# The Dynamic Pipeline Paradigm

Cristina Zoltan
Computer Science Department
Universitat Politècnica de Catalunya
Barcelona, Spain
zoltan@cs.upc.edu

Edelmira Pasarella*
Computer Science Department
Universitat Politècnica de Catalunya
Barcelona, Spain
edelmira@cs.upc.edu

Julián Aráoz
Statistics and Operation Research Department
Universitat Politècnica de Catalunya
Barcelona, Spain
araoz@upc.edu

Maria-Esther Vidal
TIB Leibniz Information Centre for Science and Technology and
L3S Research Center, Hannover, Germany
maria.vidal@tib.eu

Nowadays, in the era of Big Data and Internet of Things, large volumes of data in motion are produced in heterogeneous formats, frequencies, densities, and quantities. In general, data is continuously produced by diverse devices and most of them must be processed at real-time. Indeed, this change of paradigm in the way in which data are produced, forces us to rethink the way in which they should be processed even in presence of parallel approaches. To process continuous data, data-driven frameworks are demanded; they are required to dynamically adapt execution schedulers, reconfigure computational structures, and adjust the use of resources according to the characteristics of the input data stream. In previous work, we introduced the *Dynamic Pipeline* as one of these computational structures, and we experimentally showed its efficiency when it is used to solve the problem of counting triangles in a graph. In this work, our aim is to define the main components of the Dynamic Pipeline which is suitable to specify solutions to problems whose incoming data is heterogeneous data in motion. To be concrete, we define the *Dynamic Pipeline Paradigm* and, additionally, we show the applicability of our framework to specify different well-known problems.

## 1 Introduction

Big data, and specifically the Internet of Things, play a relevant role in promoting both manufacturing and scientific development through industrial digitization and emerging interdisciplinary research. In fact, with the advances in the technologies for data generation and ingestion, data-driven frameworks able to provide flexible computation methods are demanded to solve even traditional problems like duplicate elimination or sorting. Specifically, data-driven frameworks enable adaptation of execution schedulers and computational structures to the current conditions of the input data; they play a relevant role for managing continuous data. Parallel programming has become one promising technique to implement flexible and scalable data-driven frameworks. Albeit adjustable, existing parallel paradigms like MapReduce, resort to the *Divide & Conquer* paradigm [1] where an instance of a problem is partitioned into subproblems and each of them, have to be completely executed in order to start the next one. This blocking behavior impedes the generation of results incrementally, and limits the flexibility demanded in real-world applications over the data generated by next generation of technologies for data collection.

**Problem and Research Objective:** We address the problem of flexible parallel computation and focus on *Dynamic Pipelines* (e.g., [2, 3, 5, 7]). A Dynamic Pipeline implements an asynchronous model of

*This research is partially supported by funds from the Spanish Ministry for Economy and Competitiveness (MINECO) and the European Union (FEDER funds) under grant GRAMM (TIN2017-86727-C2-1-R) .

Submitted to:
PROLE2019

computation that is synchronized by channels; they allow for the execution of dependent tasks on different data items simultaneously. Our research aims at defining the main components of this computational structure. We also analyze the benefits that Dynamic Pipelines offer for the specification of non-blocking solutions to the problems of triangle counting, multiset to set, and connected components.

**Approach:** We formally define the Dynamic Pipeline as a computational model able to specify data-driven algorithms as one-dimensional and unidirectional pipe of stages. The computational structure of a Dynamic Pipeline is flexible and adaptable to the data received as input. Thus, Dynamic Pipeline represents a computational model that enables programs to stretch and shrink according to the spawning and duration of the stages of a program. As it will be illustrated in different examples, these characteristics of Dynamic Pipelines are of paramount relevance for the development of non-blocking solutions that enable the generation of results incrementally.

**Contributions:** We contribute with a novel computational structure of Dynamic Pipelines, as well as with the definition of three well-known problems using where the main features of Dynamic Pipelines, i.e., flexibility and adaptability, are illustrated.

The remainder of this article is structured as follows: section 2 presents a Dynamic Pipeline using the example of duplicate elimination in a multiset; this illustration enables the understanding of the basic components of a Dynamic Pipeline, as well as the program execution method of this Dynamic Pipeline. The section 3 defines the basic components of a Dynamic Pipeline; these definitions are illustrated an non-blocking implementation of the problem of sorting of a sequence. Section 4 illustrates the application of the formalism of Dynamic Pipelines for solving well-known problems. Related work is presented in section 5, and finally, section 6 concludes and gives insights for future work.

