Composable Incremental and Iterative Data-Parallel Computation with Naiad

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

Abstract

We report on the design and implementation of Naiad, a set of declarative data-parallel language extensions and an associated runtime supporting efficient and composable incremental and iterative computation. This combination is enabled by a new computational model we call *differential dataflow*, in which incremental computation can be performed using a partial, rather than total, order on time.

Naiad extends standard batch data-parallel processing models like MapReduce, Hadoop, and Dryad/DryadLINQ, to support efficient incremental updates to the inputs in the manner of a stream processing system, while at the same time enabling arbitrarily nested fixed-point iteration. In this paper, we evaluate a prototype of Naiad that uses shared memory on a single multi-core computer. We apply Naiad to various computations, including several graph algorithms, and observe good scaling properties and efficient incremental recomputation.

1. Introduction

The combination of declarative programming and data-parallel execution has made it easier to write parallel programs that process large data sets. An early example of this approach used parallel relational databases and SQL [11], but more recently the MapReduce model [10] has shown that a very simple dataflow graph, with two data-parallel stages, is both powerful and relatively simple to implement as a scalable distributed system. Later systems have extended this scalability to directed acyclic dataflow graphs [17], which can support a more expressive, composable set of declarative operators [8, 25, 26]. In this paper, we describe Naiad, which moves away from the batch-oriented, acyclic dataflow systems of the last few years by introducing efficient and composable incremental and iterative computation using a similar declarative programming model.

Data-analysis platforms are increasingly used for real-time decision making, for example to conduct advertising auctions, provide real-time query suggestions on trending topics, or allow users to visualize datasets interactively. Many analysis tasks, particularly those involving graph data, require iterative algorithms. However, data-parallel systems have been developed to support either incremental update or iteration, but not both. Materialized view-maintenance engines [16], stream processing systems [2, 14] and descendants of MapReduce [3, 15] provide efficient support for incremental input, but do not support iterative processing. Likewise, iterative data-parallel frameworks include Datalog [7], recursive SQL databases [12] and several systems that add looping constructs to a MapReduce-like model [6, 13, 20, 21, 28], but most of these provide a constrained programming model (e.g. stratified negation in Datalog), and none supports incremental input processing.

This paper has two main contributions. First, we introduce a new data-parallel execution model, called *differential dataflow*, that allows incremental and iterative constructs to be composed efficiently, and enables programmers to write nested loops that respond quickly to incremental changes in their inputs. The key innovation is that differential dataflow describes changes to collections using a *partial order* rather than a more-conventional total order. Using a partial order enables the collections to evolve in multiple independent dimensions (e.g. loop indices or input versions) without needing to reset or restart one when another changes.

The second contribution is a report on our implementation of differential dataflow in Naiad, which comprises language extensions and a data-parallel runtime system. The implementation adopts the language integrated query (LINQ) feature of the .NET framework [4], in which programmers create dataflow computations out of strongly typed collections and higher-order data-parallel operators such as `Select`, `GroupBy` and `Join`. To this set, we add an operator that iterates a subcomputation to a fixed point and can be nested arbitrarily. Moreover, all differential dataflow graphs that Naiad produces can respond efficiently to incremental changes to any of their inputs.

Naiad unifies the mechanisms for incremental computation with those for iteration and as a consequence, within a fixed-point computation, the amount of work performed is approximately proportional to the number of records that changed in the previous iteration. In many iterative computations, the number of records varying between iterations is often very much smaller than the total number of records, particularly as the iteration nears a fixed point. Naiad's execution of such a computation accelerates as the iterates converge.

We have developed several applications on top of Naiad. Many graph algorithms are ideally suited to the differential dataflow model, and we have implemented algorithms that compute shortest paths, strongly connected components, PageRank, efficient graph partitioning schemes and minimum spanning forests. To demonstrate the generality of the programming model, we have also implemented several other applications, including Smith-Waterman sequence alignment, data cubing, kernel k-means, numerical integration, parallel-prefix and a key-value store. All of these programs support incremental updates to any of their inputs.

This paper describes a multi-processor shared-memory implementation of Naiad, which suffices to present the benefits and complexities of the differential dataflow model. Many of our motivating applications require the resources of a cluster, and this paper lays the groundwork for a more scalable cluster-based implementation. The required systems engineering is non-trivial, and is the subject of ongoing work.

The rest of the paper is structured as follows. Section 2 gives an overview of Naiad concepts as seen by the programmer; and Section 3 introduces differential data flow and the use of partial orders to represent the computation's logical time. Section 4 provides a formal basis for the differential dataflow model. Section 5 describes the multi-core implementation and discusses several optimizations that improve its performance. Section 6 evaluates this implementation on a variety of example workloads. Section 7 discusses related work, and is followed by concluding remarks.

2. Programming with Naiad

The key feature that differentiates Naiad from previous data-parallel frameworks is that it supports incremental and iterative computation in a composable and efficient manner. This section illustrates the language mechanisms that a programmer uses to express these computations using a running example of graph connectivity.

2.1 Declarative programming model

DryadLINQ and FlumeJava allow the programmer to build data-parallel dataflow graphs using declarative operators such as `Select`, `Join`, and `Distinct` defined over collections of strongly-typed records. Naiad adopts this programming model as well: a .NET program is linked against the Naiad library which defines a generic `Collection<T>` type supporting most standard LINQ operators, some new operators, and the necessary mechanisms for their execution.

Figure 1 shows an example Naiad computation. The `LocalMin` function describes a query in which a collection of labeled nodes (`nodes`) are joined (`Join`) with the graph structure (`edges`) to provide each neighbor of a node with that node's label. After including the original labels

```
// 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,
                    node => node.id,
                    edge => edge.srcId,
                    (node, edge) => new Node(edge.dstId,
                                             node.label))
              .Concat(nodes)
              .Min(node => node.id, node => node.label);
}
```

Figure 1. A simple Naiad function over collections.

(`Concat`) each node determines the minimal label among its neighbors and itself (`Min`) and the resulting collection of re-labeled nodes is returned. The execution of the `LocalMin` function calls in to Naiad's supplied `Join`, `Concat`, and `Min` library methods, which construct a corresponding dataflow graph using standard techniques.

Figure 1 and other code fragments elide some details for clarity. The operator syntax is detailed in the Appendix.

2.2 Incremental updates

All Naiad programs are inherently incremental; any Naiad computation responds to changes to its input collections and produces the corresponding changes in its output collections. This is true for individual operators (`Select, Join, Min`) and for all dataflow resulting from their composition. While some streaming systems support only monotonic computations, Naiad allows the changes to collections to be arbitrary additions or subtractions of records. The cost is that Naiad must maintain sufficient state to respond to any change to the input, and this state can often be proportional to the size of the collection itself.

The example program in Figure 2 illustrates how the programmer uses incremental computation in Naiad. The `InputCollection` object `edges` encapsulates an input to the computation and defines the non-blocking `OnNext` method. Each call to `OnNext` on an input collection supplies a new batch of data, corresponding to a new "epoch" of computation. If there are multiple inputs the user must supply a new collection to each input for every epoch by calling `OnNext` on each of them, though some collections may be empty. The `Sync` method blocks until there is no more work on all preceding epochs and the system has quiesced. The `Subscribe` method allows the programmer to register an event handler, in this case `PrintOutput`, on an output collection. The event handler is called once per epoch and supplies the changes to the output that result from the input collections that were introduced in the corresponding epoch.

2.3 Declarative iteration

Imagine taking the output of the incremental dataflow graph resulting from `LocalMin` and connecting it back to the input corresponding to the `nodes` argument. This creates a cycle

```
// create an input for edges, and initialize labels
Collection<Edge> edges = new InputCollection<Edge>(...);
Collection<Node> nodes = edges.Select(x => new Node(x.srcId,
                                                     x.srcId))
                              .Distinct();

// Set up the dataflow graph, register interest in the output
LocalMin(nodes,edges).Subscribe(x => PrintOutput(x));

// introduce initial edge data
Edge[] initialGraph = {...};
edges.OnNext(initialGraph);
edges.Sync();

// PrintOutput() will be called with the labeled nodes
// of the initial graph before edges.Sync() returns

// add changes to the graph
Edge[] graphUpdates = {...};
edges.OnNext(graphUpdates);
edges.Sync();

// PrintOutput() will be called with node labels that
// have changed before edges.Sync() returns

edges.OnCompleted(); // close computation
```

Figure 2. Processing incremental inputs in Naiad.

in which changes to the output propagate back to the input and induce further computation, continuing until no more changes occur—i.e. it has reached a fixed point. We have effectively co-opted Naiad's incremental execution to perform a fixed-point computation.

The programmer expresses fixed-point iteration with a new `FixedPoint` operator on typed collections that takes as an argument a function from and to collections of the same record type. Conceptually, `FixedPoint` iteratively invokes the supplied function, starting from the source collection and using the output of the function at iteration i as the input to the function at iteration $i+1$, until convergence. In fact, only the *differences* between iterations are processed, accelerating the iteration as the computation approaches fixed point.

Figure 3 demonstrates the use of `FixedPoint`, applying the `LocalMin` function inside a fixed point to compute the connected components in a symmetric graph. When this function converges, every node in the same connected component will hold the same label. Figure 4 shows a simplified version of the `ConnectedComponents` dataflow graph. Shaded vertices are added by the Naiad runtime system, as explained in Section 5.1.1.

Figure 5 illustrates how the connected components algorithm runs on a six-node graph: the fixed point is reached after four iterations, represented by the times $t \in \{(0,0), (0,1), (0,2), (0,3)\}$. Each "time" value has two dimensions: the zero-valued first coordinate indicates that the computation is operating on the first epoch of input to an incrementalized computation and the second coordinate indicates the iteration count within the fixed point computation. Section 3 explains this notion of multi-dimensional time in much more detail. Each circle represents a "version" of the label assigned to a graph node at a particular iteration, and the stacking of the labels illustrates the fact that Naiad keeps the full version history. At time $t = (0,0)$ each node is labeled with its own id, and in subsequent iterations it is labeled with the id shown at the top of the "stack" of circles. Should any edge be deleted from the graph, the system can respond efficiently without restarting the computation as we shall see below. The work done by the system and the program's memory footprint are both approximately proportional to the total number of versions produced during the execution.

```
// produces an (id, 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.srcId,x.srcId))
              .Distinct()
              .FixedPoint(x => LocalMin(x, edges));
}
```

Figure 3. Connected components in Naiad.

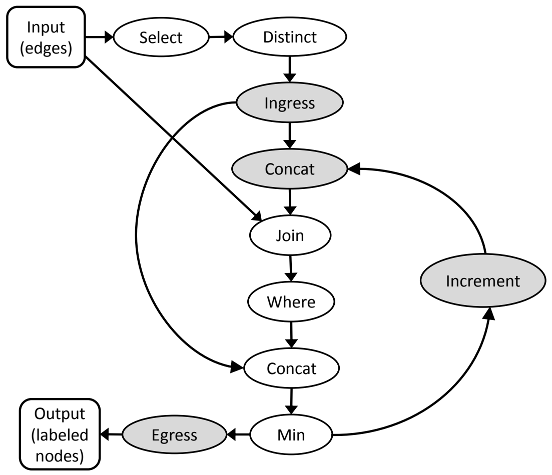


Figure 4. Simplified dataflow graph for connected components.

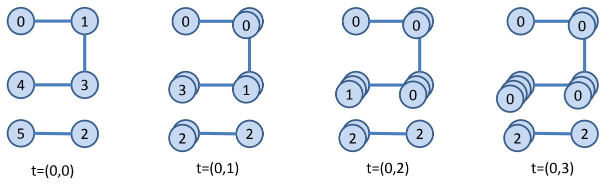


Figure 5. An execution of connected components on a six-node graph.

