that many vertices (that eventually receive small labels) will no longer propagate the larger labels that they possess during the early iterations, which reduces

Figure 5. The execution time for each iteration of the connected components algorithm, for a graph built from a Twitter conversation dataset. The “1s change” curve shows a sliding window update that requires no work for many of the iterations.



the amount of unproductive communication and computation.

The **1s change** series shows that the amount of work required to update the edge set by sliding the window forward one second—incrementally updating the connectivity structures as well—is vanishingly small by comparison.

Since differential dataflow uses the same representation for incremental and iterative changes to collections, the techniques are composable. Figure 7 shows an implementation of an algorithm for finding the strongly connected components (SCC) of a directed graph. The classic algorithm for SCC is based on depth-first search, which is not easily parallelizable. However, by nesting two connected components queries (Figure 6) inside an outer `FixedPoint`, we can write a data-parallel version using differential dataflow (Figure 7). Strictly speaking the connected components query computes directed reachability, and the SCC algorithm repeatedly removes edges whose endpoints reach different components and must therefore be in different SCCs. Iteratively trimming the graph in alternating directions—by

Figure 6. A connected components algorithm in differential dataflow that uses `FixedPoint` to perform iterative aggregation over node neighborhoods.

```
// produces a (src, label) pair for each node in the graph
Collection<Node> ConnectedComponents(Collection<Edge> edges)
{
  // start each node with its own label, then iterate
  return edges.Select(x => new Node(x.src, x.src))
    .FixedPoint(x => LocalMin(x, edges));
}

// improves an input labeling of nodes by considering the
// labels available on neighbors of each node as well
Collection<Node> LocalMin(Collection<Node> nodes,
  Collection<Edge> edges)
{
  return nodes.Join(edges, n => n.src, e => e.src,
    (n, e) => new Node(e.dst, n.label))
    .Concat(nodes)
    .Min(node => node.src, node => node.label);
}
```

Figure 7. A function to compute strongly connected components in differential dataflow that uses connected components (Figure 6) as a nested iterative subroutine.

```
// returns edges between nodes within a SCC
Collection<Edge> SCC(Collection<Edge> edges)
{
  return edges.FixedPoint(y => TrimAndReverse(
    TrimAndReverse(y)));
}

// returns edges whose endpoints reach the same node, flipped
Collection<Edge> TrimAndReverse(Collection<Edge> edges)
{
  // establish labels based on reachability
  var labels = ConnectedComponents(edges);

  // struct LabeledEdge(a,b,c,d): edge (a,b); labels c, d;
  return edges.Join(labels, x => x.src, y => y.src,
    (x, y) => x.AddLabel1(y))
    .Join(labels, x => x.dst, y => y.src,
    (x, y) => x.AddLabel2(y))
    .Where(x => x.label1 == x.label2)
    .Select(x => new Edge(x.dst, x.src));
}
```

reversing the edges in each iteration—eventually converges to the graph containing only those edges whose endpoints are in the same SCC.

4.3. Implementation

Our implementation of differential dataflow comprises several standard nodes, including `Select`, `Where`, `GroupBy`, and `Join`, as well as a higher-order `FixedPoint` node that iteratively applies an arbitrary differential dataflow expression until it converges to a fixed point. The records exchanged are of the form (data, time, difference), where data is an arbitrary user-defined type, time is a timestamp, and difference is a (possibly negative) integer.

The standard nodes have somewhat subtle implementations that nonetheless mostly follow from the mathematical definition of differential dataflow¹⁷ and the indexing needed to respond quickly to individual time-indexed updates. The `FixedPoint` node introduces a new coordinate to the timestamps of enclosed nodes, and extends “less or equal” and “least upper bound” for the timestamps according to the product order described above (one timestamp is less than or equal to another if all of its coordinates are). An important aspect of the implementation is that all differential dataflow nodes are generic with respect to the type of timestamp as long as it implements “less or equal” and “least upper bound” methods, and this means that they can be placed within arbitrarily nested fixed-point loops. When the fixed point of an expression is computed, the expression’s dataflow subgraph is constructed as normal, but with an additional connection from the output of the subgraph back to its input, via a node that advances the innermost coordinate by one (informally, this advances the iteration count).

5. LESSONS LEARNED AND OPEN QUESTIONS

Timely dataflow demonstrates that it is possible to combine asynchronous messaging with distributed coordination to generate consistent results from complex, cyclic dataflow programs. Naiad further demonstrates that we can build a system that combines the flexibility of a general-purpose dataflow system with the performance of a specialized system.

Our original Naiad implementation used C# as the implementation language. C#’s support for generic types and first-class functions makes it simple to build a library of reusable data-parallel operators like LINQ. The fact that a running C# program has access to its typed intermediate-language representation means that reflection can be used to generate efficient serialization code automatically. The advantage of automatic serialization when writing distributed applications should not be underestimated, since it allows programmers to use native language mechanisms like classes to represent intermediate values without paying the penalty of writing and maintaining serializers for every class.