## 2 A Dynamic Pipeline for Eliminating Duplicates in a Stream

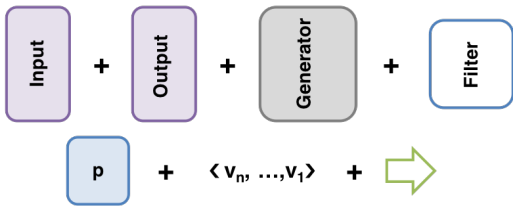
Despite its elegance and potential, it is important to state that structured parallelism still lacks the necessary critical mass to become a mainstream parallel programming technique. Its principal shortcomings are its application space, since it can only address well-defined algorithmic solutions, as well as the lack of a specification to define and exchange skeletons between different implementations. In general, finding a parallel solution stands for finding a way to give to each processor a set of values to work on and then, combine the results to give the final answer following the *Divide & Conquer* paradigm. This computational model enables processors to act on their own data and therefore schedule processes in parallel. This approach is known as a share nothing approach, but in some cases, it requires to replicate values in the distribution of values among processors. MapReduce is a programming model and framework that follows this schema; it maps a function over a given dataset and then, it combines the results. As a framework, MapReduce is utilized to implement MapReduce jobs which encapsulate the features of the model while hiding the complexities inherent in parallelism from users. This framework is, arguably, the largest pattern framework in operation, and has spun off different open source development projects such as Hadoop[6]. Contrary, the Dynamic Pipeline approach takes a different point of view: different processors can act on the same value, but a given value has a single owner at each instant of time [4].

Stream Programming is becoming very popular, and several languages are based on the model of pipelines, fork, and join operators. Its popularity comes from the fact that the programmer describes dependencies among pieces of sequential code/actors. But very few languages have constructions that allow the programmer to express a pipeline having a variable length during program execution. The pattern presented here is linear pipe, i.e., neither forks nor joins; however, it is dynamic, i.e., its length

(number of stages) may depend on input values and is not an external parameter, as for example, in MapReduce. Every stage receives data from a single stage and also produces for another single stage.

## 2.1 Example

In this section, we introduce the main concepts associated with the *Dynamic Pipeline* by means of a very simple example to produce a duplicated-free stream of values from an input stream. In Section 3, a more precise definition of the *Dynamic Pipeline Schema* is presented. A *Dynamic Pipeline (DP)* is a data-driven one-dimensional and unidirectional chain of stages connected by means of channels synchronized by data availability. This computational structure stretches and shrinks depending on the spawning and the lifetime of its stages, respectively. Modeling an algorithmic solution in this framework corresponds to defining a dynamic algorithm to be deployed on this kind of dynamic computational structure. Algorithms must specify four kind of stages: `Input`, `Generator`, `Output` and `Filter` stages. In addition, algorithms must also specify the number and the type of the I/O channels. Each stage has one or more parameterized actors. Channels are unidirectional according to the flow of the data. Before executing an algorithm deployed on a DP, the initial configuration of the pipe must be stated. This is, an initial pipe is set; this is composed of a chain of instantiated stages: input, generator, and output. Figure 1 depicts the different components of a DP and, the generator, and a filter instances that occur when solving a problem.



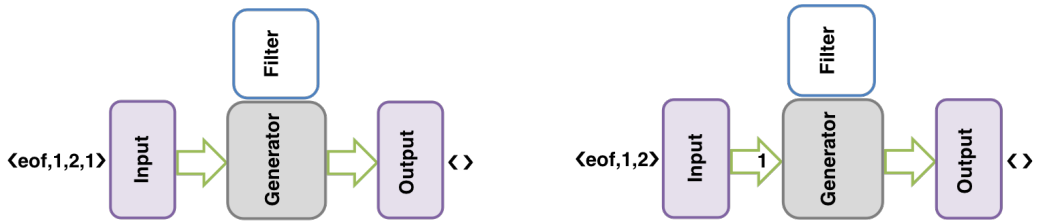

(a) Basic components: `Input`, `Output`, `Generator`, `Filter`, filter parameter (p), I/O stream ($\langle v_n, \ldots, v_1 \rangle$), and input/output channel (arrow)