Figure 6 shows quantitatively how fixed-point iteration with differences progressively reduces the amount of processing required. The measurements were taken by running `ConnectedComponents` over the Amazon product co-purchasing network of June 2003[19], which contains around 400,000 nodes and 3.4 million edges. With this dataset the label of every node is updated after the first iteration, hence we see the number of records double for the second round (the old label is subtracted and the new one

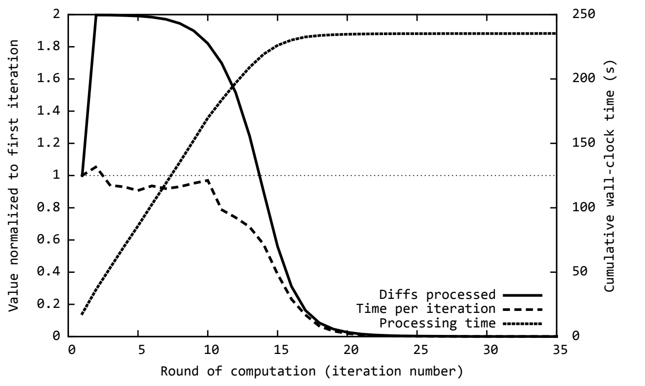


Figure 6. Performance of the fixed point computation in `ConnectedComponents`.

added). The processing time is around 18s for the first iteration, dropping to 5ms on the final iteration, which processes just 18 difference records. In fact the Amazon graph contains one large connected component with $403,364$ nodes, and 6 small components ranging in size from 2 to 15 nodes.

2.4 Incremental and Iterative Computation

The result of the reachability computation in Figure 3 is a collection that reports incremental changes to its output reflecting changes to the input of the computation. Figure 7 shows the work done if, after initially running the connected components program on the input in Figure 5, a second epoch is executed in which the edge between nodes 1 and 3 is removed from the input. In this subsequent epoch the first coordinate of t is 1 rather than 0. Iteration $t = (1, 0)$ shows the same label assignment as $t = (0, 0)$ and because of Naiad's design, explained in detail in the following sections, this unchanged labeling means that the system has almost no work to do for iteration $t = (1, 0)$. In iteration $t = (1, 1)$ the label for node 3 is 3, whereas it was 1 in iteration $t = (0, 1)$. This change from the previous epoch is represented by the removal of label 1 from node 3's history, shown as an unshaded circle. The amount of work done by Naiad is approximately proportional to the number of unshaded circles shown: the changes resulting from the removed edge are propagated forward but most of the work already done in the previous epoch is unaffected.

The approach taken in Naiad is unlike several other declarative programming models including Datalog and recursive SQL, which, while they use incremental evaluation strategies, do not support incremental updates to the inputs of iterative queries. Updating the set of base facts in a Datalog computation cannot be achieved by removing some facts and continuing iteration; undoing the consequences of a fact requires either dependency tracking, or even restarting the computation. It is the use of differential dataflow and a partial order over the logical times, explained over the following sections, that allows Naiad to provide efficient, fully composable incremental and iterative computation.

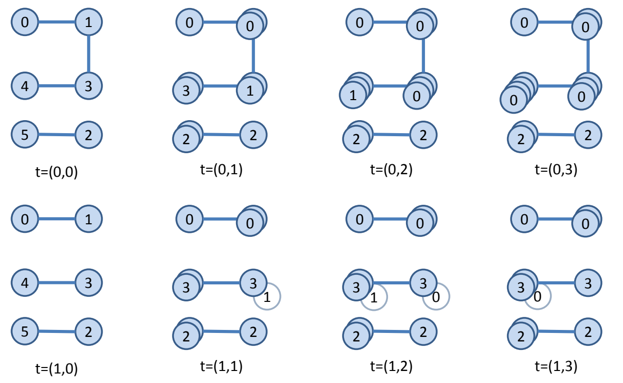


Figure 7. A second epoch of input, corresponding to the removal of an edge, is presented to Naiad after the initial run shown in Figure 5 has completed. Each unshaded circle represents a change in the label history of a graph node.

Concretely, the connected components computation can retire epochs corresponding to single edge updates of the Amazon product graph in an average of 0.49 milliseconds. The user does not have to rewrite the program at all from the original Naiad iterative computation in order to obtain a version that also computes efficient incremental updates.

2.5 Nested and composable iteration

The same mechanisms that allow Naiad to distinguish and react to both changes in the inputs and changes due to iteration also let the system distinguish between changes due to different levels of nested iteration.

Figure 8 presents a concise strongly-connected components (SCC) algorithm based on a `FixedPoint` that uses `ConnectedComponents` as a subroutine (i.e. a nested `FixedPoint`). Strictly speaking, `ConnectedComponents` only computes directed reachability, but can determine that two vertices are in separate strongly-connected components (different labels would imply different SCCs). By repeatedly performing reachability queries on the directed graph and removing edges between different SCCs, in both the forward and reverse directions, we (provably) eventually converge to exactly those edges between vertices in the same SCC.

Although the nesting of `FixedPoint` loops is straightforward for the programmer, the resulting dataflow graph is quite complicated. Figure 9 shows a simplified version with some vertices combined for clarity: in our current implementation the actual dataflow graph for this program contains 58 vertices. Nonetheless, the SCC program accepts incremental updates, and the doubly-nested fixed-point computation responds efficiently to changes in its inputs.

2.6 Prioritization

It is not always the case that one wants to restart an iterative computation from the first iteration on updated data; for many algorithms it is more efficient (and still correct) to restart the iteration from the conclusion of the previous it-

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

// trims edges by forward reachability, transposes
// trims edges by reverse reachability, transposes
Collection<Edge> TrimTwice(Collection<Edge> edges)
{
  return edges.TrimByReachability()
              .Select(x => new Edge(x.dst, x.src))
              .TrimByReachability()
              .Select(x => new Edge(x.dst, x.src));
}

// returns edges whose endpoints can reach the same node
Collection<Edge> TrimByReachability(Collection<Edge> edges)
{
  var labels = ConnectedComponents(edges);

  // struct LabeledEdge(a,b,c): edge a; int labels b, c;
  return edges.Join(labels, x => x.edge.src, y => y.src,
                    (x, y) => new LabeledEdge(x, y, 0))
              .Join(labels, x => x.edge.dst, y => y.src,
                    (x, y) => new LabeledEdge(x.edge, x.label1, y))
              .Where(x => x.label1 == x.label2)
              .Select(x => x.edge);
}
```

Figure 8. Strongly connected components in Naiad.

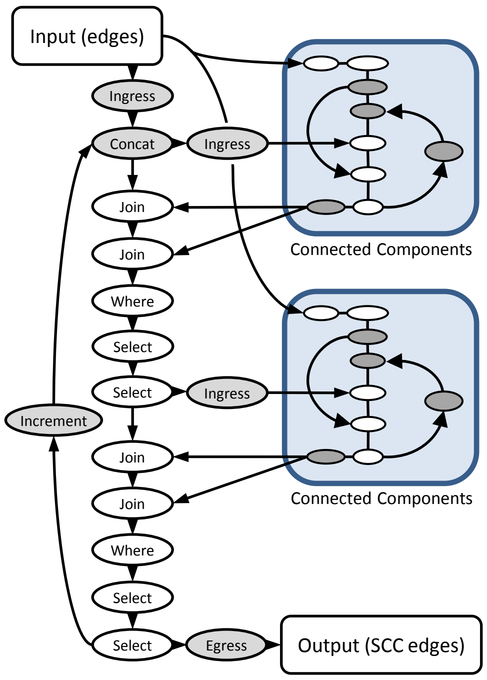


Figure 9. Simplified dataflow for strongly connected components. The outer loop contains two nested instances of the `ConnectedComponents` dataflow graph from Figure 4.

