Naiad: The Animating Spirit of Rivers and Streams

Frank McSherry Rebecca Isaacs Michael Isard Derek G. Murray

Microsoft Research

1 Introduction

Distributed data-parallel computation has taken systems research by storm in recent years, with shared-nothing architectures allowing tremendous scale. MapReduce [1] led with a simple programming model consisting solely of pairs of map and reduce operations. However, this simplicity exposed new design challenges, in particular the constraints of the MapReduce model introduce inefficiencies for a large class of programs. Since, research has focused on extending the expressiveness of the programming model without compromising on scalability or performance. Dryad [5] generalized the representation of the computation to a directed acyclic graph, while DryadLINQ [9] elevated the level of abstraction to a declarative subset of C#, enabling it to understand a program's intent and thereby optimize its behavior substantially. Even so, many potentially data-parallel computations involve iteration or incremental computation, and efficiently executing programs written in this more expressive form remains a challenge.

In Naiad, the underlying computation framework provides a mechanism for iteration and incremental update, while the language provides a structured means of expressing these computations. This yields both an expressive programming model and an efficient implementation. For instance, loops are cast as fixed-point computations in which the per-iteration execution time decreases—ultimately to nothing—as the computation converges, which compares favorably with existing systems that reprocess entire datasets in each iteration. A key characteristic of Naiad is that processing a set of updates takes effort proportional to the number of updates, rather than the size of the collection they update. This is easily done if the entire collection fits in memory, but the scale of data and shared-nothing architecture prevent this in traditional systems. Our observation is that, by expressing the computation declaratively, Naiad can identify and restrict the data that needs to be resident in memory at any time, and—in many cases—opportunistically retain and re-use existing in-memory objects from prior executions.

1.1 Related Work

While many works have begun to investigate the challenges of streaming and iterative computation, we do not believe that any have yet achieved the sweet spot of a usable programming abstraction together with scalable performance. CIEL [2] has independent programming and execution models, which limits the potential for automatic optimizations. Spark [4] retains deserialized objects in memory with recovery information, but can only cache immutable collections. HaLoop [8] introduces looping primitives in Hadoop, but ultimately results in the same unrolled computations DryadLINQ would produce. Pregel [3] and PrIter [7] allow repeated MapReduce computations with in-memory state but fall short of general looping constructs, for example by restricting the loop body to be a single MapReduce step. Similarly, Incoop [6] supports incremental computation, but only for MapReduce-style computations.

2 System Design

Following prior work, Naiad uses a dataflow execution model, in which nodes represent data-parallel computations, and directed edges represent data flow. Similar to MapReduce and Dryad, the state associated

with each node is a collection of *records*, partitioned across many computers and/or threads. In the present implementation, each computer or thread, known as a *shard*, executes the entire dataflow graph, and keeps its fraction of the state of all nodes resident in local memory throughout. Execution occurs in a coordinated fashion, with all shards processing the same node at any time, and graph edges are implemented by channels that route records between shards as required. Unlike previous systems, a Naiad graph may contain *cycles*: therefore, all data produced on a channel is associated with an *epoch*, which (approximately) indicates the iteration count when a record was produced. Although our present implementation processes these iterations synchronously, we are currently developing a protocol for asynchronous execution.

Records in the dataflow constitute *increments*; a new record is incorporated into the node state, which is defined as the accumulation of all prior records along the same edge. This enables the program state to be updated efficiently, because the computation engine only does work when a record changes. We have re-engineered all data-parallel operators in LINQ (with the exception of a few order-sensitive operators) to support efficient incremental updates. We also introduce a *fixed-point operator* that introduces cycles into the dataflow graph, and converges when no new increments are produced on the internal edges of the loop.

The indexed state in our design corresponds to the increments seen along incoming edges. For fault tolerance, the state must periodically be persisted to disk, and the problem of random access to index data becomes one of cache management. By exposing the program's structure through the dataflow graph we make clear which data are required when (in particular, what data constitute the working sets of loops), and by persisting the data on disk we have recourse should we evict some indices to make space for others.

The result is a data-parallel computation platform not unlike DryadLINQ: programmers use a declarative language to express data-parallel computation over streams and with loops, which is then rendered to a distributed computation among coordinated peers. The difference from existing systems is the improved performance for incremental updates to existing inputs. This performance manifests in many computations, but most notably those involving streaming inputs and iteration.

3 Current Status

We are developing a prototype, currently running on a small number of computers, and are exploring the scalability and limitations of the system. We have found the behavior of the system on various graph algorithms to be promising, and even our parallel implementation of Strongly Connected Components, which contains a doubly-nested fixed point operation, is expressed elegantly and scales well.

The management of in-memory collections is likely to become the main challenge, as this is the most limiting aspect of the system and the crux of its performance. We are considering a variety of approaches to caching the requisite state, all exploiting our understanding of how and when the data are to be used. We are also carefully exploring opportunities for more relaxed scheduling, to take advantage of task and pipeline parallelism when data parallelism is scarce.

References

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

[2] Derek G. Murray *et al.* CIEL: a universal execution engine for distributed data-flow computing. In *8th USENIX NSDI*, Mar. 2011.

[3] Grzegorz Malewicz *et al.* Pregel: a system for large-scale graph processing. In *ACM SigMod*, June 2010.

[4] Matei Zaharia *et al.* Spark: Cluster computing with working sets. In *2nd USENIX HotCloud*, June 2010.

[5] Michael Isard *et al.* Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, Mar. 2007.

[6] Pramod Bhatotia *et al.* Incoop: MapReduce for incremental computations. In *ACM SOCC*, Oct. 2011.

[7] Yanfeng Zhang *et al.* PrIter: A distributed framework for prioritized iterative computations. In *ACM SOCC*, Oct. 2011.

[8] Yingyi Bu *et al.* HaLoop: Efficient iterative data processing on large clusters. In *36th VLDB*, Sept. 2010.

[9] Yuan Yu *et al.* DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *8th USENIX OSDI*, Dec. 2008.