Some of C#’s productivity benefits come at a cost to performance and we had to work to minimize that cost. The .NET runtime uses a mark-and-sweep garbage collector (GC) to reclaim memory, which simplifies user programs but presents challenges for building an

efficient system based on maintaining a large amount of state in memory. While we were able to use C# value types to reduce the number of pointers on the heap—and hence the amount of GC work required—it was not possible to eliminate GC-related pauses completely. Since building the original version of Naiad we have investigated alternative designs that would reduce the impact of garbage collection: the Broom project shows encouraging improvements in the throughput of Naiad programs using region-based memory allocation,⁹ and a reimplementing of timely dataflow in the Rust language eliminates the garbage collector altogether.⁶

Many distributed dataflow systems exploit deterministic execution to provide automatic fault tolerance,^{13, 20, 27} but Naiad embraces non-determinism and asynchrony to produce results sooner. Furthermore, Naiad vertices can maintain arbitrary state, which makes it non-trivial to generate code that produces a checkpoint of a vertex. As explained in the introduction our current implementation of fault tolerance is based on restoring from a global checkpoint, which requires code in each stateful vertex to produce and consume a checkpoint of its state. Global checkpointing introduces a large amount of skew into the distribution of epoch and iteration execution times, and forces non-failing processes to roll back in the event of a failure. We have developed a model that permits different vertices to implement different checkpointing policies,² and are working on a Naiad implementation of the model, which exposes a range of performance tradeoffs that in many cases allow high-throughput, low-latency, and fault-tolerant execution.

Finally we note that, while Naiad supports the composition of many different models of computation in the same program, it lacks a high-level programming language—such as SQL—and an optimizer that chooses the most appropriate models for a particular task. Other authors have applied program analysis and query optimization techniques to Naiad. Sousa et al.²² achieved speedups over Naiad’s built-in operators by analyzing user-defined functions and generating new operators. Gog et al.¹⁰ achieved encouraging results with Musketeer, which transforms possibly iterative programs written in a high-level language into code that uses a variety of systems including Naiad. Still, we believe that there is scope for a more powerful compiler that can target Naiad’s different libraries, including differential dataflow, and generate optimized vertex code.

Acknowledgments

We worked on Naiad at the Microsoft Research Silicon Valley Lab between 2011 and the lab’s closure in September 2014. We thank Dave Andersen, Amer Diwan, and Matt Dwyer for suggestions on improving this paper. We are grateful to all of our former colleagues who commented on previous versions of the work, and especially to Roy Levin and Mike Schroeder, who created a unique environment in which this kind of research was encouraged and nurtured. 

⁶ <https://github.com/frankmcsherry/timely-dataflow>.