```
// replaces nodes.FixedPoint(...) in ConnectedComponents
nodes.Prioritize(n => Math.Log(1 + n.label),
                 n => n.FixedPoint(x => LocalMin(x, edges)));
```

Figure 10. Extension of `ConnectedComponents` to prioritized computation.

eration. In the specific case of `ConnectedComponents` we might introduce the elements of the `nodes` input progressively, first flooding the graph with the small labels the vertices will prefer, and only then introduce larger labels which will only propagate through those vertices unreached by the small labels.

Naiad provides the ability to sequence fixed-point computation through the `Prioritize` operator. This operator requires the programmer to specify an appropriate function for mapping an input record to an integer representing relative priority, as well as a body to which the prioritization should apply. Elements with smaller priority values are introduced to the fixed point loop before larger-priority data. Figure 10 presents the fragment replacing the `FixedPoint` invocation in the `ConnectedComponents` method of Figure 3. In this case, a sensible `priorityFunction` assigns earlier priority to nodes with a lower identifier.

Prioritization inherently reduces the amount of available parallelism, and, if the priority assignment is too fine-grained, can adversely affect performance. Here, we use $\log(l+1)$ as the priority for label l, which is a good trade-off for this program. As prioritization is also fully composable with other Naiad operators, `SCC` can exploit the improved `ConnectedComponents` without additional modification.

Figure 11 expands on the evaluation of a prioritized computation. An additional time coordinate is added to represent the priority, For example node label 2 is introduced at time $(0, 2, 0)$. The total number of label versions generated using this schedule is much lower than the number generated by the non-prioritized version in Figure 5, even though there are more iterations before convergence.

Figure 12 shows the savings in both records processed and time per iteration when prioritization is used to order the records processed in `ConnectedComponents`. The graph compares the number of differences processed and the time per iteration relative to the same baseline values for the first iteration without prioritization that were used in Figure 6. Since the fixed point is now run multiple times with different priority values, the horizontal axis now measures "rounds" of computation rather than simply the fixed-point iteration number as before. Each round corresponds to a distinct pair of priority and iteration number. Most of the work occurs early on, as labels with low values are flooded. Due to the choice of a priority function that is logarithmic in label value, many more labels are added in later epochs than in earlier ones, and this generates the small peaks in running time towards the right of the graph.

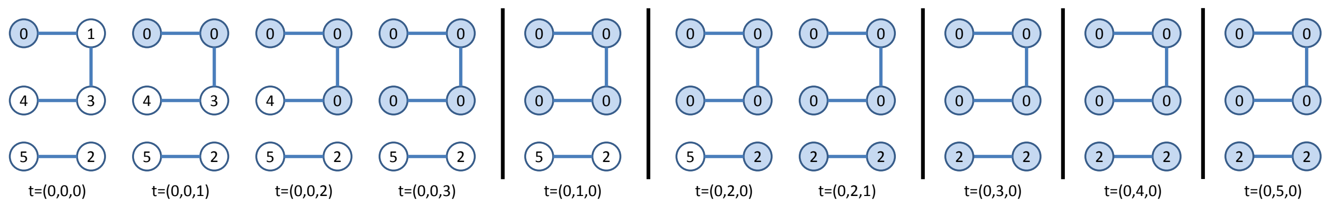


Figure 11. Prioritized version of connected components. For simplicity, here we use the node label as the priority. The work performed by Naiad in each step is proportional to the number of differences from the previous step, often none.

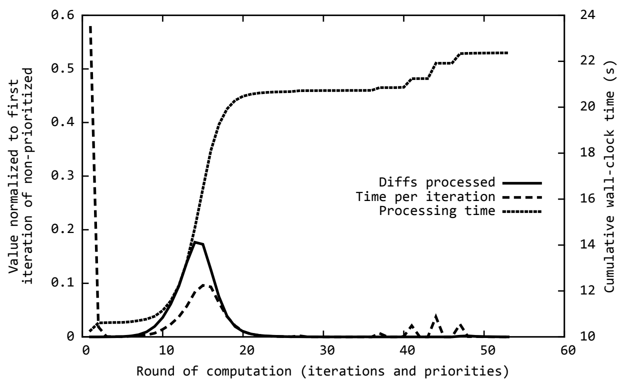


Figure 12. Effects of prioritization on the performance of the fixed point computation in `ConnectedComponents`.

2.7 Multiset semantics

Naiad represents collections using unordered multisets, in which each record has an integral frequency but no information is exposed about, for example, relative order. This makes it much easier to reason about incremental computation since addition and subtraction are commutative, but abandons features like set semantics and ordering. Set semantics can be recovered by applying operators like `Distinct` to multisets, and many of the idioms associated with ordered sequences can be recovered using Naiad's operators. For example, the common use of sorting for "top-k" queries can be supported by operators like `Min` and `Max`, combined with iteration. The addition of sequences to Naiad's type system would introduce some additional implementation complexity but we do not believe it would conflict with any of the other design features.

3. Differential dataflow

The combination of incremental and iterative computation in Naiad is not only straightforward for the programmer, but also results in significant performance optimizations. In this section we describe how differential dataflow enables these optimizations, though we defer the more formal presentation until Section 4.

3.1 Collections and differences

A Naiad programmer manipulates objects representing multiset "collections" of records. In a traditional dataflow execution model, the edges in the dataflow graph transmit collections between operators. An operator receives collections on its input edges, transforms them, and outputs new collections on its output edges. If the collections are unlikely to change much between re-executions, it is natural to only transmit the difference between the new and old collection. Not only can this reduce the amount of data transferred, but in a data-parallel setting the differences identify and restrict which subcomputations may need to be re-executed. For example, if a single record changes in the input to a groupwise aggregation we only need to update the aggregate of the group associated with the record.

Naiad transmits collections exclusively as multiset differences, in which a positive weight w corresponds to adding w copies of the record to the collection, and a negative weight indicates the removal of the corresponding number of copies. Figure 13 illustrates the difference between standard dataflow and differential dataflow using the `Distinct` operator, where the large arrows indicate dataflow edges in and out of the operator. The greyed sets (e.g. $\{A, A, B, C\}$) show the collections that would be transmitted along the edges in a standard dataflow implementation, while the "Input differences" and "Output differences" columns show the differences that Naiad transmits. The input collection varies as time goes from $t = 0 \ldots 3$. A set of records is added at time $t = 1$ and the operator emits the corresponding positive differences. At the next time an "A" is removed from the collection, which updates the internal operator state but has no effect on the output. At the third invocation the remaining "A" record is removed from the input and the operator emits a negative difference, effectively deleting the record from the output. In order to be able to respond to arbitrary differences, the `Distinct` operator maintains state summarizing the differences it has seen. The details of this state are explained in Section 5.

3.2 Partially-ordered logical time

Like other systems, Naiad defines its differences so that collections at a given time t are the accumulation of all preceding differences. Unlike previous systems, Naiad uses a *partial order* over times t, which gives it the flexibility to derive a new collection from more than one previous collection.

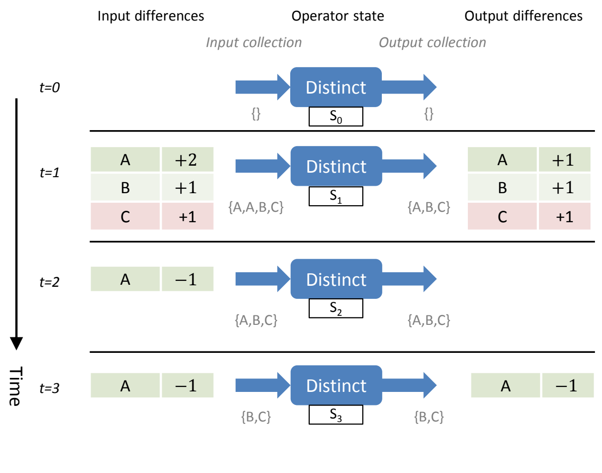


Figure 13. Differential dataflow with `Distinct`, see Section 3.1.

For example, as a fixed-point computation proceeds it is natural to describe each iterate using the difference from the previous iterate. However, if the input collection is then changed, we have at least two possible starting points for the updated iterates. As before, we can compute the difference between successive iterates starting from the changed input collection, doing approximately as much work as the preceding fixed-point computation. However, the updated sequence of iterates might evolve in a similar way to the original sequence of iterates, and we could also justify deriving the updated i^{th} iterate from the *original* i^{th} iterate. A third approach, taken by Naiad, combines the two: differences contributing to both precursors are accumulated (being careful not to double-count) and taken as the basis for differencing. Conceptually, the differences can be laid out in a grid, rather than a sequence of independent lines (as in the first two cases). By preserving this multi-dimensional grid structure of differences, changes to collections resulting from either type of change to the inputs can be efficiently exploited, resulting in a composite reduction in the size of differences and redundant recomputation for many iterative data-parallel computations.

Furthermore, this grid is not restricted to two dimensions. Naiad uses *multi-dimensional lattice times* to support the composition of incremental computation with nested iteration. For ease of exposition, a lattice time in Naiad can be thought of as a tuple of integers. In an incremental computation, the input varies in one dimension, which corresponds to the input epoch number. Upon entering a nested fixed-point computation, all times are extended by one element (corresponding to an iteration counter), and this element is stripped on egress from the fixed point. In general, the lattice time is hidden from the programmer: the `OnNext()` method on the input collection increments the epoch number of the next batch of records, and the structured nesting of fixed-point operators ensures that the correct dimension of lattice times is used throughout a Naiad program.

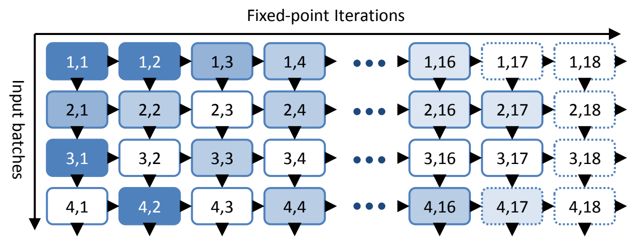


Figure 14. Two-dimensional lattice times in a computation containing repeated invocations of a fixed point loop. Every iteration of the loop increments the "inner" time. Every new batch of inputs advances the "outer" time. The times are taken from a product lattice indicated using the black arrowheads, where $t_1 < t_2$ if there is a path along the arrowheads from t_1 to t_2.

3.3 A hypothetical example

In Figure 14, each box represents the state of a collection in a particular loop iteration and is labeled with its two-dimensional time. The horizontal axis corresponds to iterations of a fixed point loop, while the vertical axis corresponds to batches of input data introduced into the computation. As the program executes, it first runs the fixed point to convergence, shown on the first row. Once the first loop has converged, changes are made to the input and a new fixed point must be computed that reflects these updates. This computation is shown on the second row. Every time a new batch of inputs is supplied, a new row is started, and the updated fixed point computed.

The lattice in Figure 14 is indicated by the black arrowheads: time $t_1 \leq t_2$ if there is a path from t_1 to t_2 following the arrowheads. The boxes with dotted outlines represent times after the fixed point has converged for a given batch. The boxes with solid outlines are those times at which a traditional dataflow system would have to do work: after convergence there is nothing to be done. The shading of the boxes indicates the amount of work that Naiad has to do for the corresponding time in this example, where a white background indicates no work at all, and lighter shading suggests less work. Understanding exactly what work is necessary under differential dataflow can seem quite counterintuitive; the box labeled $(4, 17)$ is shaded even though the fixed point converged at $(4, 16)$. Keeping enough information to reconstruct the full lattice of differences can occasionally lead to work one might not expect to perform, however in general this overhead is much lower than the work saved compared to an incremental implementation with totally-ordered time.

The interplay between differential dataflow and lattice time is illustrated in Figure 15, where again the lattice is two-dimensional with the horizontal dimension indicating

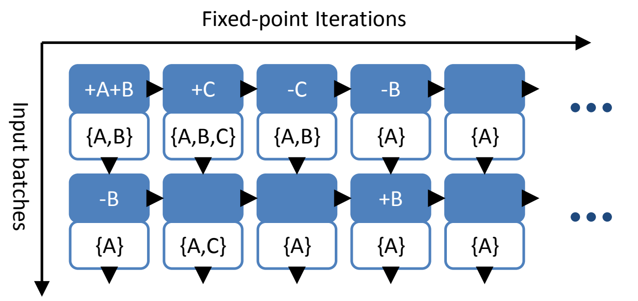


Figure 15. Differences and the corresponding induced collections for a dataflow edge with two-dimensional lattice time. The value of the induced collection at time t (white background) is computed by summing the differences (shaded background) at every time $t' \leq t$ where the lattice ordering is indicated using the black arrowheads.

