Tagged Dataflow: a Formal Model for Iterative Map-Reduce*

Angelos Charalambidis
University of Athens
a.charalambidis@di.uoa.gr

Nikolaos Papaspyrou
National Technical University of Athens
nickie@softlab.ntua.gr

Panos Rondogiannis
University of Athens
prondo@di.uoa.gr

ABSTRACT

In this paper, we consider the recent iterative extensions of the Map-Reduce framework and we argue that they would greatly benefit from the research work that was conducted in the area of dataflow computing more than thirty years ago. In particular, we suggest that the *tagged-dataflow* model of computation can be used as the formal framework behind existing and future iterative generalizations of Map-Reduce. Moreover, we present various applications in which the tagged model gives elegant solutions with increased parallelism. The tagged-dataflow approach for iterative Map-Reduce creates a number of interesting research challenges which deserve further investigation, such as the requirement for a more sophisticated fault tolerance model.

1. INTRODUCTION

The introduction of Map-Reduce [11] has been an important step towards the efficient processing of massive data. The success of the Map-Reduce framework is mainly due to its simplicity, its declarative nature, its ability to run on commodity clusters and its effective handling of task failures that occur during execution. Despite its huge success, Map-Reduce has often been criticized for a number of different reasons. For example, it has often been argued that not all problems can be effectively (and naturally) solved using map and reduce tasks. Moreover, it has often been stressed that the framework suffers from a lack of support for *iteration*, which is, without doubt, a cornerstone of most interesting forms of computation.

The realization of the above problems has led to the introduction of various extensions of the framework (see for example [16, 7, 8]) and to the development of new systems that try to overcome the shortcomings (such as for example [13, 19, 14, 27, 20, 9, 21]). It appears that the creation of a framework that preserves the advantages of Map-Reduce and at the same time lifts its shortcomings, is not an easy task. In particular, it seems that the clear and efficient support of iteration is far from straightforward. Furthermore, there does not exist at present a unifying theoretical framework that could form the basis of the iterative extensions of Map-Reduce. Such a framework would allow the theory of such systems to be properly developed. In particular, it would allow for a proper specification and analysis of proposed algorithms, for the development of a corresponding complexity theory, for the proof of correctness of proposed algorithms, and so on.

In this paper we argue that such a formal framework was actually developed more than thirty years ago, in a similar but somewhat more restricted context. More specifically, during the 70s and 80s, the *dataflow model* of computation was developed [10, 12], many dataflow architectures were built and several dataflow programming languages were proposed and implemented. The aim of the dataflow paradigm was to exploit the inherent parallelism that exists in many applications. The dataflow architectures were specialized parallel computers and the dataflow programs were designed to eventually run on such machines. Of course, the Internet was not fully developed at that time and the idea of parallelism over clusters of workstations did not exist. Moreover, the distributed processing of massive data was far less important at that time than it is today. However, the main ideas behind the dataflow programs of the past and the distributed Map-Reduce applications of the present, share many common characteristics. In particular, the effective handling of iteration was also an important problem in the dataflow era and was handled by the introduction of the so-called paradigm of *tagged-dataflow* [25, 5, 6].

The main contribution of the present paper is therefore to demonstrate how the ideas from the dataflow research area can be applied to the new area of iterative Map-Reduce. In particular, we demonstrate how the tagging schemes of dataflow computers can be used in order to achieve a more asynchronous form of iteration for the systems of today. The rest of the paper is organized as follows. Section 2 summarizes several extensions of Map-Reduce and discusses the way iteration is handled in these extensions. Section 3 presents the basic ideas of the traditional model of tagged-dataflow. Section 4 extends the classical tagged-dataflow model so as to be applicable to the Map-Reduce setting and demonstrates

*This research was supported by the project "Handling Uncertainty in Data Intensive Applications", co-financed by the European Union (European Social Fund) and Greek national funds, through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Program: THALES, Investing in knowledge society through the European Social Fund.

how the new model can be used in order to specify various applications on massive data that appear to require iteration in their implementation. Finally, Section 5 describes possible directions for future work.

2. ITERATIVE MAP-REDUCE

Despite its simplicity and usefulness, Map-Reduce [11] has certain shortcomings that restrict its wider applicability. First, the framework allows only two types of tasks, namely the map and reduce tasks; obviously, it is not always possible (or natural) to model any problem using these two types of tasks. The flow of the data is also rather rigid since the output of the map tasks is used as input only to the reduce tasks. Again, there exist problems whose distributed solution requires a more involved flow of data: in many applications the data must be processed iteratively before the final output is obtained. Finally, Map-Reduce imposes the so-called *blocking property*: the reducers start processing data when the mappers have completely finished their work. This property is crucial in order to deal with task-failures: when a task fails, we can restart it from the beginning. Therefore, the blocking property ensures that the data are not "garbled" and that the computation proceeds in clear, discrete steps. However, the blocking property obviously has its disadvantages. During the time that the map tasks are processing, the reduce tasks are idle waiting for the map computation to complete and this obviously reduces the amount of potential parallelism.

