Naiad: Timely Dataflow

Frank McSherry, Rebecca Isaacs, Michael Isard, and Derek G. Murray, Composable Incremental and Iterative Data-Parallel Computation with Naiad, MSR-TR-2012-105, 2012.

Distributed Dataflow Programming

map (key, value, context) {
    words = value.split(' '); 
    foreach (word in words) {
        context.write(word, 1);
    }
}

reduce (key, values, output) {
    output.collect(key, values.length);
}
Incremental and Iterative Processing

δ

Want to avoid starting from scratch
Incremental and Iterative Processing

δ

Need to introduce cycles into task flow.
Incremental Computation

Figure 2: A sequence of input collections $A_0, A_1, \ldots$ and the corresponding output collections $B_0, B_1, \ldots$. Each is defined independently as $B_t = \text{Op}(A_t)$.

Figure 3: The same sequence of computations as in Figure 2, presented as differences from the previous collections. The outputs still satisfy $B_t = \text{Op}(A_t)$, but are represented as differences $\delta B_t = B_t - B_{t-1}$. 
Synchronous vs Asynchronous

**Batching** (synchronous) vs. **Streaming** (asynchronous)

- **Batching**
  - ✗ Requires coordination
  - ✓ Supports aggregation

- **Streaming**
  - ✓ No coordination needed
  - ✗ Aggregation is difficult
Programming Model: Messages

B. **SEND**BY(edge, message, time)

C. **ONRECV**edge, message, time)

**Messages** are delivered asynchronously
Programming Model: Notifications

No more messages at \textbf{time} or earlier \quad \rightarrow \quad D.\texttt{ON\_NOTIFY}(time)

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

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

        counts[time][msg]++;
    }

    void OnNotify(T time)
    {
        foreach (var pair in counts[time])
            this.SendBy(output2, pair, time);
        counts.Remove(time);
    }
}
Differential computation in which multiple independent collections $B_{ij} = Op(A_{ij})$ are computed. The rounded boxes indicate the differences that are accumulated to form the collections $A_{11}$ and $B_{11}$. 

Figure 4: Differential computation in which multiple independent collections $B_{ij} = Op(A_{ij})$ are computed. The rounded boxes indicate the differences that are accumulated to form the collections $A_{11}$ and $B_{11}$. 

Differential Computation
Revisiting Iteration

Ingress: add new dimension

Feedback: advance timestamp

Egress: strip dimension
Engineering for low latency

- Reduce TCP delayed ACK and retransmit times
- Use finer grain scheduling timers to reduce impact of data structure contention
- Reduce impact of garbage collection by modifying GC parameters and utilising reusable types.
Evaluation

CDFs for 24 hour windowed SCC of @mention graph.
Evaluation

PageRank

Twitter graph
42 million nodes
1.5 billion edges

64 x 8-core 2.1 GHz AMD Opteron
16 GB RAM per server
Gigabit Ethernet

Iteration length (s)

Number of computers
Evaluation

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