iterations of a fixed point loop and the vertical dimension indicating batches of input. Now each shaded box shows the differences transmitted by differential dataflow along a particular dataflow edge in the program, and the unshaded boxes show the corresponding collection that a traditional dataflow system would transmit. The most important concept to understand in differential dataflow is that the collection at t is determined by summing the differences at *all* times $t' \leq t$ in the lattice, i.e. all shaded boxes above and to the left of the collection in the figure. The consequence can initially seem surprising, for example indicating that during the second fixed point loop shown, the transition from A to A between the third and fourth iterations (after fixed point has been reached!) requires a difference of $+B$ (intuitively, correcting for the behavior in the previous iteration, where a B should be subtracted). Nevertheless the reader can check correctness by manually summing the differences above and to the left of the corresponding box.

The fundamental intuition behind the performance gains of Naiad over traditional dataflow can be understood by considering the second time on the second row, where the shaded box is empty despite the fact that the collection has changed from the previous iteration in this batch (and indeed the second iteration at the previous batch). One way of understanding this is that, during the first batch, the system "discovered" that the difference between the first and second loop iterations is that a C is added; and thus adding a C the second time around requires no extra computation. Because the operator consuming this dataflow edge has been incrementalized, an empty box actually corresponds to the operator doing no work; the destination operator is already expecting the C to arrive. Of course, although the operator does no work at this timestep the system does have to do work to determine that the difference set is empty, but a careful implementation can minimize the overhead of these computations over differences.

Example	Description
A	Collection (a function from records to counts)
$A(x)$	Frequency of record x in collection A
A_k	Records in collection A mapping to key k
$f(A)$	Data-parallel operator f on collection A
$\mathbf{A}$	Collection trace
	(a function from lattice elements to collections)
$\mathbf{A}[t]$	Collection A at time $t \in L$
$\delta\mathbf{A}$	Difference trace, describes changes in $\mathbf{A}$.
	(a function from lattice elements to differences)
$\delta\mathbf{A}[t]$	Differences in A at time t
$\delta\mathbf{a}$	Difference trace updating difference trace $\delta\mathbf{A}$

Table 1. Legend of notation.

3.4 Using different lattices

An attractive property of generalizing time using a lattice is that Naiad can choose different lattices, for example, to improve the performance of an iterative computation. In practice, we have found the two most useful lattice constructions to be the product construction depicted in Figure 14, where $(a, b) \leq (c, d)$ iff both $a \leq c$ and $b \leq d$, and the lexicographic construction, where $(a, b) \leq (c, d)$ iff either $a < c$ or both $a = c$ and $b \leq d$. The product construction can result in substantial performance improvements in algorithms such as `SCC` in which the nested invocations of `ConnectedComponents` are made over successive versions of a graph that may differ by as little as a single edge. The lexicographic construction is beneficial for prioritization (as described the previous section), where successive fixed-point computations should reflect the final states of previous fixed-point computations, incorporating all differences rather than just those preceding the corresponding iterations. In each case, the ability to choose an appropriate lattice leads to sparse differences and relatively few recomputations.

4. Formalism

We now give a formal presentation of the differential dataflow model, which reassures us that Naiad has a sound theoretical basis on which we can reason about its behavior. A legend for the notation used in this section is included in Table 1.

A user defines computations over strongly typed *collections* of records. We model a collection using a function mapping records of some type R to integer counts, writing $A : R \to \mathbb{Z}$ and writing $A(x)$ for the frequency of record $x \in R$ in collection A. Collections can be added or subtracted by adding or subtracting their corresponding counts: $(A+B)(x) = A(x)+B(x)$ and $(A-B)(x) = A(x)-B(x)$.

A function from one or more collections to a collection is referred to as an *operator*. Operators on collections result in new collections which may be used in further computation, forming a dataflow graph. Although cycles may be introduced into a differential dataflow graph, to implement fixed-point iteration, individual operators do not introduce cycles.

Many operators express data-parallelism through *key functions* for each of their inputs. A key function maps an input record to a key type K that is common across all of the operator's inputs. The key space defines a notion of independence for an operator f, which can be written as

$$f(A, B) \quad = \quad \sum_{k \in K} f(A_k, B_k)\ . \qquad (1)$$

where a set A_k (or B_k) is defined in terms of its associated key function key as

$$A_k(r) \quad = \quad A(r) \text{ if } key(r) = k, \text{ 0 otherwise}, \qquad (2)$$

For example, a `Join` is parameterized by key functions for its two inputs, and only produces pairs of records whose keys match. A Naiad programmer specifies key functions on a per-operator basis, so for example each instance of `Join` may adopt a different key function. Rather than explicitly name the key functions of an operator f, we will just use A_k and B_k to reflect their role. In practice the independence in (1) allows the computation of $f(A, B)$ to be partitioned arbitrarily across threads, processes, and computers as long as elements mapping to the same key are kept together.

4.1 Lattice-Varying Collections

Central to differential dataflow is the ability of a collection to vary as a function of some lattice, T, intuitively describing progress through the computation. A *collection trace* is a function from lattice elements to collections, written using bold letters: $\mathbf{A} : T \to (R \to \mathbb{Z})$. For a lattice element $t \in T$, $\mathbf{A}[t]$ indicates the collection associated with that element.

The functional dependence of operators between input and output collections extends to collection traces: for an operator f over collections, we lift f to operate over collection traces by requiring the output collection trace to reflect at each t the operator applied to the input collections at t:

$$f(\mathbf{A}, \mathbf{B})[t] = f(\mathbf{A}[t], \mathbf{B}[t])\ . \qquad (3)$$

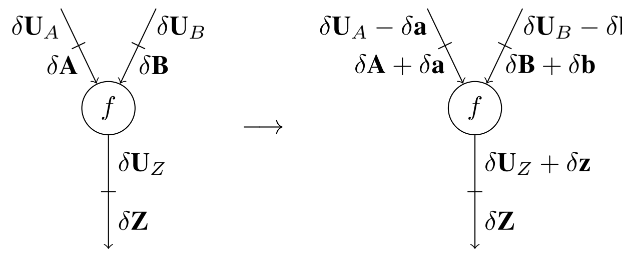
This lifting extends from individual operators to arbitrary dataflow over operators, and we will speak freely of dataflow defined over collections as applying to collection traces.

We now introduce an isomorphic representation for a collection trace, named a *difference trace*, a function from lattice elements to *differences*. A difference has the same type as a collection, $R \to \mathbb{Z}$, but instead reflects changes in the counts of the collection. For a collection trace $\mathbf{A}$ we write its corresponding difference trace as $\delta\mathbf{A}$, chosen so that

$$\mathbf{A}[t] = \sum_{s \le t} \delta\mathbf{A}[s]\ . \qquad (4)$$

This implicit definition of $\delta\mathbf{A}$ can be rearranged to define coordinates of $\delta\mathbf{A}$ explicitly in terms of $\mathbf{A}$ and prior differences

$$\delta\mathbf{A}[t] = \mathbf{A}[t] - \sum_{s < t} \delta\mathbf{A}[s]\ . \qquad (5)$$



$\delta\mathbf{U}_A$
$\delta\mathbf{U}_B$
$\delta\mathbf{A}$
$\delta\mathbf{B}$
f
$\delta\mathbf{U}_Z$
$\delta\mathbf{Z}$
$\longrightarrow$
$\delta\mathbf{U}_A - \delta\mathbf{a}$
$\delta\mathbf{U}_B - \delta\mathbf{b}$
$\delta\mathbf{A} + \delta\mathbf{a}$
$\delta\mathbf{B} + \delta\mathbf{b}$
f
$\delta\mathbf{U}_Z + \delta\mathbf{z}$
$\delta\mathbf{Z}$

Figure 16. A step in the Naiad differential dataflow model. Update difference traces ($\delta\mathbf{a}$ and $\delta\mathbf{b}$) are subtracted from the unprocessed traces ($\delta\mathbf{U}_A$ and $\delta\mathbf{U}_B$) and added to the processed traces ($\delta\mathbf{A}$ and $\delta\mathbf{B}$), while at the same time the output trace $\delta\mathbf{z} = \delta f(\mathbf{A} + \mathbf{a}, \mathbf{B} + \mathbf{b}) - \delta f(\mathbf{A}, \mathbf{B})$ is produced.

When viewed as an function δ is linear: $\delta(\mathbf{A}+\mathbf{B}) = \delta\mathbf{A}+\delta\mathbf{B}$. The definition of a difference trace $\delta\mathbf{A}$, and ultimately its efficacy, depends critically on the $\le$ relation of the lattice T, which does not need to be a total order.

We will often use *trace* for a collection trace or difference trace, writing the two $\mathbf{X}$ and $\delta\mathbf{X}$, with the understanding that we can go from one to the other and back again mathematically, but that it is the difference trace that is typically stored and acted upon by a practical implementation. Not only are difference traces often much more compact than collection traces, but they explicitly indicate when and how a collection has changed and allow us to restrict our recomputation appropriately. On the other hand, our computations are typically specified in terms of operators over collections, and in general we may need to be able to reproduce the collections as well.

4.2 Execution Model

The Naiad differential dataflow model is a directed and possibly cyclic dataflow graph where edges correspond to traces and vertices correspond to sources, sinks, or data-parallel operators. A source has no incoming edges, and a sink has no outgoing edges, and they encode the computation's inputs and outputs respectively. The intended final state of such a computation is an assignment of traces to the edges so that for each operator vertex, the trace on its output edge reflects the operator applied to the traces on its input edges, as in Equation (3).

The execution model is based on an assignment of two traces to each edge: the first has been *processed* by the recipient and is reflected in its output, and the second is *unprocessed* and calls for attention. Initially, all traces are empty. The system advances from one configuration to the next in one of two ways: either a source adds a trace to the unprocessed trace on its output edge; or an operator subtracts a trace from its unprocessed input trace (possibly, but not necessarily, leaving that region empty), adds the trace to its processed input trace, and adds a trace to the unprocessed trace of each output edge as defined by the logic of the operator. After each step it is the case that each output edge of an operator implementing a function f has its two output traces sum to f applied to the processed trace on its input edges. The computation quiesces when all unprocessed traces are empty, and so each operator's output traces equal f applied to its input traces. No more computation will ensue unless a source emits a new trace.

In Figure 16 we present the general case, where an operator f has previously processed traces $\delta\mathbf{A}$, $\delta\mathbf{B}$ resulting in $\delta\mathbf{Z}$, and now incorporates arbitrary traces $\delta\mathbf{a}$, $\delta\mathbf{b}$ as additions to $\delta\mathbf{A}$, $\delta\mathbf{B}$, resulting in output $\delta\mathbf{z}$ as the necessary coresponding addition to $\delta\mathbf{Z}$. Although $\delta\mathbf{a}$, $\delta\mathbf{b}$, $\delta\mathbf{z}$ are lower-case, they are still fully general traces and may reflect arbitrary sets of times t. Despite our use of difference traces (the δ notation), we may need to transform the traces to collection traces in order to evaluate operators f defined only over collections. This is mathematically simple using equations (4) and (5), and we explore the computational aspect in more detail in the coming subsection.

4.3 Updating Difference Traces

We now derive some details of the operator update rules, to serve as a basis for our implementation. Following the primitive step in our execution model, our goal is to determine, for a general data-parallel operator f, the necessary change $\delta\mathbf{z}$ to apply to its output traces to reflect the introduction of $\delta\mathbf{a}$ and $\delta\mathbf{b}$ to its input traces. We first determine the necessary correction to the output using collection traces, and then convert to difference traces to flesh out the implementation.

Let $\mathbf{A}$ and $\mathbf{B}$ be processed input collection traces, and let $\mathbf{a}$ and $\mathbf{b}$ be collection traces we intended to add to them. The necessary change to the output collection trace $\mathbf{z}$ is given by

$$\mathbf{z} \quad = \quad f(\mathbf{A}+\mathbf{a}, \mathbf{B}+\mathbf{b}) - f(\mathbf{A}, \mathbf{B}) \ . \qquad (6)$$