2.1 Iteration

A natural generalization of Map-Reduce is to allow the use of arbitrary tasks. The systems Dryad [16] and Hyracks [7] generalize Map-Reduce by allowing the use of a set of arbitrary operators connected as a directed acyclic graph.

Apart from generalizing the types of tasks, a significant extension of Map-Reduce is the support of iteration. Many common data analysis algorithms require some form of iteration in order to be appropriately implemented. For example, the PageRank algorithm is a recursive algorithm that is usually implemented as an iteration until a fixed point is reached. One can attempt to implement such algorithms in the Map-Reduce framework using ad hoc techniques, but this is neither a natural nor efficient approach. HaLoop [8] tackles this inadequacy of Map-Reduce by providing iteration-related constructs that allow iterative algorithms to be expressed more succinctly. A characteristic of HaLoop (as well as other similar systems) is that iteration is performed in a synchronized manner, i.e., the next iteration starts when every task of the previous iteration has completely finished its work. A shortcoming of this approach is that tasks that have completed their processing remain idle until the next iteration starts, and therefore the potential parallelism is not fully exploited.

There have also been proposals for a more asynchronous form of iteration. For example, Afrati *et al.* in [3] perform a theoretical investigation of the iterative execution of recursive Datalog queries in an extended Map-Reduce environment. In that work, the task graph is not necessarily acyclic and the execution of tasks need not be synchronized. In other words, in the framework of [3] the blocking property is lifted.

Recapitulating, it appears that in current and future extensions of Map-Reduce, iteration is a vital component. It also appears that iteration can either be synchronous or asynchronous, depending on the objectives for which a particular system has been built.

2.2 Iterative Stream Processing

The problem of iteration becomes much more challenging when combined with the processing of stream data or stream queries. In many applications, the data to be processed are not always fully available at the time of execution. For example, in a social network, the edges and the vertices of the friends-of-friends graph can be added or removed dynamically. In such cases, the data are often most valuable as soon as they are produced (i.e., it is not possible for the data to be delayed and processed later as part of a batch). This state of affairs leads to the quest for iterative extensions of Map-Reduce that also take into account the temporal nature of the incoming data. A similar situation occurs when we would like to process a stream of different queries over the same data. If we would like to process the queries asynchronously (i.e., start processing a query before the previous one has completed execution), and if each query involves iterative computations, then care must be taken so as that the data produced during the processing of one query will not affect the processing of the other ones.

Certain extensions of Map-Reduce have been proposed that can handle situations such as the above. The system D-Streams [27] handles streaming input in a fault tolerant manner. The Naiad system realizes the concept of the differential dataflow [20] in which operators act upon "difference input traces" (i.e., the set of changes with respect to the previous input) in order to produce difference traces of output. The output then can be constructed by combining all the difference traces. The difference traces are indexed both by the version of the input and the iteration number. As a result the computation of an iterative incremental algorithm is drastically reduced since the framework can reuse computations done both in previous versions of the input and in previous iterations. Hadoop Online [9] is an extension of Map-Reduce that supports pipelined stream queries. In [21] multiple similar Map-Reduce jobs are combined in order to be executed as one. The key idea is to add tags in the data, in order to distinguish between different jobs.

The systems mentioned in this last subsection appear to be using a common technique in order to handle iteration in streaming data or queries: they employ some form of *tagging* in order to discriminate between the data that belong to different iteration levels and different versions of the input (or different queries). In the next sections we argue that this tagging mechanism is not just a coincidence, but instead a more general mechanism that can be used in order to implement arbitrary iteration in a generalized Map-Reduce framework.

3. TAGGED-DATAFLOW

The *dataflow model of computation* [10, 12] was developed more than thirty years ago, as an alternative to the classical "von-Neumann" computing model. The key motivation was the creation of architectures and programming languages that would exploit the massive parallelism that is inherent in many applications. A dataflow program is essentially 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. Moreover, this processing of data "while in motion" comes in sharp contrast with the traditional "von-Neumann" model in which data wait passively in memory until they are fetched by the central processing unit of the (sequential) computer in order to be processed.

The dataflow model was extensively studied during the 70s and 80s. Many theoretical results were obtained, many dataflow programming languages were developed and several dataflow machines were built. In the beginning of the 90s, the dataflow research area started to decline. The main reason was the fact that dataflow hardware never proved extremely successful and never achieved a performance that would justify the massive production and use of dataflow machines. However, the theory and the programming languages that were developed were quite sophisticated, leaving much hope for further developments in case the hardware problems were ever bypassed.

