Tagged Dataflow: A Formal Model for Iterative MapReduce

A. Charalambidis, N. Papaspyrou and P. Rondogiannis

Algorithms for MapReduce and Beyond
March 2014

Work supported by the project Handling Uncertainty in Data Intensive Applications, co-financed by the European Union (European Social Fund - ESF) and Greek national funds, through the Operational Program "Education and Lifelong Learning", under the program THALES.

Outline

Outline

Pros and Cons of MapReduce:

The introduction of MapReduce has been an important step towards the efficient processing of massive data.

Pros:

- It is simple and declarative.
- It runs on commodity clusters.
- It handles effectively task failures.

Cons:

- Many problems can't be easily solved with map and reduce tasks.
- It lacks support for *iteration*.
- It lacks a formal underlying model of parallelism.

MapReduce vs Dataflow

There have been proposed various sophisticated extensions of MapReduce, but the question still remains: "*What is the appropriate formal model behind the iterative extensions of MapReduce?*"

The main contributions of this paper:

- We argue that there exist close connections between MapReduce and the *dataflow computational model* (of the 70s and 80s!).
- We extend the dataflow model to better suit the needs of modern MapReduce-based systems.

Disclaimer:

- We are very new in the MapReduce area (so please double-check our claims).
- We are pretty knowledgeable regarding dataflow (so you can trust us more in this respect).
- Our paper and presentation are not really technical (but if what we claim is true, we promise to write more technical stuff on this issue).

Outline

Why is iteration needed?

There exist many natural algorithms that require some form of iteration:

- The transitive closure of a graph.
- The PageRank algorithm.
- Algorithms in data clustering, machine learning, etc.

Extensions that Support Iteration:

The problem of supporting iteration has been addressed by a number of research works:

- HaLoop supports a synchronous form of iteration (the next iteration starts when every task of the previous iteration has finished its work).
- Certain systems address the issue of iteration in the presence of stream data (D-Streams, Naiad, Hadoop Online, etc).
- Afrati *et al.* perform a theoretical investigation of the iterative execution of recursive Datalog queries in which tasks need not be synchronized.

Can we find a pattern in existing solutions?

Certain systems appear to support some form of *tagging* or *time-stamping* in order to discriminate between data that belong to different iteration levels or different versions of the input.

Questions:

- Is this tagging mechanism just an ad-hoc solution or an instance of a more general computational model?
- Can tagging be used to implement general forms of iteration?
- Can tagging be used to implement recursion?

Outline

The Dataflow Model of Computation

The *dataflow model of computation* was developed more than thirty years ago, as an alternative to the "von-Neumann" model of computation.

Dataflow Graphs:

A dataflow graph is a directed graph in which vertices correspond to processing elements and edges correspond to channels. The data that need to be processed start "flowing" inside the channels; when they reach a node they are being processed and the data produced are fed to the output channels of the node.

Since various parts of the dataflow graph can be working concurrently, the parallel nature of the model should be apparent.

Dataflow Evolution:

In the first steps of dataflow, data were assumed to flow in a specific order inside channels (*pipeline dataflow*). It soon became apparent that a model that would not impose any particular temporal ordering of the data would be much more general and useful.

Tagged Dataflow:

In *tagged-dataflow* data flow is accompanied by *tags* (i.e., labels). Tags make dataflow much more asynchronous since data need not be processed in any particular predetermined order.

Tags and Iteration:

Tags can carry essential information that can be used in order to implement iterative or even recursive algorithms.

The Manchester Dataflow Machine [1985]:

"Each separate iteration (of a loop) reuses the same code but with different data. To avoid any confusion of operands from the different iterations, each data value is tagged with a unique identifier known as the iteration level that indicates its specific iteration. Data are transmitted along the arcs in tagged packets known as tokens."

Formal Model of Tagged Dataflow:

Channels:

Channels carry tuples of the form $\langle \mathsf{t}, \mathsf{d} \rangle$ where d is an element of a data domain D and t is an element of a set of tags T. For every tag t, an edge can contain at most one token $\langle \mathsf{t}, \mathsf{d} \rangle$. Therefore, edges are functions in $\mathsf{T} \to \mathsf{D}$.

Nodes:

A processing node of a dataflow network that has $\mathfrak{n} \geq 1$ inputs and $\mathfrak{m} \geq 1$ outputs is a function in $(\mathsf{T} \to \mathsf{D})^{\mathfrak{n}} \to (\mathsf{T} \to \mathsf{D})^{\mathfrak{m}}$.