Reconstructing all these collections from difference traces seems daunting. However, in the general case where f is an arbitrary data-parallel operator we do need to determine the value of f on its new inputs. Fortunately, this reconstruction only needs to happen for those keys present in $\delta\mathbf{a}$ or $\delta\mathbf{b}$. Moreover, for several specific operators we can provide optimized implementations, discussed in Section 5.2.1. Following the data-parallel definition of f we only need to consider differences produced from keys k for which at least one of $\mathbf{a}_k$ or $\mathbf{b}_k$ are non-empty:

$$\mathbf{z} \quad = \quad \sum_k (f(\mathbf{A}_k+\mathbf{a}_k, \mathbf{B}_k+\mathbf{b}_k) - f(\mathbf{A}_k, \mathbf{B}_k)) \ . \quad (7)$$

Let $\mathbf{z}_k$ be the term in this sum corresponding to key k. From $\mathbf{a}_k$ and $\mathbf{b}_k$ we can determine which $\mathbf{z}_k$ may potentially be non-zero and restrict our attention to determining them.

Of course, our goal is to determine $\delta\mathbf{z}$, whose representation may be much more compact than that of $\mathbf{z}$. For specific t, individual elements $\delta\mathbf{z}_k[t]$ can be derived using the equality $\mathbf{z}_k[t] = \sum_{s \leq t} \delta\mathbf{z}_k[s]$, as

$$\begin{aligned}\delta\mathbf{z}_k[t] &= \mathbf{z}_k[t] - \sum_{s<t} \delta\mathbf{z}_k[s] & (8)\\ &= f(\mathbf{A}_k+\mathbf{a}_k, \mathbf{B}_k+\mathbf{b}_k)[t] - f(\mathbf{A}_k, \mathbf{B}_k)[t] - \sum_{s<t} \delta\mathbf{z}_k[s] \ . & (9)\end{aligned}$$

One can explicitly assemble $\mathbf{A}_k[t] + \mathbf{a}_k[t]$, $\mathbf{B}_k[t] + \mathbf{b}_k[t]$, $\mathbf{A}_k[t]$, and $\mathbf{B}_k[t]$, then apply f appropriately, followed by the subtraction of the $\delta\mathbf{z}_k[s]$.

We would like to avoid explicitly determining $\mathbf{z}_k[t]$ for all values of t, and instead leap directly to the non-zero entries of $\delta\mathbf{z}_k$. We can conservatively approximate the set $\{t : \delta\mathbf{z}_k[t] \neq 0\}$ using the sets

$$T_1 = \{t : \delta\mathbf{a}_k[t] \neq 0 \text{ or } \delta\mathbf{b}_k[t] \neq 0\}$$
$$T_2 = \{t : \delta\mathbf{A}_k[t] \neq 0 \text{ or } \delta\mathbf{B}_k[t] \neq 0\}$$

For $\delta\mathbf{z}_k[t]$ to be non-zero, t must be the least upper bound of a subset of $T_1 \cup T_2$ containing at least one element of T_1. If t is not such a least upper bound, then the differences at times less than it are reflected at their least upper bound, from which there is no further difference to report at t. If t is not preceded by at least one non-zero difference from $\delta\mathbf{a}_k$ or $\delta\mathbf{b}_k$, the input remains unchanged at t and no output change is required. A formal proof can be made by induction over the partial order. Note that, since this approximation is conservative, it does not affect the correctness of the computation. When the approximation is not exact it simply means that some unnecessary computation ensues, as work is done for values of t that correspond to empty entries of $\delta\mathbf{z}_k[t]$.

5. Implementation

We have implemented a prototype version of Naiad that uses multi-core parallelism in a single shared-memory computer. In this section, we discuss four aspects of the implementation that are independent of the mechanism used to provide parallel execution: the translation of a Naiad query into a cyclic dataflow graph (Subsection 5.1), the efficient execution of fixed-point queries (Subsection 5.1.1), the data-dependent prioritization of execution (Subsection 5.1.2) and the incrementalization of standard data-parallel operators. The full set of Naiad operators is presented in Appendix A.

5.1 From query to differential dataflow graph

A Naiad query is written in terms of updatable input *sources* and data-parallel *operators* that transform these sources into new collection traces. As in LINQ, the sources and operators are strongly typed, and many of the operators are higher-order functions that allow the user to call arbitrary user-defined code on the elements of the trace. Many of the operators Naiad supports are standard LINQ operators—such as `Select`, `Where`, `GroupBy`, and `Join`—which are analogous to their similarly named counterparts in SQL.

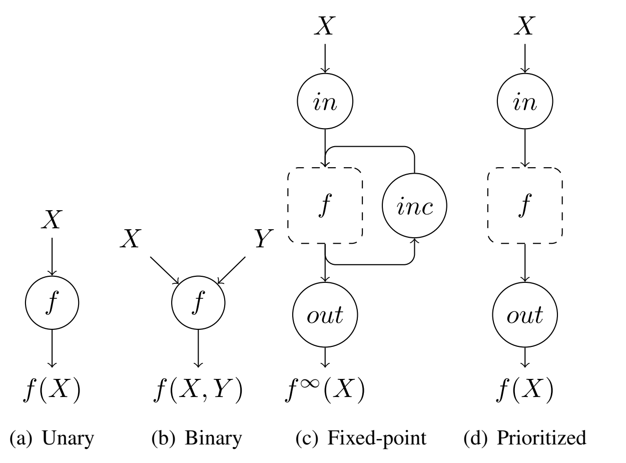


Figure 17. The four primitive dataflow graphs for the Naiad operators. A dashed box represents a subquery.

For Naiad to be able to execute a query, we must transform it into a differential dataflow graph. Previous work has shown how to transform declarative languages like LINQ and Pig Latin into a directed acyclic dataflow graph, whose vertices can be scheduled in topological order. In Naiad a new dataflow vertex for an operator is constructed from the one or two vertices whose outputs it consumes. The two cases of unary and binary operators are presented in Figures 17(a) and 17(b). For these operators, the inputs must be defined before the operator is constructed, and so we can simply create a new vertex representing the operator and connect the appropriate inputs.

To support data-parallel execution, all operators are divided into one or more *shards*, each of which is responsible for processing one part of a partition of its input differences. Some operators—such as `Select`, `Where` and `Concat`—process each record independently of all others, and the partitioning is inherited from the upstream operator in the dataflow. All other operators are equipped with a key function for each input, and the partitioning is performed according to the operator's key function. Each shard of an operator can then be executed independently in parallel; in the current implementation, each shard of an operator is mapped to a different processor core. Some operators—such as `Join` and `Min` —require that all records with the same key be processed by the same shard. For these operators Naiad hashes the key of each record to assign it to a shard, and then performs a data-exchange to route the differences to consistent shards.

5.1.1 Fixed Point

Naiad's `FixedPoint` operator is not instantiated by a single vertex, but rather by the subgraph shown in Figure 17(c). The `FixedPoint` operator takes as a parameter a function `f` from some collection trace to a collection trace of the same type. The function `f` can be an arbitrary Naiad query, and may itself contain the `FixedPoint` operator. We first generate the dataflow graph for `f`, and then insert it in a cyclic harness that computes the fixed point of `f` (where it exists). In this section, we explain how that harness uses differential dataflow to compute that fixed point efficiently.

The first vertex in the cyclic harness is an ingress operator that extends the lattice elements associated with incoming records with a new integer coordinate that will hold the iteration counter. For each difference $\delta\mathbf{a}$ received as input, the vertex outputs a difference $\delta\mathbf{z}$ satisfying

$$\delta\mathbf{z}[(t,0)] = \delta\mathbf{a}[t] \quad \text{and} \quad \delta\mathbf{z}[(t,1)] = -\delta\mathbf{a}[t]\ .$$

This corresponds to adding the input collection in iteration 0, and subtracting the input collection in iteration 1. As we will now explain, these initial conditions are appropriate to compute the fixed-point of the given function.

The supplied function `f` is applied to the ingress vertex, generating a differential dataflow subgraph (which may itself contain more nested cycles). We then attach an *incrementer* vertex that takes an input difference $\delta\mathbf{a}$ to an output difference $\delta\mathbf{z}$ as

$$\delta\mathbf{z}[(t,i+1)] = \delta\mathbf{a}[(t,i)]\ .$$

The output of the incrementer vertex is then returned to the input of `f` and concatenated with the input from the ingress.

If the input to `FixedPoint` is some trace $\mathbf{A}$, then `f` sees $\mathbf{A}$ at iteration 0, as the only input at that time comes from the ingress. At iteration 1, `f` is presented with both $\mathtt{f}(\mathbf{A})$ from the feedback edge and $-\mathbf{A}$ from the ingress. This update logically changes the input from $\mathbf{A}$ to $\mathtt{f}(\mathbf{A})$ and prompts the subgraph corresponding to `f` to produce $\mathtt{f}(\mathtt{f}(\mathbf{A})) - \mathtt{f}(\mathbf{A})$ as output (the necessary correction to its prior output $\mathtt{f}(\mathbf{A})$). This difference has its iteration index incremented and continues the cycle. Generally, the combined ingress and backedge collection $\mathbf{X}$ satisfies

$$\delta\mathbf{X}[(t,i)] = \mathtt{f}^{i}(\mathbf{A}[t]) - \mathtt{f}^{i-1}(\mathbf{A}[t])$$

The loop ceases to propagate updates on the back edge only once $\delta\mathbf{X}[(t,i)] = 0$, and so fixed point has been achieved.

Finally, we attach an egress vertex which strips off the loop index and accumulates all differences, setting

$$\delta\mathbf{z}[t] = \sum_{i} \delta\mathbf{a}[(t,i)]\ .$$

This vertex reports the limit of the iteration, the fixed point, and we return it as the output of `FixedPoint(f)`.

The ingress, incrementer and egress vertices are all implemented as stateless operators that rewrite the lattice times on records. They introduce negligible runtime overhead and do not require a data exchange.

5.1.2 Prioritization

The `Prioritize(priority, f)` operator is also parameterized by a subgraph (Figure 17(d)), but its implementation is much simpler than FixedPoint. The graph also has an ingress vertex, which in this case introduces the lattice element selected by the priority function, followed by the f subgraph, followed by an egress vertex that strips off the lattice element that was introduced in the ingress. Importantly, the introduced priority results in a new lattice not through the product construction, but through the lexicographic construction. In the context of the subcomputation f, differences at earlier priorities are reflected by all later priorities, independent of the relation of subsequently added coordinates (as might be added by a nested fixed point computation).

5.2 Incremental operators

Conceptually, a non-incremental operator can be made incremental by maintaining enough state to reconstruct its input collections at the previous times, adding differences to those collections, and re-executing the operator on the new collections. However, such an approach would perform work proportional to the size of the collections for each update, and would not provide the performance properties that we desire for Naiad. In this subsection, we describe the implementation of a generic Naiad incremental vertex that improves on this baseline, and then how this implementation can be specialized for many different LINQ operators.

As a Naiad program executes, our generic operator is invoked repeatedly with difference traces $\delta\mathbf{a}$ and $\delta\mathbf{b}$ to incorporate into its inputs, and must produce an output difference trace $\delta\mathbf{z}$ that reflects their addition. Equation (9) indicates that our output $\delta\mathbf{z} = \sum_k \delta\mathbf{z}_k$ should satisfy

$$\delta\mathbf{z}_k[t] = f(\mathbf{A}_k + \mathbf{a}_k, \mathbf{B}_k + \mathbf{b}_k)[t] - f(\mathbf{A}_k, \mathbf{B}_k)[t] - \sum_{s<t} \delta\mathbf{z}_k[s] \ .$$

In order to efficiently compute $\delta\mathbf{z}$ for arbitrary inputs, our generic operator will store its full input difference traces $\delta\mathbf{A}$ and $\delta\mathbf{B}$ indexed in memory. The present operator implementation stores this trace in triply-nested sparse array of counts, indexed first by key k, then by lattice time t, then by record r. Naiad maintains only non-zero counts, and as records are added to or subtracted from the difference trace Naiad dynamically adjusts the allocated memory.

With $\delta\mathbf{A}$ and $\delta\mathbf{B}$ indexed by key, we can reconstruct $\mathbf{A}_k$ and $\mathbf{B}_k$ and compute $\delta\mathbf{z}_k$ explicitly using the pseudocode of Figure 18, avoiding the reconstruction for keys whose records have not changed. While reconstruction may seem expensive, and counter to incremental computation, it is necessary to be able to support operators such as GroupBy for which the programmer may specify an arbitrary (non-incremental) function to processes all records associated with a particular key. We will soon see that many specific operators have more efficient implementations.

One general optimization to the algorithm in Figure 18 reduces the number of lattice elements that are considered in reconstructing the trace. As discussed in Section 4.3, we only need to evaluate $\delta\mathbf{z}_k$ at lattice elements t that are the