(b) The `Generator` with parameter `Filter`, channel containing a value v, and a the `Filter` with parameter p

Figure 1: **Dynamic Pipeline Framework.** The `Input` and the `Output` correspond to the interface of the DP, this means that they are in charge of managing the input and output data, respectively. The `Generator` is the responsible of spawning new filters so that it has as a parameter the `Filter`. This is, on receiving an input value spawns a new filter, usually instantiated by the received value. Eventually, the `Generator` can produce an output. In particular, a `Filter` is parameterized by the value of the data, and thus, the filter's parameters are of the type of input data. The `Filter` can have its own memory that is not shared with other filters. Neighbor stages in the pipe communicate with each other via one or several channels. Channels carry sequences of values of the same type, and the output channels of a stage are the corresponding input channels (arrow in Figure 1(a)) for its successor neighbor.

In the rest of this section, Figures 2-5 show to the reader, step by step, a DP solution to the problem of eliminating duplicates from an input stream. Figure 2(a) shows the initial DP used to eliminate duplicates from the input stream $\langle \texttt{eof}, 1, 2, 1 \rangle$. Afterwards, the activation of the computational capacities

programmed in the different stages of the pipe, can be launched by feeding with data the input via the input channels. During the execution of an algorithm, the configuration of the pipe evolves (stretches and shrinks) according to the received input data, as follows: The input receives a value and passes it to its neighbor. If the neighbor of the `Input` is a `Filter`, the value is filtered, and depending on the result of this action, the value is consumed (store in the filter's memory/dropped) or it is passed to the next neighbor of the filter until, possibly, the value in the input arrives to the `Generator`. Figure 2(b) illustrates this state of the DP of our example. In this situation, the generator spawns a new filter instantiated by the received value. The spawned filter is placed before the `Generator` in the chain of stages. That is, the newly created instance of the `Filter` will have as a provider all the channels that were providers to the `Generator`; further, it will be the provider of the `Generator`.

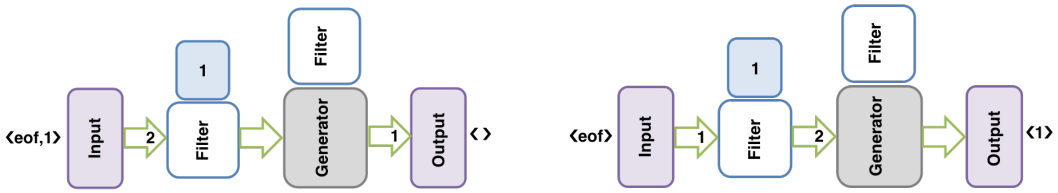



(a) Pipeline initial configuration: `Input`, `Output` and the Generator parameterized by the `Filter`

(b) Pipe configuration. The input set is (1,2) and the value 1 is to be read by the `Generator`

Figure 2: **Example of the DP Configuration.** Initial configuration of the pipe and computation first step

Beside the behavior described above, as the data is processed by the `Filters`, results are produced. Values can be computed by the `Filters` and passed through the `Filters`'output channels until they arrive at the `Output` stage. In the example, the only action of a `Filter` is to check if the received input value is the same as its parameter. If so, the value is dropped and no result is produced. The `Generator` after spawning a new `Filter`, passes the value to the output. Figure 3(a) shows the situation in which the value 1 is passed to the `Output` after the `Generator` has created the `Filter` parameterized by that value in the DP of our example. Figure 3(b) also depicts the value 2 arriving to the `Generator` after passing through the `Filter` associated with the value 1. Additionally, the first output value is left in the output stream.




(a) The `Generator` spawns a `Filter` and produces 1 to `Output`. `Filter` with parameter 1 is reading value 2 and value 1 is ready to be read by `Output`

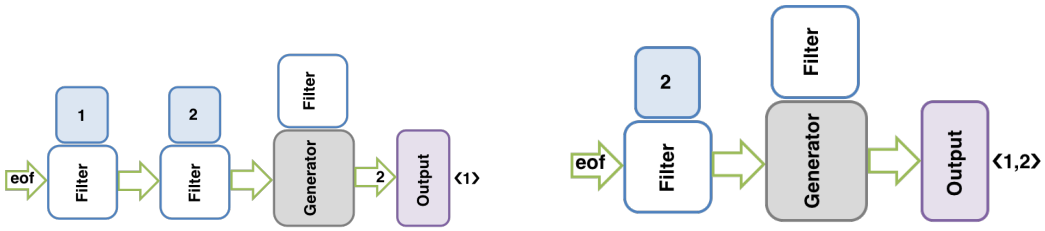
(b) The `Generator` reading 2 spawns a Filter and produces 2 to `Output` and the `Filter` 1 reading 1drops the value