In the initial dataflow model (usually called *pipeline dataflow*), channels were assumed to be unbounded FIFO queues, i.e., the data were assumed to flow in a specific order inside the channels. However, 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. This resulted in the so-called *tagged-dataflow* model [25, 5, 6]. The basic idea behind tagged-dataflow is that data can flow inside a network accompanied by *tags* (i.e., labels). The use of tags makes dataflow much more asynchronous since data need not be processed in any particular predetermined order. Moreover, as we are going to see, the tags can carry essential information that can be used in order to implement iterative or even recursive algorithms. A key notion in our foregoing discussion is that of a *dataflow graph*:

Definition 1. A *dataflow graph* (or *dataflow network*) is a directed graph $G = (V, E)$, where V is the set of *nodes* of the graph and E is the set of edges connecting elements of V. The set V is partitioned into disjoint subsets V_I (*input nodes*), V_O (*output nodes*) and V_P (*processing nodes*), subject to the following restrictions:

- Every input node has no incoming edges and has one outgoing edge towards a processing node.
- Every output node has no outgoing edges and has one incoming edge from a processing node.
- Every processing node has incoming edges (at least one) from input nodes and/or from other processing nodes and outgoing edges (at least one) to output nodes and/or other processing nodes.

Intuitively, input nodes provide the input data to the dataflow graph, processing nodes are performing the processing of data and output nodes are collecting the output data produced by the network.

The semantics of dataflow networks can be given with standard techniques of denotational semantics [23]. Our presentation below follows the exposition given in [26]. Intuitively, edges of our dataflow networks carry tuples of the form $\langle t, d\rangle$ where d is an element of a data domain D and t is an element of a set of tags T. The set T may be quite involved; in its simplest form it can be a set of natural numbers, or in more demanding cases it can be the set of lists of naturals numbers, etc. Pairs of the form $\langle t, d\rangle \in T \times D$ are usually referred in the dataflow literature as *tokens*. It is often assumed that for a given tag t, an edge can contain at most one token $\langle t, d\rangle$. In other words, it is a standard assumption that edges correspond to functions in $T \to D$. As we are going to see in the next section, this assumption needs to be extended when considering applications in the Map-Reduce framework.

Consider now a processing node of our dataflow graph. A standard assumption in dataflow computing is that processing nodes are functions that transform their inputs to outputs. There have been extensions of dataflow that support non-functional nodes, for example, non-deterministic ones [1]. However, such extensions will not be considered here. It should be noted that determinism is also a key assumption in Map-Reduce systems (see, for example, [11, page 109]). Therefore, based on the foregoing discussion, a processing node f of a dataflow network that has $n \geq 1$ inputs and $m \geq 1$ outputs is a function[1] in $(T \to D)^n \to (T \to D)^m$. The input and output nodes are in fact equivalent to channels, ie., they are functions in $T \to D$.

In the rest of the paper we will refer to the class of dataflow networks presented above as *functional dataflow networks* (since channels are functions). We will present an extension of this model in the next section.

Example 1. We demonstrate these ideas with the well-known example of Hamming numbers (first posed as a programming problem by Edsger Dijkstra). Recall that Hamming numbers are all numbers of the form $2^i \cdot 3^j \cdot 5^k$ where i, j, k are non-negative integers. The problem is to enumerate the Hamming numbers in numeric order and a dataflow solution of it is depicted in Figure 1. We explain the nodes that appear in the figure. First consider the nodes that contain constants (such as 1, 2, etc). One can assume that these nodes produce constant streams. For example, the node 5 is nothing more than the constant function in $\mathbb{N} \to \mathbb{N}$ which assigns to every $n \in \mathbb{N}$ the constant value 5. The *fby* node (read as "followed-by"), is a function in $(\mathbb{N} \to \mathbb{N}, \mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N})$. The operation of *fby* can be intuitively described as follows: initially, it checks its first input until a token of the form $\langle 0, d\rangle$ arrives, and as soon as it sees such a token, it delivers it to its output channel. From that point on, *fby* never consults again its first input but it continuously checks its second input. Every time a token of the form $\langle t, d\rangle$ arrives in its second input, the node puts in its output channel the token $\langle t + 1, d\rangle$. It can be easily checked that this operational behavior corresponds to the following functional definition of *fby*:

$$fby(X, Y)(t) = \begin{cases} X(0) & \text{if } t = 0 \\ Y(t-1) & \text{if } t > 0 \end{cases}$$

The *merge* node is also a function in $(\mathbb{N} \to \mathbb{N}, \mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N})$. Given two inputs $X, Y \in \mathbb{N} \to \mathbb{N}$ that are increasing functions, *merge* produces as output an increasing function that results from merging the two inputs. For the

[1] Actually, computability reasons dictate that the functions corresponding to the nodes of our networks have to be additionally *continuous* (see [23] for a standard introduction on this issue and [18] for a corresponding discussion regarding pipeline dataflow networks).