```
$\delta\mathbf{z} \leftarrow 0$
for all keys $k \in \delta a$ or $\delta b$ do
  $dz_k \leftarrow 0$
  for all elements $t \in$ lattice do
    $A_k \leftarrow a_k \leftarrow 0$
    $B_k \leftarrow b_k \leftarrow 0$
    for all elements $s \in$ lattice do
      if $s \leq t$ then
        $A_k \leftarrow A_k + \delta\mathbf{A}_k[s]$
        $B_k \leftarrow B_k + \delta\mathbf{B}_k[s]$
        $a_k \leftarrow a_k + \delta\mathbf{a}_k[s]$
        $b_k \leftarrow b_k + \delta\mathbf{b}_k[s]$
        $dz_k[t] \leftarrow dz_k[t] - dz_k[s]$
      end if
    end for
    $dz_k[t] \leftarrow dz_k[t] + f(A_k + a_k, B_k + b_k) - f(A_k, B_k)$
  end for
  $\delta\mathbf{z} \leftarrow \delta\mathbf{z} + dz_k$
end for
return $\delta\mathbf{z}$
```

Figure 18. Pseudocode for subvertex update logic.

least upper bound of a subset of non-zero times from $\delta\mathbf{A}_k$, $\delta\mathbf{B}_k$, $\delta\mathbf{a}_k$, or $\delta\mathbf{b}_k$, containing at least one non-zero time from $\delta\mathbf{a}_k$ or $\delta\mathbf{b}_k$. This substantially reduces the amount of work we need to perform.

Additionally, rather than reconstruct each $\mathbf{A}_k[t]$ from scratch, we can simply update whatever previous $\mathbf{A}_k[s]$ we reconstruct to $\mathbf{A}_k[t]$. Doing so only involves differences

$$\{\delta\mathbf{A}_k[r] : (r \leq s) \neq (r \leq t)\} \ .$$

This often results in relatively few r in difference, often just one in the case of advancing loop indices. Ensuring that we process differences in a sequence that respects the partial order, we need only scan from the greatest lower bound of s and t until we pass both s and t.

5.2.1 Special Implementations

Although this generic vertex algorithm can be used to implement any Naiad operator, we have specialized the implementation of the following operators to achieve better performance:

Pipelined Operators Several operators—including Select, Where, Concat and Except—are *linear*, which means they can determine $\delta\mathbf{z}_k$ as a function of only $\delta\mathbf{a}_k$, with no dependence on $\delta\mathbf{A}_k$. These operators do not need to maintain any state, and apply record-by-record logic to the non-zero elements of $\delta\mathbf{a}_k$—respectively transforming, filtering, repeating and negating the input records.

Join The Join operator combines two input collections by computing the cartesian product of those collections, and yielding only those records where both input records have the same key. Due to the distributive property of Join, the relationship between inputs and outputs is straightforward:

$$\mathbf{z}_k = \mathbf{A}_k \bowtie \mathbf{b}_k + \mathbf{a}_k \bowtie \mathbf{B}_k + \mathbf{a}_k \bowtie \mathbf{b}_k \ .$$

While the implementation of Join does need to keep its input difference trace resident, its implementation is much simpler than the general case. An input $\delta\mathbf{a}_k$ can be handled by making a sequential pass through the non-zero elements of $\delta\mathbf{B}_k$, and analogously for $\delta\mathbf{b}_k$ and $\delta\mathbf{A}_k$.

Aggregations Many of the data-parallel aggregations have very simple update rules that do not require all records to be re-evaluated. Count, for example, only needs to retain the difference trace of the number of records for each key, defined by the cumulative weight, rather than the set of records mapping to that key which can be discarded. Sum has a similar optimization. Min and Max must keep their full input difference traces—because the retraction of the minimal (maximal) element leads to the second-least (greatest) record becoming the new output—but can often quickly establish that an update requires no output by comparing the update to the prior output, without reconstructing $\mathbf{A}_k$.

6. Experimental Evaluation

In this section we evaluate the performance and scalability of Naiad using several algorithms. Our goal is to assess the hypothesis that differential dataflow can lead to efficient implementations of incremental and iterative computation. At this stage of Naiad's development, absolute performance is not our primary concern and so we have not yet expended much effort on performance-related improvements. Nevertheless, we feel that the results presented in this section show that our approach is viable.

Throughout this section we present results from a prototype implementation that is written entirely in C#, using version 4.0 of the .NET Framework.

6.1 Algorithms / Computations

We have selected four representative computations capable of taking advantage of our incremental computations, but poorly served by many existing data-parallel systems:

- **Single-Source Shortest Paths** The program is based on the Bellman-Ford algorithm in which each node repeatedly broadcasts its distance from the source to all of its neighbors, each of which accumulates their incoming messages and selects the nearest. Figure 19 shows the program using the FixedPoint operation to repeatedly apply the update rule. We evaluate two versions of this algorithm, one with prioritization (approximating Δ-stepping) and one without.
- **Connected Components** We use the algorithm from Section 2 (see Figure 3). We also consider the prioritized version shown in Figure 10.