Figure 3: **Dynamic pipeline evolution.** The pipe stretches

In a DP, when any of the stages (i.e., the `Input`, the `Filter`, the `Generator`, and the `Output`)

receives the mark of end of input data (`eof`), then the stage disappears/dies and the pipeline shrinks. In Figure 4, it can be observed the evolution of the DP when the `eof` arrives to the first filter and the DP starts to shrink. Notice that filters die one by one according to the order in which they receive the `eof`. The last stage that dies is the `Output`, and this happens once the last result is produced. Finally, Figure 5 depicts the arriving of the `eof` to the `Generator` and then, to the `Output` stage; further, this figure shows how the result is computed completely. It is worth noting that we have explained a simple configuration of a DP where the filter has a single actor. In the general case, the filter could have several actors like the solution to the problem of counting triangles in a graph [8] using the DP approach. In Section 3, we formally define a DP schema and illustrate the main features of this formalism using the sorting problem. Furthermore, in Section 4, the specifications of solutions a well-known problem using our framework are reported.

Moreover, it is important to notice that the execution of the DP illustrated in Figures 2-5 generates results incrementally. This feature of the Dynamic Pipeline enables the development of flexible data-driven frameworks demanded to address the needs of the next generation of big data technologies.




(a) The end of input produces the elimination of the box *Input*

(b) The end of input produces the elimination of the box *Filter* with parameter 1

Figure 4: **Dynamic pipeline evolution.** The pipe starts to shrink. The first stage to disappear is the `Input`.


(a) Now is time that `Generator` disappears

(b) Now is time to the box `Output` to disappear

Figure 5: **Shrinking a Dynamic Pipeline.** The pipe shrinks until only the result is left.

## 3 Definition of the Components of a Dynamic Pipeline

This section defines the basic components of a *Dynamic Pipeline* (a.k.a. DP). As shown in Section 2.1, the specification of a DP solution requires the formulation of each of the stages of the DP, i.e., the *Input*

(a.k.a. I), *Filter* (a.k.a. F), *Generator* (a.k.a. G), and *Output* (a.k.a. O) stages should be defined. Further, the actors of the filters, the number and type of I/O channels, and the way in which stages are connected via the channels, must be indicated.

**Definition 3.1** *Given the components* $code_{\mathsf{I}}$, $code_{\mathsf{G}}$, *and* $code_{\mathsf{O}}$ *that encode the behavior of the Input, Generator, and Output components, respectively, a Dynamic Pipeline is a tuple* $\mathsf{DP} = (IC, \mathsf{I}, \mathsf{SF}, \mathsf{G}, \mathsf{O}, RC)$*:*

- *IC is the sequence of input channels with corresponding type given by* I*. IC carries the original input data.*
- *RC is a result channel that carries a sequence of output channels with corresponding type given by* O*. RC carries the result of the computation of the* DP*.*
- $\mathsf{I} = (IC, code_{\mathsf{I}}, C)$*, where C is the sequence of output channels with corresponding type given by the filters in* SF*.*
- $\mathsf{G} = (\mathsf{F}, code_{\mathsf{G}}, CT, O)$*, where CT is a tuple of channels and O is the sequence of output channels with corresponding types given by* G*.*
- SF *is a sequence of m triples* $(C_i, \mathsf{F}_i, C_{i+1})$ *such that:*
  - $\mathsf{F}_i$ *is an instance of a filter template* F*.*
  - $C_i, C_{i+1}$ *are tuples of channels with corresponding types given by CT and also the output channels of* I*.*
- $\mathsf{O} = (O, code_{\mathsf{O}}, RC)$

**Example 3.1** *The problem illustrated in this example corresponds to the sorting of a sequence S represented as a sequence of pairs* $(key, value)$*. A dynamic pipeline* $\mathsf{DP}_{\mathsf{sorting}} = (IC, \mathsf{I}, \mathsf{SF}, \mathsf{G}, \mathsf{O}, RC)$ *for sorting the sequence S, is defined in terms of the input and output as follows:*