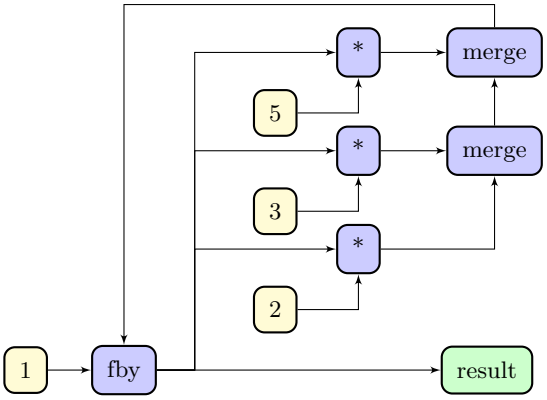

Figure 1: The dataflow representation of Hamming numbers.

formal (recursive) definition of *merge*, see [24]. The "*" nodes simply multiply the data part of the token that arrives in their input, with the constant that exists in their other input.

One can easily verify that the network just described, produces the sequence of Hamming numbers.

4. TAGGED-DATAFLOW AND ITERATIVE MAP-REDUCE

In this section we argue that a mild extension of the tagged-dataflow model of computation can be used as the formal framework behind existing and future iterative generalizations of Map-Reduce. We present various applications in which the tagged model gives elegant solutions with increased asynchronous parallelism.

It is important to clarify what exactly is being claimed by the following examples. First, we argue that the *implementations* of iterative Map-Reduce systems should support tags and tag-manipulation operations in the same way that tagged-dataflow machines of the 80s supported such operations [15, 6]. The tags can be used to ensure the implementation of asynchronous iteration in an elegant and effective way. The main idea is precisely defined in the following excerpt from [15]:

> *Each separate (loop) iteration 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.*

The second thing that is being claimed below is that the languages used to program iterative Map-Reduce applications should also support the declaration and manipulation of tags. In this way the programmer will be free to declare and use the types of tags that are essential in the specific application being developed. It is important to stress that dataflow languages of the 80s and 90s allowed the declaration of user-defined tags (also called *dimensions*). For example, the latest versions of Lucid [24] and its extension GLU [17] allowed many different dimensions.

4.1 An Extension of Tagged-Dataflow for Map-Reduce

The basic principles of tagged-dataflow described in the previous section have been used in the design of many functional dataflow programming languages. In particular, the interpretation of channels as functions and of processing nodes as (second-order) functions, clearly emphasizes the connections between dataflow networks and functional programming.

However, there exist applications in which it is not sufficient for channels to be just functions from T to D. For example, it is conceivable for the same tag to be used in two different tuples that "flow" inside a channel (i.e., to have $\langle t, d_1 \rangle$ and $\langle t, d_2 \rangle$ appear in the same channel). Moreover, it is possible in certain cases to have the same tuple $\langle t, d \rangle$ appear more than once in a channel. In particular, as we are soon going to see, in the Map-Reduce framework both of these cases show-up quite naturally.

We are thus led to a generalization of the tagged-dataflow model in which channels are *multisets* over $T \times D$ and processing nodes take multisets as inputs and produce multisets as outputs. More formally, given a nonempty set S let us denote by $\mathcal{M}(S)$ the set of multisets (or *bags*) of elements of S. Then, in our extended tagged-dataflow model, a channel is an element of $\mathcal{M}(T \times D)$ while a node f of a dataflow network that has n inputs and m outputs is a function in $[\mathcal{M}(T \times D)]^n \to [\mathcal{M}(T \times D)]^m$. In the following subsections, we examine three different applications where this extended tagged model is applicable and useful.

4.2 Streaming Queries

One of the most promising applications of iterative Map-Reduce is in the execution of Datalog queries [3, 4]. The key idea here is that the least fixed point of a Datalog program can be computed in a bottom-up way using a network of *join* and *dup-elim* processes (see [3] for details). For example, assume that we have an EDB relation `edge` and also the following IDB relation:

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

Assume also that we want to locate all the nodes that are reachable from an initial vertex `a`, i.e., we assume that we also have the fact `start(a)`. Using the techniques of [3], one can easily construct a simple network that calculates the set of nodes reachable from `a`.

Assume however that we want to have multiple queries, e.g., we want to locate all the vertices reachable from `a`, `b` and `c`. Moreover, for reasons of efficiency, we want these queries to be run in parallel as much as possible (which means for example that the query for `b` should not wait for the computation of the query for `a` to complete before starting to execute itself). The problem with running the queries in an overlapped manner is obvious: the output of the Map-Reduce network will be a set of `reach` facts, but with no indication of which fact corresponds to which query. So, if we want to do some further processing on the nodes reachable from vertex `a`, we have no way of knowing which exactly these nodes are.

In the tagged setting, the support of such streaming queries is quite straightforward. Each different `start` query is tagged with a different natural number. Then, every time one of the `reach` rules is used to produce some new fact, the tag that

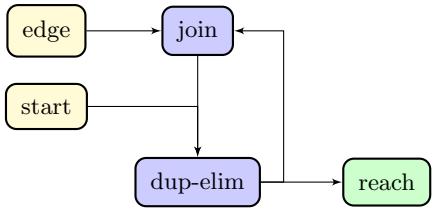

Figure 2: The set of join and dup-elim nodes working with streaming queries. Queries flow from the start node tagged with a distinct number.