```
struct Node { int id; int pred; int dist }
struct Edge { int src; int dest; int wgt }

// Initially, nodes contains the single source.
// The result contains all nodes, their distances
// from the source, and the node leading to them.
Collection<Node> SSSP(Collection<Node> nodes,
                      Collection<Edge> edges)
{
  return nodes.FixedPoint(xs => Broadcast(xs, edges));
}

// Extends nodes to include any neighbors of nodes.
Collection<Node> Broadcast(Collection<Node> nodes,
                           Collection<Edge> edges)
{
  return nodes.Join(edges,
                    node => node.id,
                    edge => edge.src,
                    (node, edge) =>
                      new Node(id:   edge.dst,
                               pred: edge.src,
                               dist: node.dist + edge.wgt))
              .Concat(nodes)
              .Min(node => node.id, node => node.dist);
}
```

Figure 19. Naiad program for single-source shortest paths.

- **Strongly Connected Components** This algorithm is also described in Section 2, with code shown in Figure 8.
- **Smith-Waterman Sequence Alignment** Smith-Waterman is a dynamic programming algorithm for aligning two sequences. For each pair (i, j) of indices into the two strings, it computes the optimal alignment score, using the scores for $(i, j-1)$, $(i-1, j)$, and $(i-1, j-1)$. From an initial set of scores at $(i, 0)$ and $(0, j)$, the dynamic program is a fixed point computation repeatedly augmenting the set of known scores.

6.2 Absolute performance and memory footprint

We begin by examining the impact, without any parallelism, of the Naiad system in terms of computation and memory footprint. For these experiments we use an Intel Core i7 at 2.67 GHz. Table 2 reports the running time and working set size for single-threaded LINQ and single-threaded Naiad implementations of Single Source Shortest Paths (SSSP), Strongly Connected Components (SCC) and Connected Components (CC). Inputs to SSSP and SCC are randomly generated graphs with 1M nodes. For CC we use the Amazon graph from [19], which contains around 400,000 nodes and 3.4 million edges.

Naiad's FixedPoint and Prioritize operations result in much faster running times than the 'out-of-the-box' LINQ version. More interestingly, the incremental nature of Naiad ensures that it is orders of magnitude faster than LINQ when processing single element updates. The penalty paid is in the memory footprint, which can be significant (up to a factor of 9 in Table 2).

We stress that LINQ is not intended for high-performance computing, and our choice of it as a comparison is mostly to evaluate the same algorithm in the same language with sim-

Prog	Edges	Running time		Footprint (MB)		
		LINQ	Naiad	LINQ	Naiad	Updates
SSSP	$1M$	11.88s	4.71s	386	309	0.25ms
SSSP	$10M$	200.23s	57.73s	1,259	2,694	0.15ms
SCC	$200K$	30.56s	4.36s	99	480	1.12ms
SCC	$2M$	594.44s	51.84s	514	3,427	8.79ms
CC	$3.4M$	66.81s	9.90s	1,124	985	0.49ms

Table 2. Impact of Naiad for single-threaded executions.

ilar runtime overheads. Bespoke solutions for SSSP, SCC, and CC implemented directly in C# have about an order of magnitude lower elapsed time than Naiad, but crucially they are still up to three orders of magnitude slower than Naiad's incremental recomputation.

6.3 Multi-processor evaluation

Moving to a parallel execution environment, we are interested in Naiad's scaling properties. We ran the experiments on an AMD Opteron 'Magny-Cours' with 48 (4×12-core) 1.9GHz processors and 64GB of RAM. Unless otherwise stated, the input data are randomly generated with size chosen to ensure that all computations fit in main memory, and we repeat every run 5 times and report the mean.

6.3.1 Scalability

As the number of concurrent compute threads increases while holding the input collection fixed, we hope to see performance improve proportionally. The typical obstacle to such "strong scaling" is that as the computation gets faster, the relative time spent in synchronization and other overheads increase. Figure 20 shows the scaling of Smith-Waterman, SSSP, SCC and CC by plotting the ratio of single-thread running time to x-thread running time for values of x from 1 to 48 (the maximum core count on our server). The lowest curve plots Smith-Waterman on a string length of just 20,000, whose input size is "too small" to fully engage Naiad's scalability; the choice of block size (1,000) results in a theoretical maximum scaling of 10x which we almost achieve.

6.3.2 Incremental updates

The next experiment examines the performance associated with incremental changes to inputs. Each computation involves at least one loop, and we might expect that a single input alteration could have far reaching effects. In this experiment we introduce multiple epochs of input to the Naiad computation, and the main variable we have to investigate is the size of the batch we process in each epoch; larger batches should result in higher throughput, due to additional parallelism, but at the cost of higher per-epoch latency. We ran SSSP on a one-million node graph using thread counts from 1 to 48, and then submitted updates to the graph in batches of 1, 10, 100, 1000 and 10,000 records. We found that through-

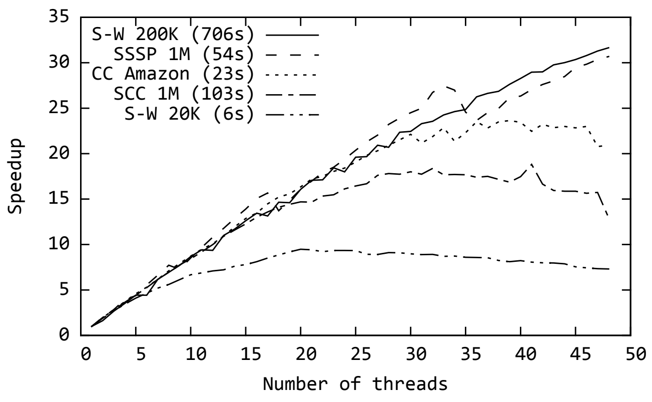


Figure 20. Speed-up of multi-threaded computation over single-threaded computation, varying the number of threads. The single-threaded running time is noted in the key.

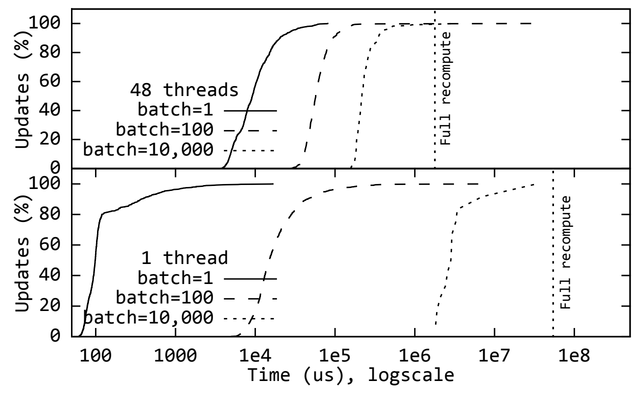


Figure 21. Cumulative density functions for incremental update latency in SSSP, varying the increment size (number of records changed) from 1, to 100, to 10,000. Different distributions result for single-threaded and 48-threaded.

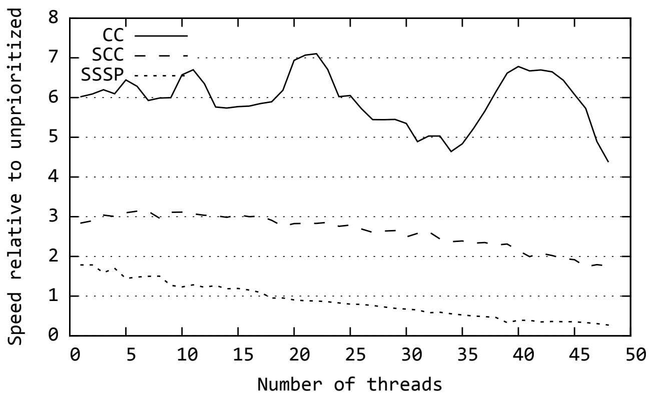


Figure 22. Effect of prioritization on scalability for various computations.

put increased in proportion to the batch size, except for the single-threaded case when it declined slightly.

We consider latency in Figure 21. There we see the cumulative density plots as we vary thread count and batch size, revealing that the coordination of multiple threads costs when there is not much work (batch size 1) but helps when there is much to do (batch size 10,000). The plot has vertical lines showing the times for the full re-computations, and we see that it is almost always much faster to update the inputs rather than re-run the computation from scratch.

6.4 Prioritization

The three algorithms admit prioritized implementations, where SSSP propagates smallest distances first, and CC/SCC propagate low identifiers first. Doing so results in less computation, but potentially less parallelization due to ordering of the increments. Figure 22 shows the speed increase/decrease of CC, SCC and a version of SSSP where we increased the diversity of the edge weights 100-fold over previous experiments. Prioritization improves the single-threaded runtime in each case and still exhibits positive scaling. We deliberately performed the SSSP experiment to highlight a pitfall of prioritization—making priorities too fine-grain reduces parallelism so much that scaling suffers. In this case the absolute performance of the non-prioritized version is actually better than the prioritized once more than about 18 cores are used. A coarser prioritization recovers the performance and scalability for SSSP. The task of automating the selection of a prioritization function is left to future research.

7. Related work

In recent years the use of dataflow abstractions has become a popular way to simplify the implementation of parallel programs. To write a dataflow program the programmer defines a set of purely functional sub-programs and the input/output dependencies between those sub-programs, which form a directed graph [9]. In this section, we survey some of the dataflow systems that have inspired Naiad.

7.1 Parallel dataflow systems

If a program can be expressed in a functional style, with at least some functions applied independently across a large data set, it is amenable to automatic parallelization. The most popular recent example of this approach is Dean and Ghemawat's MapReduce system [10], which allows programs to be expressed using a pair of higher-order functions, `map` and `reduce`. A runtime system can then execute these functions in parallel on a shared-nothing cluster [10] or a shared-memory multiprocessor [23].

Higher-level programming models can be built on top of the MapReduce model. For example, Pig Latin [22] implements several relational-style operators on top of MapReduce, while FlumeJava [8] transforms programs written as a composition of user-defined functions into data-parallel pipelines of MapReduce jobs.

A MapReduce job has a fixed, two-stage dataflow which transfers the outputs of the parallel `map` invocations to the inputs of the parallel `reduce` invocations. The Dryad system [17] generalizes MapReduce by executing programs that form an arbitrary directed-acyclic dataflow graph. Dryad also admits several higher-level programming models, including DryadLINQ [26], which transforms programs written as LINQ queries [4] (resembling SQL queries embedded in a C# program) into Dryad graphs. DryadLINQ also encourages a functional style, by allowing programmers to specify the behavior of each query operator as a C# lambda expression (which may invoke arbitrary C# code).

While these systems and programming models have been successful at extracting large-scale parallelism, their expressive power is limited. In particular, none of the above programming models supports a native (data-dependent) iteration construct, which forces programmers to write iterative programs as multiple independent MapReduce or Dryad jobs. Furthermore, since these jobs are independent, neither system has sufficient information to perform optimizations across iterations.

7.2 Iterative dataflow

To extend the generality of dataflow systems, several researchers have investigated ways of adding data-dependent control flow constructs to parallel dataflow systems.

The Twister [13], HaLoop [6] and iMapReduce [29] systems add support for unbounded iteration to MapReduce. Twister allows the programmer to specify a boolean function on the merged results of a single MapReduce job, which can be used as a convergence test. HaLoop and iMapReduce are more declarative: both systems allow the programmer to define a distance metric between the results of two consecutive jobs. All three systems provide optimizations for iterative computation, such as storing invariant input data in memory between iterations. The execution model of these systems is based on repeatedly executing a single MapReduce job (or chain of MapReduce jobs). This prevents these systems from executing more complicated dataflow graphs, such as Pig, FlumeJava or DryadLINQ programs that form an arbirary DAG.

More general systems have been developed for iterative dataflow. Spark [27] supports a programming model that is similar to DryadLINQ, with the addition of explicit in-memory caching for frequently-reused inputs. Spark also provides a "resilient distributed dataset" abstraction that allows cached inputs to be reconstructed in the event of failure. The CIEL distributed execution engine [21] supports the reliable execution of arbitary Turing-complete programs that are transformed into "dynamic task graphs"; however because CIEL does not constrain the programming or data model, it is less amenable to automatic optimization than the other systems.

The foregoing systems are all based on an acyclic dataflow model, in which each vertex executes only once, and iteration is supported by extending the dataflow graph. An alternative dataflow model allows *cycles* in the dataflow graph, and repeated execution of the vertex code [9]. Cyclic dataflow is used extensively in stream processing languages, such as StreamIt [24] and SPADE/SPL [14], which use cycles to represent feedback loops. StreamIt adopts a synchronous dataflow model [18], in which each vertex consumes and produces a static number of *tokens* on its incoming and outgoing edges. This property allows StreamIt programs to be scheduled and aggressively optimized at compile time, but it inhibits the implementation of programs that produce variable or data-dependent numbers of outputs (such as database queries or MapReduce programs). SPADE (now known as SPL) supports relational-style operators on incoming streams of data and static collections. In SPADE, feedback edges can be used to adapt the behavior of upstream operators, but they may not be used to produce additional tuples (out of fear of non-termination).

The principal advantage of cyclic dataflow over acyclic (but dynamic) dataflow is that vertices may retain mutable local state between invocations. Most Naiad operators take advantage of such state to offer significantly lower latency than acyclic dataflow systems. Unlike existing cyclic dataflow systems, Naiad supports non-iterative MapReduce- or Dryad-style programs with unconstrained inputs and outputs, as well as Turing-complete programs with data-dependent iteration (and possible non-termination).

7.3 Incremental computation

MapReduce and Dryad have origins in batch processing, which means that they retain no state and reprocess the entire data set when the inputs change. When the change is relatively small—as it may be in an iterative computation—it would be more attractive to use an *incremental* model of computation.

The simplest form of incremental computation is *memoization*. Of the above-mentioned systems, Spark supports the explicit memoization of frequently-used queries, CIEL uses a deterministic naming scheme and lazy evaluation to memoize the results of identical tasks, and DryadLINQ can use Nectar [15] to identify common subexpressions across multiple queries. Memoization only works when the inputs are identical, so Nectar additionally supports incremental computation when records are appended to the input, by caching the intermediate results and combining them with partial results from the appended records.

A more general approach to incremental computation is *self-adjusting computation*, which Acar proposed in his thesis [1]. In a self-adjusting computation, the data dependencies are tracked explicitly, and changes to the inputs are handled by a fine-grained change propagation algorithm. Until recently, self-adjusting computation was applied only to sequential computations. Bhatotia *et al.* developed Incoop [3], which is an incremental version of MapReduce based on Acar's model. Incoop executes unmodified MapReduce computations, and incrementally executes programs that use associative aggregation functions. However, it does not support iterative MapReduce jobs.

In contrast to general self-adjusting computation, Naiad programs are composed of data-parallel operators that have an incremental implementation. While this is a more restrictive model, it includes any composition of MapReduce jobs, as well as more general data-parallel functionality such as our `FixedPoint` operator.

8. Conclusions

We have introduced a new computational framework, differential dataflow with partially ordered logical time, and shown how it can be used as the basis for a data-parallel system that is fundamentally incrementalized and also allows fully composable nested iteration and prioritization. We have shown that a careful implementation can minimize the overhead of maintaining and reasoning over partially-ordered execution traces, leading to a system that supports incrementalized computation of such properties as strongly connected components over real-world graphs, where recomputing components after graph updates is several orders of magnitude faster than re-running the full analysis.

The implementation described in this paper is a proof of concept of the viability of differential dataflow, but is simplified in some ways and restricted to run on a single shared-memory computer. Nevertheless, the framework is intended to be suitable for scalable computations over a computing cluster. The coordination between vertices is already explicit, through messages, and can straighforwardly be distributed over a network. However, numerous issues emerge from an attempt to scale up the system, most significantly in scheduling. Presently, the compute threads proceed through the computation in a coordinated fashion, exploiting data-parallelism but largely ignoring task and pipeline parallelism. Fortunately, our incremental update rules allow any computation to be safely processed in any order; the coordination is present only to ensure timely convergence. We are currently exploring these issues in a prototype cluster implementation. Our current implementation also suffers from a memory footprint that strictly grows as computation progresses, however in many cases state can be consolidated once all effects of an input batch are reflected in the outputs.

We have begun to explore uses of more exotic lattices within differential dataflow, and have several promising leads. While some are relatively direct (e.g. letting collections vary as a function of a security lattice), one appears to lift the level of abstraction again, from fixed point iteration to recursion. Specifically, nested data-parallelism [5] arises from a lattice whose elements are sequences of elements from some base lattice, where $s \leq t$ iff $s_i \leq t_i$ for each component i. Differences are passed into and returned from recursive calls by adding to and removing from the end of the lattice sequence, but all flow through the same (cyclic) dataflow graph.

The problems of adding both incrementality and iteration to scalable dataflow computations have been extensively investigated in recent years. We believe that the differential dataflow framework is the first general solution to these problems, elegantly combining both mechanisms within the same system.

References

[1] U. A. Acar. *Self-adjusting computation*. PhD thesis, Carnegie Mellon University, 2005.

[2] R. S. Barga, J. Goldstein, M. H. Ali, and M. Hong. Consistent streaming through time: A vision for event stream processing. In *3rd CIDR*, Jan. 2007.

[3] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquini. Incoop: MapReduce for incremental computations. In *ACM SOCC*, Oct. 2011.

[4] G. M. Bierman, E. Meijer, and M. Torgersen. Lost in translation: Formalizing proposed extensions to C$^\sharp$. In *22nd OOPSLA*, Oct. 2007.

[5] G. E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39:85–97, Mar. 1996.

[6] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient iterative data processing on large clusters. In *36th VLDB*, Sept. 2010.

[7] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1:146–166, March 1989.

[8] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. In *ACM PLDI*, June 2010.

[9] A. L. Davis and R. M. Keller. Data flow program graphs. *IEEE Computer*, 15(2):26–41, Feb. 1982.

[10] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *6th USENIX OSDI*, 2004.

[11] D. J. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Communications of the ACM*, 35(6):85–98, 1992.

[12] A. Eisenberg and J. Melton. SQL: 1999, formerly known as SQL 3. *SIGMOD Record*, 28(1):131–138, 1999.

[13] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative MapReduce. In *19th ACM HPDC*, June 2010.

[14] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. SPADE: The System S declarative stream processing engine. In *2008 ACM SigMod*, June 2008.

[15] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: automatic management of data and computation in datacenters. In *9th USENIX OSDI*, Oct. 2010.

[16] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *Data Engineering Bulletin*, 18(2), June 1995. Special Issue on Materialized Views and Data Warehousing.

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

[18] E. A. Lee and D. G. Messerschmitt. Sychronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[19] J. Leskovec, L. A. Adamic, and B. A. Huberman. The dynamics of viral marketing. *ACM Transactions on the Web*, 1 (1), May 2007. Amazon product co-purchasing network, June 1 2003, from http://snap.stanford.edu.

[20] 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 *2010 ACM SigMod*, June 2010.

[21] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. CIEL: a universal execution engine for distributed data-flow computing. In *8th USENIX NSDI*, Mar. 2011.

[22] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-foreign language for data processing. In *2008 ACM SigMod*, 2008.

[23] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *13th HPCA*, 2007.

[24] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *2002 ICCC*, Apr. 2002.

[25] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive—a petabyte scale data warehouse using Hadoop. In *26th IEEE International Conference on Data Engineering*, Mar. 2010.

[26] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *8th USENIX OSDI*, Dec. 2008.

[27] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, and M. Franklin. Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing. Technical Report UCB/EECS-2011-82, Electrical Engineering and Computer Sciences, University of California at Berkeley, July 2011.

[28] 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 *9th USENIX NSDI*, Apr. 2012.

[29] Y. Zhang, Q. Gao, L. Gao, and C. Wang. iMapReduce: A distributed computing framework for iterative computation. In *1st International Workshop on Data Intensive Computing in the Clouds*, May 2011.

[30] Y. Zhang, Q. Gao, L. Gao, and C. Wang. PrIter: A distributed framework for prioritized iterative computations. In *ACM SOCC*, Oct. 2011.

Type	Operator	Stateful?	Deviation from LINQ?
Unary	Select	No	
	SelectMany	No	
	Where	No	
	Aggregate	Yes	Supports subtraction
	Count	Yes	Returns collection
	Sum	Yes	Returns collection
	Min	Yes	Returns collection
	Max	Yes	Returns collection
	Distinct	Yes	Returns collection
	GroupBy	Yes	
Binary	Join	Yes	
	Union	Yes	Multiset semantics
	Intersect	Yes	Multiset semantics
	Except	No	Multiset semantics
	Concat	No	

Table 3. LINQ operators implemented in Naiad.

A. Naiad language definition

This Appendix specifies all the operators provided by Naiad and outlines differences with standard LINQ operators where applicable. The base type of collections in Naiad is `Collection<R,T>` which corresponds to a multiset of records of type `R` parameterized by a lattice of type `T`. By including the lattice as part of the collection and operator types we give the C# compiler the required information to detect invalid uses of collections with incompatible lattices.

A.1 Naiad versions of LINQ operators

Table 3 lists the LINQ operators that have corresponding implementations in Naiad, indicates whether they are stateful (for example, as described for the `Distinct` operator in Section 3.1), and notes where the signature and semantics deviate from the LINQ version. Most LINQ methods return either sequences or singleton values, whereas in Naiad operators always return a collection. For example, the LINQ version of `Count` takes an entire collection and returns an integer, while the Naiad version additionally takes a key function, and returns a collection containing a count corresponding to each unique key mapped from an element in the input. We similarly define data-parallel versions of other common aggregation operators, (e.g. `Sum`, `Min`, etc.), based on their LINQ counterparts with an additional key function. This allows these aggregates to participate in further dataflow, and support optimized implementations as cases of `GroupBy`.

We give the syntax of `Select`, a representative unary operator that produces the collection that results from mapping a given function across each element in an input collection:

```
Collection<S,T>
  Select<R,S,T>(Collection<R,T> input,
                Func<R,S> selector)
```

The type parameters `R` and `S` correspond to the record types in the input and output collections, respectively, and the `selector` is a function from `R` to `S`.

`Select` and the other operators modeled on standard LINQ version are additionally parameterized by a single lattice type `T`. Informally, this means that the operator matches, and does not change, the lattice associated with records in the input collection.

The most general unary operator is `GroupBy`, which takes a collection, a key function, and a reduction function from groups to output lists. It collates the input by key, applies the reduction function to each group, and accumulates the result:

```
Collection<S,T>
  GroupBy<R,K,S,T>(Collection<R,T> input,
                   Func<R,K> key,
                   Func<K,
                        IEnumerable<R>,
                        IEnumerable<S>> reducer)
```

The `reducer` is a function from the key and a group of input records to an `IEnumerable<R>` of output records. `IEnumerable<R>` is the generic interface in .NET representing a collection of objects that can be enumerated by an iterator, and allows the programmer the flexibility to express the reducer as a LINQ query [4]. In LINQ, `GroupBy` returns an object of type `IGrouping` that is frequently subsequently processed by a reducer. The Naiad syntax includes the reducer in the call to `GroupBy` directly, since it simplifies some optimizations, and because `IGroupings` cannot be directly constructed in C#.

Among the binary operators in Table 3, the set operators, along with `Concat`, are functions of the frequency of each element in either collection. For example, `Concat` produces the collection where the frequencies of each element in either collection are added:

```
Collection<R,T> Concat<R,T>(Collection<R,T> input1,
                            Collection<R,T> input2)
```

The set operators `Union`, `Intersect` and `Except` vary from their LINQ namesakes by acting on multisets as needed for Naiad collections where elements have counts. For instance, the Naiad version of `Union` returns a collection in which the count of each record is the maximum over the two input collections. In contrast, in the LINQ version of `Union` the output is a set containing each element that appears in either of the two input sets. The other Naiad set operators `Intersect` and `Except` are defined analogously. Set semantics can be recovered by post-applying the `Distinct` operator.

Finally, `Join` is based on the relational equi-join operator, $\bowtie$, which logically computes the cartesian product of two input collections and emits pairs of records which map to the same key:

```
Collection<W,T>
  Join<R,S,K,W,T>(Collection<R,T> input1,
                  Collection<S,T> input2,
                  Func<R,K> key1,
                  Func<S,K> key2,
                  Func<R,S,W> selector)
```

Type	Operators	Comment
Ordering	OrderBy	See text
	OrderByDescending	
	ThenBy	
	ThenByDescending	
	Reverse	
	Skip, SkipWhile	
	Take, TakeWhile	
	SequenceEqual	
	First, FirstOrDefault	
	Last, LastOrDefault	
Unplanned	Average	Use Sum and Count
	Contains	Use Where and Empty
	Single	
	DefaultIfEmpty	Convenience method
	SingleOrDefault	Convenience method
	GroupJoin	Use CoGroupBy
	All, Any, Empty	Use Select

Table 4. LINQ operators not implemented in Naiad.

In Naiad as in LINQ, `Join` does not emit pairs of records, but rather applies the function `selector` to each pair of records with matching keys.

Table 4 lists the LINQ operators (for data processing) that we have not implemented in Naiad. Most of the unplanned operators are convenience methods and can be derived from others. Several are still "to do" simply because they have not been needed for any of the application programs written so far. The most notable omissions from the Naiad operators are the ordering functions, a consequence of Naiad's use of multisets that was discussed at the end of Section 2.

A.2 New Naiad operators

The operators introduced by Naiad are listed in Table 5. `CoGroupBy` is the most general binary operator, analogous to `GroupBy`, but does not exist in LINQ:

```
Collection<W,T>
  CoGroupBy<R,S,K,W,T>(Collection<R,T> start,
                       Collection<S,T> other,
                       Func<R,K> key1,
                       Func<S,K> key2,
                       Func<K,
                            IEnumerable<R>,
                            IEnumerable<S>,
                            IEnumerable<W>> reducer)
```

It groups records by the supplied key, and applies the reduction function to corresponding pairs of groups if either is non-empty. `CoGroupBy` simplifies the implementation of a number of algorithms, and we believe it was an unfortunate omission from LINQ: its inclusion in Naiad is not specifically related to incremental computation or differential dataflow.

Operators `FixedPoint`, `Prioritize`, and `ExtendTime` are used to access the core Naiad functionality that arises from the differential dataflow model and are explained below. `Consolidate` is used as a performance hint to specify locations in the dataflow graph where it may be useful to pause for record cancellation. As an example, consider a `Select` operator that classifies incoming records, so each output record is a member of a small finite alphabet. Since `Select` is a stateless, streaming operator, it will output a long sequence of records each with weight 1. The addition of a `Consolidate` operator after `Select` would replace this long sequence with a more compact multiset representation, in which each alphabet symbol appears only once for any given lattice time, with its weight corresponding to the number of times the `Select` emitted it at that time. `Monitor` is purely a debugging aid, allowing user code to report statistics of the records as they pass through.

Type	Operator	Stateful?	Comment
Unary	FixedPoint	Yes	
	Prioritize	No	
	ExtendTime	No	
	Consolidate	No	Tests for cancellation of records
	Monitor	No	Debugging convenience: runs user-specified lambda on a list
Binary	CoGroupBy	Yes	Analogous to GroupBy

Table 5. New Naiad operators.

A.2.1 `FixedPoint`

`FixedPoint` in Naiad is a declarative operator specifying (potentially unbounded) iteration. The programmer provides an input collection and a function that will be repeatedly applied to the collection until a fixed point is reached:

```
Collection<R,T>
  FixedPoint<R,T>(Collection<R,T> input,
                  Func<Collection<R,U>,
                       Collection<R,U>> f)
```

Conceptually, the `FixedPoint` operator returns $\texttt{f}^{\infty}\texttt{(input)}$. If the repeated application of `f` to `input` has a fixed point, there will exist some n such that $\texttt{f}^{i}\texttt{(input)} = \texttt{f}^{i+1}\texttt{(input)}$ for all $i \geq n$. If not, the result is undefined and the computation may diverge.

In the fixed-point function, `f`, the lattice type `U` is the augmentation of `T` with an additional integer component that corresponds to the current iteration count.

A.2.2 `Prioritize`

In many data-parallel programming models, operators are applied to every record in a collection at once. In a Naiad program, the programmer can use the lattice-based times to specify the order in which elements in the same collection are processed. Systems such as PrIter [30] have used the same intuition to accelerate the convergence of many algorithms in a MapReduce setting. In Naiad, we introduce the `Prioritize` operator, which is a declarative form of PrIter's priority queue:

```
Collection<R,T>
  Prioritize<R,T>(Collection<R,T> input,
                  Func<R,int> priority,
                  Func<Collection<R,U>,
                       Collection<R,U>> f)
```

As with `FixedPoint`, the `Prioritize` operator extends the lattice element associated with each record in the input, and reverts to the original lattice in the output. The `priority` function in the above prototype associates an integer with each record, and the operator constructs a record in a new lattice, `U`.

Prioritization has an effect only when `f` contains a `FixedPoint` operator (e.g. the connected components query of Figure 3). In this case, the records will appear to be injected into the body of `FixedPoint` computation ordered first by their priority, then by their original time in `T`. If processing the high-priority elements first leads to less variation in the collections (e.g. by "locking in" minimal values in the connected components algorithm) the size of the intermediate data collection will be more compact and require less computation.

A.2.3 `ExtendTime`

In order for the types of lattices of collections to agree, the user sometimes has to manually extend them. Consider for example the connected components program of Figure 3, in which the lattice types were deliberately omitted for clarity. In fact, the `LocalMin` function requires its arguments to be collections with the same lattice type. However, the `FixedPoint` operator extends the lattice of `nodes` with an additional loop coordinate, so in the invocation

```
nodes.FixedPoint(x => LocalMin(x, edges))
```

the `x` and `edges` collections have different lattice types, since `x` has been extended with the loop coordinate and `nodes` has not.

The user deals with the corresponding mismatch using the `ExtendTime` operator to add a corresponding coordinate to `edges`:

```
nodes.FixedPoint(x => LocalMin(x,edges.ExtendTime()))
```

Since `edges` does not vary over the fixed-point execution, the extended lattice coordinate of every record in `edges` is set to 0.

There is an analogous `ExtendTime` method for prioritization, which takes a priority function and sets the extended coordinate using that function.