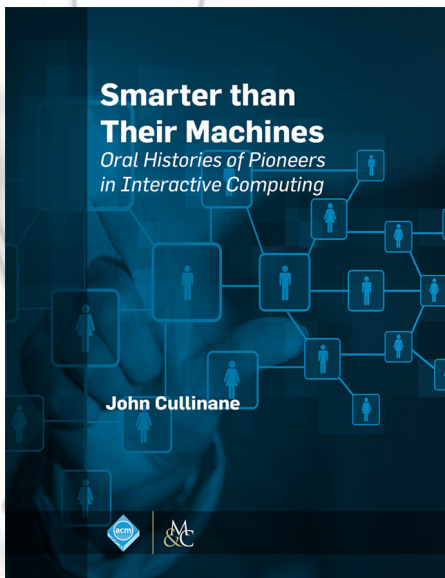
References

1. Abadi, M., Isard, M. Timely dataflow: A model. In *Proc. FORTE* (2015), 131–145.
2. Abadi, M., Isard, M. Timely rollback: Specification and verification. In *Proc. NASA Formal Methods* (April 2015), 19–34.
3. Akidau, T., Balikov, A., Bekiroğlu, K., Chernyak, S., Haberman, J., Lax, R., McVeety, S., Mills, D., Nordstrom, P., Whittle, S. MillWheel: Fault-tolerant stream processing at internet scale. *Proc. VLDB Endow.* 6, 11 (Aug. 2013), 1033–1044.
4. Chandramouli, B., Goldstein, J., Maier, D. On-the-fly progress detection in iterative stream queries. *Proc. VLDB Endow.* 2, 1 (Aug. 2009), 241–252.
5. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E. Bigtable: A distributed storage system for structured data. In *Proc. OSDI* (Nov. 2006), 205–218.
6. Dean, J., Ghemawat, S. Mapreduce: Simplified data processing on large clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113.
7. DeWitt, D., Gray, J. Parallel database systems: The future of high performance database systems. *Commun. ACM* 35, 6 (June 1992), 85–98.
8. Ewen, S., Tzoumas, K., Kaufmann, M., Markl, V. Spinning fast iterative data flows. *Proc. VLDB Endow.* 5, 11 (July 2012), 1268–1279.
9. Gog, I., Giceva, J., Schwarzkopf, M., Vaswani, K., Vytiniotis, D., Ramalingam, G., Costa, M., Murray, D.G., Hand, S., Isard, M. Broom: Sweeping out garbage collection from big data systems. In *Proc. HotOS* (May 2015).
10. Gog, I., Schwarzkopf, M., Crooks, N., Grosvenor, M.P., Clement, A., Hand, S. Musketeer: All for one, one for all in data processing systems. In *Proc. EuroSys* (Apr. 2015).
11. Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proc. OSDI* (Oct. 2012), 17–30.
12. Gonzalez, J.E., Xin, R.S., Dave, A., Crankshaw, D., Franklin, M.J., Stoica, I. GraphX: Graph processing in a distributed dataflow framework. In *Proc. OSDI* (Oct. 2014), 599–613.
13. Isard, M., Budi, M., Yu, Y., Birrell, A., Fetterly, D. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proc. EuroSys* (Mar. 2007), 59–72.
14. Lee, E., Messerschmitt, D.G. Synchronous data flow. *Proc. IEEE* 75, 9 (1987), 1235–1245.
15. Li, M., Andersen, D.G., Park, J.W., Smola, A.J., Ahmed, A., Josifovski, V., Long, J., Shekita, E.J., Su, B.-Y. Scaling distributed machine learning with the parameter server. In *Proc. OSDI* (Oct. 2014), 583–598.
16. McSherry, F., Isard, M., Murray, D.G. Scalability! But at what COST? In *Proc. HotOS* (May 2015).
17. McSherry, F., Murray, D.G., Isaacs, R., Isard, M. Differential dataflow. In *Proc. CIDR* (Jan. 2013).
18. Melnik, S., Gubarev, A., Long, J.J., Romer, G., Shivakumar, S., Tolton, M., Vassilakis, T. Dremel: Interactive analysis of web-scale datasets. *Proc. VLDB Endow.* 3, 1–2 (Sep. 2010), 330–339.
19. Murray, D.G., McSherry, F., Isaacs, R., Isard, M., Barham, P., Abadi, M. Naiad: A timely dataflow system. In *Proc. SOSP* (Nov. 2013), 439–455.
20. Murray, D.G., Schwarzkopf, M., Smowton, C., Smith, S., Madhavapeddy, A., Hand, S. CIEL: A universal execution engine for distributed data-flow computing. In *Proc. NSDI* (Mar. 2011), 113–126.
21. Peng, D., Dabek, F. Large-scale incremental processing using distributed transactions and notifications. In *Proc. OSDI* (Oct. 2010), 251–264.
22. Sousa, M., Dillig, I., Vytiniotis, D., Dillig, T., Gkantsidis, C. Consolidation of queries with user-defined functions. In *Proc. PLDI* (June 2014), 554–564.
23. Tel, G., Mattern, F. The derivation of distributed termination detection algorithms from garbage collection schemes. *ACM Trans. Program. Lang. Syst.* 15, 1 (Jan. 1993), 1–35.
24. Tucker, P.A., Maier, D., Sheard, T., Fegaras, L. Exploiting punctuation semantics in continuous data streams. *IEEE Trans. Knowledge Data Eng.* 15, 3 (2003), 555–568.
25. Yu, Y., Gunda, P.K., Isard, M. Distributed aggregation for data-parallel computing: Interfaces and implementations. In *Proc. SOSP* (Oct. 2009), 247–260.
26. Yu, Y., Isard, M., Fetterly, D., Budi, M., Erlingsson, U., Gunda, P.K., Currey, J. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proc. OSDI* (Dec. 2008), 1–14.
27. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M., Shenker, S., Stoica, I. Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. NSDI* (Apr. 2012).

Derek G. Murray, Michael Isard, Rebecca Isaacs, Paul Barham, and Martín Abadi (dmurray, misard, risaacs, pbar, abadij@google.com) Google, Mountain View, CA.

Frank McSherry (fmcsherry@me.com) is still at large.

Copyright held by owners/authors.



A personal walk down the computer industry road. BY AN EYEWITNESS.

Smarter Than Their Machines: Oral Histories of the Pioneers of Interactive Computing is based on oral histories archived at the Charles Babbage Institute, University of Minnesota. These oral histories contain important messages for our leaders of today, at all levels, including that government, industry, and academia can accomplish great things when working together in an effective way.



ISBN: 978-1-62705-550-5 DOI: 110.1145/2663015
<http://books.acm.org>
<http://www.morganclaypoolpublishers.com/acm>