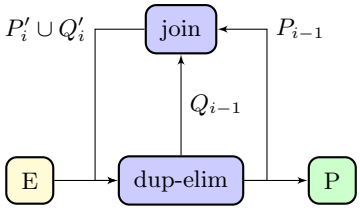

Figure 3: The set of join and dup-elim nodes that computes the transitive closure by recursive doubling.

has been used in the body of the rule is inherited by the fact that is produced in the head of the rule. In this way, at the end of the computation a set of tuples $\langle t, \texttt{reach(u)} \rangle$ is collected, and one can discriminate the results of the different queries by examining the tag t.

It is not hard to see how the above example fits the tagged-model introduced in Subsection 4.1. The dataflow is depicted in Figure 2. The channels of our network are elements of $\mathcal{M}(\mathbb{N} \times D)$; the data domain D is the set of tuples flowing in a particular channel. Notice that we need to have multisets because the same tuple under the same tag may appear in a channel more than one times. For example, the join processing node may produce many identical tokens. Consider now the two processing nodes, namely *join* and *dup-elim*. The join node takes as input two tag-extended relations; each such relation does not contain duplicates and is therefore an element[2] of $\mathcal{P}(\mathbb{N} \times D)$ (which is a special case of $\mathcal{M}(\mathbb{N} \times D)$). The output of the join is an element of $\mathcal{M}(\mathbb{N} \times D)$), since the joining of two relations may create duplicate tuples. Overall, the type of join is $[\mathcal{P}(\mathbb{N} \times D)]^2 \to \mathcal{M}(\mathbb{N} \times D)$. Consider now the *dup-elim* node, which eliminates duplicates from its input channel. Its type is $\mathcal{M}(\mathbb{N} \times D) \to \mathcal{P}(\mathbb{N} \times D)$. In fact, the dup-elim is nothing more than the function which given a multiset returns the corresponding underlying set.

4.3 Algorithms with Increased Asynchronicity

The transitive closure of an arbitrary relation can be calculated with the technique shown in the previous section, using a dataflow network that corresponds to an iterative Map-Reduce system. In this section, we calculate the transitive closure by *recursive doubling*, using a set of join and dup-elim nodes, as suggested in [2, 3]. In the form of pseudocode, the algorithm that we use (copied from [3]) is the following:

1: $Q_0 := E$
2: $P_0 := \{(x, x) \mid x \text{ is a graph node}\}$
3: $i := 0$
4: **repeat**
5: $\quad i := i + 1$
6: $\quad P'_i(x, y) := \pi_{x,y}(Q_{i-1}(x, z) \bowtie P_{i-1}(z, y))$
7: $\quad Q'_i(x, y) := \pi_{x,y}(Q_{i-1}(x, z) \bowtie Q_{i-1}(z, y))$
8: $\quad P_i := P'_i \bigcup P_{i-1}$
9: $\quad Q_i := Q'_i - P_i$
10: **until** $Q_i = \emptyset$

[2]By $\mathcal{P}(A)$ we denote the power-set of a given set A.

We deviate in the details of the implementation by introducing tags of the form (i, l) which are used to annotate every row (u, v). A tag conveys two pieces of information: i is the number of iteration in which a row was produced, and l is the length of the path. Hashing can be used to determine which node will receive a given tagged row, in the same way as in [3]; we omit the details here.

Each *join node* has two inputs, left and right, and two distinct internal stores, again left and right. The join is performed as follows:

1. For every tuple (x, z) with tag (i_1, l_1) in its left input, the node searches its right store for tuples (z, y) with tag (i_2, l_2) such that $l_2 \leq l_1$. For every such tuple it emits a tuple (x, y) with tag (i, l) where $i = \max\{i_1, i_2\} + 1$ and $l = l_1 + l_2$. The initial tuple (x, z) is stored in the left store together with its tag.

2. For every tuple (z, y) with tag (i_2, l_2) in its right input, the node searches its left store for tuples (x, z) with tag (i_1, l_1) such that $l_2 \leq l_1$. For every such tuple it emits a tuple (x, y) with tag (i, l) where $i = \max\{i_1, i_2\} + 1$ and $l = l_1 + l_2$. The initial tuple (z, y) is stored in its right store together with its tag.

Each *dup-elim node* receive tuples (x, y) with tag (i, l) and performs the following steps:

1. If the tuple already exists with smaller tag (i.e., either smaller length or same length and smaller iteration number) then it is ignored. If the tag exists with bigger tag then the tag is updated to the smaller one and the tuple continues.

2. If $l = 2^i$ then it is fed back both as left and as right input to the appropriate join nodes, otherwise it is fed only as right input.