- *IC is a sequence of one input channel comprising the sequence to be sorted, i.e., the pairs in S.*
- $\mathsf{I} = (IC, code_{\mathsf{I}}, C_1)$ *where* $C_1$ *is a sequence of two channels, both of type of the elements of the sequence being sorted. The component* $code_{\mathsf{I}}$ *implements the behavior of the input channel and is described in example 3.4.*
- SF *is a sequence of instantiated filters, having as input and output channels* $C_1$*.*
- $\mathsf{G} = (\mathsf{F}, code_{\mathsf{G}}, CT, O)$ *where* $CT = C_1$ *and O is a single output channel carrying the pairs in S sorted in ascending order. The component* $code_{\mathsf{G}}$ *encodes the behavior of the generator* G *and is described in example 3.2.*
- $\mathsf{O} = (O, code_{\mathsf{O}}, RC)$*. The component* $code_{\mathsf{O}}$ *encodes the behavior of the output channel; it will be described in example 3.5.*

**Definition 3.2** *A Generator is defined as* $\mathsf{G} = (\mathsf{F}, code_{\mathsf{G}}, CT, O)$*, where*

- F *is a filter template used to create filter instances.*
- $code_{\mathsf{G}}$ *is the behavior of the Generator that includes the condition of when an instance* $\mathsf{F}_i$ *of the filter template is generated, and under which circumstances an output is produced (if any). The created filter instances will be placed between the generator and its provider, i.e., the old generator provider will be the provider of this instance. Further, this instance will be the generator's provider. The component code also encompasses the conditions about when and how the generator will die.*
- *The output channels O carries the values produced by* G*, if any;*

*Note that when a filter is created by a generator, the following arguments need to be provided: a value $p_i$ for the parameter p and an initial value for the filter's memory $M_i$.*

**Example 3.2** *A generator $\mathsf{G} = (\mathsf{F}, code_{\mathsf{G}}, CT, O)$ for $\mathsf{DP}_{\mathsf{sorting}}$ is defined as follows:*

- $\mathsf{F}$ *is the filter instance to be spawned by the generator. This instance will be placed between the generator and its provider.*
- *CT is a pair of channels of pairs $(key, value)$. The first channel carries the pairs of S to be sorted whilst the second carries the result.*
- *The output channels O would carry the values produced by $\mathsf{G}$ but in $\mathsf{DP}_{\mathsf{sorting}}$ it does not produce any output.*
- $code_{\mathsf{G}}$ *implements the behavior of the Generator: An instance $\mathsf{F}_i$ of the filter template $\mathsf{F}$ is created when a $(key, value)$ arrives on the first input channel of the Generator. If the construction of a new filter instance proceeds: The filter's parameter is set to key and the initial value of the filter's memory is set to value. When receiving $eof$, the generator dies after connecting its input channel with its output channel.*

A filter can have its own memory, and this memory is not shared with other filters. Note that *Input* and *Output* define all the channel types in the solution; filters have the same sequence of input and output channels and this sequence coincides with the sequence of *Generator*'s input channel.

**Definition 3.3** *An actor corresponds to a pair, $\mathsf{A} = (code_{\mathsf{A}}, CT_{\mathsf{A}})$, where:*

1. $code_{\mathsf{A}}$ *is a script describing the actor's behavior.*
2. $CT_{\mathsf{A}}$ *is a sequence of channels.*

**Definition 3.4** *A filter template is a 4-tuple $\mathsf{F} = (p, \langle \mathsf{A}_1, \ldots, \mathsf{A}_m \rangle, CT, M)$ where:*

- *p is the filter parameter;*
- $\langle \mathsf{A}_1, \ldots, \mathsf{A}_m \rangle$ *is a stack of actors*
- *CT is a tuple of channels*
- *M stands for the memory of* $\mathsf{F}$

An instance of a filter template $\mathsf{F}$ gives values to the parameters of the filter template; it is defined as follows:

**Definition 3.5** *Given an filter template $\mathsf{F}$, an instance $\mathsf{F}_i$ of $\mathsf{F}$ is a 4-tuple: $\mathsf{F}_i = (p_i, \langle \mathsf{A}_1, \ldots, \mathsf{A}_m \rangle, CT, M_i)$, where:*