Outline

A Simple Application

Example

Compute all nodes of a `graph` reachable from a `start` node:

```
reach(Y) :- start(Y).
reach(Y) :- graph(X,Y), reach(X).
```

This can be easily performed using the techniques of Afrati *et al.* A dataflow network is constructed consisting of *join* and *dup-elim* nodes.

A Simple Application (continued...)

Example

Compute all nodes of a `graph` reachable from `a`; then compute all nodes reachable from `b`.

```
reach(Y) :- start(Y).
reach(Y) :- graph(X,Y), reach(X).
```

We can not run the two queries in parallel since we have no way of distinguishing to which of the two queries each reachable node belongs.

A Simple Application (continued...)

Example

Add a tag to the predicates of the program (or in the underlying implementation).

```
reach(T,Y) :- start(T,Y).
reach(T,Y) :- graph(X,Y), reach(T,X).
```

If we start with two queries `start(1,a)` and `start(2,b)` we can distinguish the results from their first (tag) component.

An Extension of Traditional Dataflow

Two remarks:

- For a given tag `t` we can have more than one identical atoms of the form `reach(t,node)` appear during the execution.
- For a given tag `t` we can have two or more atoms of the form `reach(t,node1)`, `reach(t,node2)` appear during the execution.

Extended Formal Model of Tagged Dataflow:

Channels:

Channels are *multisets* over $\mathsf{T} \times \mathsf{D}$, i.e., elements of $\mathcal{M}(\mathsf{T} \times \mathsf{D})$.

Nodes:

A processing node of a dataflow network that has $\mathfrak{n} \geq 1$ inputs and $\mathfrak{m} \geq 1$ outputs is a function in $[\mathcal{M}(\mathsf{T} \times \mathsf{D})]^{\mathfrak{n}} \to [\mathcal{M}(\mathsf{T} \times \mathsf{D})]^{\mathfrak{m}}$.

It can be seen that the *join* and *dup-elim* nodes conform to the above description.

Some Results:

Recursive Doubling:

We have devised a new algorithm for computing the transitive closure of an arbitrary relation by *recursive doubling*, using the tagged approach sketched above.

Dataflow Mergesort:

We have proposed a distributed mergesort procedure based on the tagged approach. In this procedure, the tags used are lists of natural numbers.

Outline

Open Questions:

Kahn Principle

A dataflow network built from functional nodes is the least fixpoint of a system of equations associated with the network.

Question:

Does (a version of) the Kahn principle hold for the generalized version of dataflow introduced in the paper?

The Kahn principle gives a formal way for reasoning about programs (e.g., by using fixed-point induction).

Open Questions:

Dataflow Programming Languages

There have been designed several dataflow programming languages in the past. Many of them use the notion of *tag* in an essential way. Most of them are functional (and some of them are logic-based).

Question:

Can we reuse ideas from dataflow languages in order to build languages for processing massive data?

Open Questions:

Fault tolerance:

The issue of fault tolerance in tagged dataflow has not been widely studied.

Question:

Can we build a theory of fault tolerance for tagged dataflow?

Open Questions:

Implementing Recursion:

It has been demonstrated that one can implement all first-order recursive functional programs using a tagged-dataflow, demand-driven approach [Rondogiannis & Wadge, JFP, 1997].

Question:

Can we use (a variant of) the above approach in order to implement recursive algorithms in a MapReduce context?

Thank you for your attention!

Recursive Doubling

Main Idea

Recursive doubling computes the nonlinear version of transitive closure and still does not discover each path many times. Each path consists of a prefix whose length is a power of 2 and a suffix whose length is no greater than the length of the prefix.

In round i we compute:

1. $P_i(x, y)$ the pair of nodes (x, y) whose shortest path is of length no more than $2^i - 1$.
2. $Q_i(x, y)$ the pair of nodes (x, y) whose shortest path is of length exactly 2^i.

Implementation of Recursive Doubling

Synchronously with two pairs of join-dupelim

Recursive doubling can be implemented using one pair of join-dupelim to compute relation Q_i and one pair of join-dupelim to compute relation P_i.

Asynchronously with one pair of join-dupelim and tagging

We tag each pair of node (x, y) with its iteration number and length and we enhance join-dupelim to handle tags. We can now distinguish between the edges that must be joined by their tag.

Remark

If the initial graph changes over time (new edges are added) we don't need to recompute the transitive closure over again as in the synchronous case.