Each tuple (x, y) in the relation E is sent to the corresponding dup-elim node with tag $(0, 1)$.

The left input of a join node corresponds to relation Q (the paths of size 2^i), whereas the right input corresponds to relation P. We choose to join each tuple of Q with every tuple of P of strictly smaller length. That join corresponds to line (6) of the recursive doubling algorithm. Moreover, we join each tuple of Q with every right tuple of equal length. That corresponds to line (7). Thus the output of the join is the union of both steps, but since we have tagged each tuple with the length of the path we can distinguish the two relations.

In fact, the dup-elim will pick only the tuples of size 2^i and redirect them to the left input of the join. But since

$Q_i \subseteq P_{i+1}$ those tuples will also be redirected to the right input of the join. All the other tuples will be redirected only to the right input as part of P. Suppose now that a new tuple is generated with iteration number i and the same tuple already exists in dup-elim store with different tag. If the tag has a smaller iteration j then the tuple exists in $P_j \subseteq P_i$. In that case the new tuple is ignored. If that tuple is for the relation Q, omitting it corresponds to line (9) of the algorithm. On the other hand, if it was for the relation P, this corresponds to line (8). Since the iteration is not synchronous, it may occur that the new tuple has smaller tag (smaller length) than the one already stored in the dup-elim. In that case, the tuple is not ignored since it precedes in time and the node will emit the tuple as if it is encountered for the first time. It will also update the tag with the smaller one, to prevent for other duplicates to propagate.

Note that the relations Q and P up to iteration i are accumulated in the left and right stores of join nodes, respectively. Moreover, the relation P is also stored in the dup-elim nodes.

The dataflow is slightly different of that described in Subsection 4.2. In this case the tags are elements of $\mathbb{N} \times \mathbb{N}$ and the channels are elements of $\mathcal{M}((\mathbb{N} \times \mathbb{N}) \times D)$. The type of join node is $[\mathcal{P}((\mathbb{N} \times \mathbb{N}) \times D)]^2 \to \mathcal{M}((\mathbb{N} \times \mathbb{N}) \times D)$. On the other hand, the type of the dup-elim node is $\mathcal{M}((\mathbb{N} \times \mathbb{N}) \times D) \to [\mathcal{P}((\mathbb{N} \times \mathbb{N}) \times D)]^2$. The dataflow graph is depicted in Figure 3. Note that the dup-elim node differs from the simple version since it has two output channels, one for each relation.

Consider the case where the structure of the graph changes, for example, new edges are added in a timely manner. A simple setup of the recursive doubling algorithm must restart the computation each time a change of the graph is detected. In the aforementioned tagged-dataflow graph a new edge will trigger the join of certain paths only, based on their tag, minimizing the redundant computations.

4.4 Recursive Algorithms

To our knowledge, the existing extensions of Map-Reduce have mainly dealt with iterative algorithms. However, not all algorithms can be expressed elegantly in an iterative way; there exist problems whose solution is more naturally expressed in a recursive manner. In the rest of this subsection we consider one such problem and at the end of the subsection we discuss the possibility of extending the proposed technique to all recursively defined functions.

The algorithm we present below is a distributed sorting procedure that is based on the tagged-dataflow approach. However, we do not claim at present that this is actually an *efficient* way to perform sorting in a distributed environment nor that this example is fully-compatible with the nature of Map-Reduce. The algorithm described below is only intended to demonstrate that the tagging mechanism can also be used to implement *recursive algorithms* in a distributed manner. Intuitively, the tags can be used in order to keep track of the paths in the recursion tree.

Let us assume that we would like to sort a considerably large list of data, which is possibly distributed in a number of nodes over the network. One possibility is to use *merge sort*: we split the list into two parts, we sort each one of them separately and then merge the results. This is clearly a recursive procedure. The problem is therefore how we can

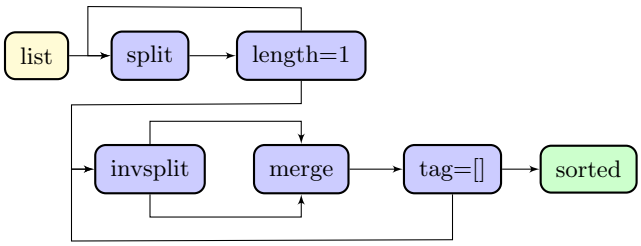

Figure 4: The mergesort depicted as dataflow.

implement the recursive *merge sort* in a distributed way, using tags to coordinate the whole process.

A *merge sort* function usually has the following form:

```
msort([])  = []
msort([x]) = [x]
msort(l)   = merge(msort(l1), msort(l2))
  where (l1, l2) = split(l)
```

The function `split` divides a list into two disjoint lists of half size and returns a tuple consisting of these two parts. The base cases of the recursion (`l == []` and `l == [x]`) need not be so fine-grained; for example, in a Map-Reduce context, when a list given to an intermediate step of merge sort is relatively small, we can sort it locally on a particular machine instead of continuing to divide it into smaller lists, which would circulate in the network and impose further communication costs. To simplify the presentation, we assume however that `msort` is defined as above and that the recursion reaches down to lists of trivial size.

The key idea behind the distributed implementation of merge sort is the following. We tag the all values in our list with two different tags. The first tag is a natural number (initially equal to 0) which will be used in the merge process of the algorithm. The second tag is a list of natural numbers (initially empty) which indicates the steps of the splitting procedure that have been applied on a particular element of the initial list. For example, if during the recursive execution of `msort` an element of the initial list was placed in the component `l1`, during the first invocation of `split`, and in the component `l2`, during the second invocation of `split`, then this element will be tagged by $[2, 1]$.

The distributed algorithm for merge sort is depicted in Figure 4. It contains two phases.

In the first phase, the algorithm starts by splitting the initial list into two sublists; this is achieved by the `split` node in Figure 4. The elements placed in the first component `l1` by `split` will have their list tag prefixed by 1, while those plaved in the second component `l2` will be prefixed by 2. This process of splitting and tagging is continued until all elements of the initial list have different tags; this is achieved by the `length=1` node, which checks whether the list of elements corresponding to a given tag contains a single element and, if not, returns all elements of the list to `split`. Every element whose tagging process has been completed, passes to the second phase of the dataflow network.

In the second phase of the algorithm, the `invsplit` node examines each element that arrives to it; if the head of the element's tag is 1, `invsplit` removes the head and sends the element to its left output; if the head is 2, it removes it and sends the element to the right output. The `merge` node sorts

all those elements that have the same list tag by changing the natural number tag so as to reflect the order of each element. E.g., if `merge` receives the element $\langle\langle 0, [2, 1]\rangle, \texttt{George}\rangle$ and the element $\langle\langle 0, [2, 1]\rangle, \texttt{Steve}\rangle$, then, since `George` is alphabetically first compared to `Steve`, the `merge` node will output $\langle\langle 0, [2, 1]\rangle, \texttt{George}\rangle$ and $\langle\langle 1, [2, 1]\rangle, \texttt{Steve}\rangle$. If in the next step it receives $[\langle\langle 0, [1]\rangle, \texttt{Ann}\rangle, \langle\langle 1, [1]\rangle, \texttt{Suzan}\rangle]$ in the left input and $[\langle\langle 0, [1]\rangle, \texttt{George}\rangle, \langle\langle 1, [1]\rangle, \texttt{Steve}\rangle]$ in the right input, then it will merge these two ordered sets and produce $[\langle\langle 0, [1]\rangle, \texttt{Ann}\rangle, \langle\langle 1, [1]\rangle, \texttt{George}\rangle, \langle\langle 2, [1]\rangle, \texttt{Steve}\rangle, \langle\langle 3, [1]\rangle, \texttt{Suzan}\rangle]$. The process will end when two ordered sets whose elements are all tagged with the empty list appear as inputs to `merge`; in this case, `merge` will do the final merging of these two sets, and the sorted file will come to the output.

The above procedure may seem ad hoc at first sight and one may assume that the distributed tag-based execution may not be applicable to all forms of recursion. However, this does not appear to be the case. A purely dataflow tag-based scheme for implementing first-order recursive functional programs was proposed in [26] and was theoretically justified in [22]. However, the scheme of [26] is appropriate for demand-driven execution while the main applications in the area of Map-Reduce appear to require a data-driven approach. The mergesort example presented above was obtained by adapting the technique of [26] to run in a data-driven way. Whether this can be done in general appears to be an interesting research problem.

5. FUTURE WORK

There exist several aspects of the connection between the Map-Reduce framework and tagged-dataflow that require further investigation. In the following, we outline certain such problems that we feel are particularly interesting and worthwhile for further study:

- Every functional dataflow network of the form presented in Section 3 can be shown to compute a function which is the least fixed point of a system of equations associated with the network. This result is an easy generalization of the so-called *Kahn principle* [18]. The least fixed point can be computed inductively, based on standard results of recursion theory (in particular, Kleene's fixed point theorem). Therefore, there exists a way of computing the *meaning* of functional dataflow networks or, in other words, we have a formal theory for reasoning about functional dataflow programs. It would be interesting to examine whether the Kahn principle also holds for the more general (i.e., multiset-based) dataflow networks introduced in Subsection 4.1. If this is the case, then this opens up the possibility of performing formal proofs of correctness for various dataflow algorithms (such as the transitive closure algorithm of Section 4).

- At present, not many specialized programming languages have been proposed that can be used to program applications for the processing of massive data. We feel that this is probably one of the next steps in the evolution of the area. Such languages would definitely benefit by adopting the features and philosophy of the dataflow programming languages of the past.

- Based on the model of tagged-dataflow, one should be able to define a formal theory of fault tolerance for dataflow networks. Since fault tolerance is a primary characteristic of (standard) Map-Reduce, it would be interesting to see how such a desired property can be ensured in the more general setting of the tagged-dataflow model.

We strongly believe that the further investigation of the interactions between dataflow and the novel approaches to distributed processing that have resulted from Map-Reduce, will prove very rewarding.

6. REFERENCES

[1] S. Abramsky. A generalized Kahn principle for abstract asynchronous networks. In M. G. Main, A. Melton, M. W. Mislove, and D. A. Schmidt, editors, *Mathematical Foundations of Programming Semantics*, volume 442 of *Lecture Notes in Computer Science*, pages 1–21. Springer, 1989.

[2] F. N. Afrati, V. R. Borkar, M. J. Carey, N. Polyzotis, and J. D. Ullman. Cluster computing, recursion and datalog. In O. de Moor, G. Gottlob, T. Furche, and A. J. Sellers, editors, *Datalog*, volume 6702 of *Lecture Notes in Computer Science*, pages 120–144. Springer, 2010.

[3] F. N. Afrati, V. R. Borkar, M. J. Carey, N. Polyzotis, and J. D. Ullman. Map-reduce extensions and recursive queries. In A. Ailamaki, S. Amer-Yahia, J. M. Patel, T. Risch, P. Senellart, and J. Stoyanovich, editors, *EDBT*, pages 1–8. ACM, 2011.

[4] F. N. Afrati and J. D. Ullman. Transitive closure and recursive datalog implemented on clusters. In E. A. Rundensteiner, V. Markl, I. Manolescu, S. Amer-Yahia, F. Naumann, and I. Ari, editors, *EDBT*, pages 132–143. ACM, 2012.

[5] Arvind and D. Culler. The tagged token dataflow architecture (preliminary version). Technical report, Laboratory for Computer Science, MIT, Cambridge, MA, 1983.

[6] Arvind and R. S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Trans. Computers*, 39(3):300–318, 1990.

[7] V. R. Borkar, M. J. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In S. Abiteboul, K. Böhm, C. Koch, and K.-L. Tan, editors, *ICDE*, pages 1151–1162. IEEE Computer Society, 2011.

[8] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient iterative data processing on large clusters. *PVLDB*, 3(1):285–296, 2010.

[9] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *NSDI*, pages 313–328. USENIX Association, 2010.

[10] A. L. Davis. The architecture and system method of DDM1: A recursively structured data driven machine. In E. J. McCluskey, J. F. Wakerly, E. D. Crockett, T. E. Bredt, D. J. Lu, W. M. van Cleemput, S. S. Owicki, R. C. Ogus, R. Apte, M. D. Beaurdy, and J. Losq, editors, *ISCA*, pages 210–215. ACM, 1978.

[11] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[12] J. B. Dennis and D. Misunas. A preliminary architecture for a basic data flow processor. In W. K. King and O. N. Garcia, editors, *ISCA*, pages 126–132. ACM, 1974.

[13] C. Engle, A. Lupher, R. Xin, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: fast data analysis using coarse-grained distributed memory. In K. S. Candan, Y. Chen, R. T. Snodgrass, L. Gravano, and A. Fuxman, editors, *SIGMOD Conference*, pages 689–692. ACM, 2012.

[14] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl. Spinning fast iterative data flows. *PVLDB*, 5(11):1268–1279, 2012.

[15] J. R. Gurd, C. C. Kirkham, and I. Watson. The Manchester prototype dataflow computer. *Commun. ACM*, 28(1):34–52, 1985.

[16] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.

[17] R. Jagannathan, C. Dodd, and I. Agi. GLU: A high-level system for granular data-parallel programming. *Concurrency - Practice and Experience*, 9(1):63–83, 1997.

[18] G. Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.

[19] 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 *SPAA*, page 48, 2009.

[20] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard. Differential dataflow. In *CIDR*. www.cidrdb.org, 2013.

[21] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. MRShare: Sharing across multiple queries in mapreduce. *PVLDB*, 3(1):494–505, 2010.

[22] P. Rondogiannis and W. W. Wadge. First-order functional languages and intensional logic. *J. Funct. Program.*, 7(1):73–101, 1997.

[23] R. D. Tennent. *Principles of programming languages*. Prentice Hall International Series in Computer Science. Prentice Hall, 1981.

[24] W. W. Wadge and E. A. Ashcroft. *LUCID, the Dataflow Programming Language*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[25] I. Watson and J. R. Gurd. A prototype data flow computer with token labelling. In *Proceedings of the National Computer Conference*, pages 623–628, 1979.

[26] A. A. Yaghi. *The Intensional Implementation Technique for Functional Languages*. PhD thesis, Department of Computer Science, University of Warwick, Coventry, UK, 1984.

[27] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: fault-tolerant streaming computation at scale. In M. Kaminsky and M. Dahlin, editors, *SOSP*, pages 423–438. ACM, 2013.