- $p_i$ *is an instance of the parameter of* $\mathsf{F}$
- $\langle \mathsf{A}_1, \ldots, \mathsf{A}_m \rangle$ *is a stack of actors*
- *CT corresponds to the concatenation of the channel $CT_{\mathsf{A}_j}$ of the actors $\mathsf{A}_j = (code_{\mathsf{A}_j}, CT_{\mathsf{A}_j})$ in* $\mathsf{F}$
- $M_i$ *stands for the memory of the filter instance* $\mathsf{F}_i$

*For each of the actor templates, $\mathsf{A}_j$, the behavior expressed by the component $code_{\mathsf{A}_j}$, must be defined as follows:*

- *action upon an input value on each input defined in $CT_{\mathsf{A}_j}$*
- *transformations on memory $M_i$*

- *output values on each output*
- *action upon receiving an end of input on each input*

*The filter instance dies whenever the actor's stack is empty.*

**Example 3.3** *Considering the* $\mathsf{DP}_{\mathsf{sorting}}$*, an instance filter* $\mathsf{F}_i = (p_i, \langle \mathsf{A} \rangle, CT, M_i)$ *is defined as follows:* $CT = \langle C_1 \rangle$ *is a sequence of one channels of pairs* $(key, value)$*. The type of the parameter* $p_i$ *is the same as the type of the key and* $M_i$ *is a particular value. For the only actor* $\mathsf{A}$*, its behavior, i.e.,* $code_{\mathsf{A}}$ *is defined as follows:*

- *Given the sequence of channels and the pair* $(key, value)$ *read from the input* $C_1$*, the key is compared to the filter's parameter* $p_i$*. If* $key > p_i$*, the actor outputs* $(key, value)$ *on channel* $C_1$*. Otherwise, it produces a new pair* $(p_i, M_i)$ *and outputs this pair on* $C_1$ *and sets* $p_i = key$ *and* $M_i = value$*.*
- *When receiving* $eof$ *on* $C_1$ *the actor passes the pair* $(p_i, M_i)$ *on channel* $C_1$ *and deactivates itself.*

**Definition 3.6** *An Input is defined as a 3-tuple* $\mathsf{I} = (IC, code_{\mathsf{I}}, C_1)$*.*

- *IC stands for a sequence of channels of the same type as the pairs* $(key, value)$ *in sequence S.*
- $C_1$ *is a sequence of channels of the same type as the input channels of the Filter and the Generator.*
- $code_{\mathsf{I}}$ *corresponds to the behavior of the Input. This component implements the transformation of the source input data into data for the input channels of the Filter/Generator.*

**Example 3.4** *For the sorting example,* $\mathsf{I} = (IC, code_{\mathsf{I}}, C_1)$ *is defined as follows: IC is a sequence of one channel of type as the elements in the input sequence of* $\mathsf{DP}_{\mathsf{sorting}}$*. The component* $C_1$ *is a sequence of one channel of same type as the pair* $(key, value)$*. The component* $code_{\mathsf{I}}$ *applies the identity transformation to the input data and passes the input pairs to* $C_1$*. When the eof appears in IC,* $code_{\mathsf{I}}$ *outputs* $eof$ *on* $C_1$ *and dies.*

**Definition 3.7** *An Output is defined as a 3-tuple* $\mathsf{O} = (O, code_{\mathsf{O}}, RC)$*.*

- *O stands for a sequence of channels of the same type as the output channels of the Generator.*
- *RC is a sequence of channels of the same type as the output data.*
- $code_{\mathsf{O}}$ *corresponds to the behavior of the Output. This component implements the transformation of the pairs* $(key, value)$ *into data for the output channels of the Filter/Generator.RC*

**Example 3.5** *For the case of sorting a sequence* $\mathsf{O} = (O, code_{\mathsf{O}}, RC)$ *is defined as follows: O corresponds to a channel comprising pairs* $(key, value)$*. The component* $code_{\mathsf{O}}$ *transforms the pairs in O into the type require by the channel RC. The component RC carries the output of the* $\mathsf{DP}_{\mathsf{sorting}}$*.*

# 4 Example of using a DP: Connected Components of a Graph

We show the implementation of the problem of finding the connected components in a graph using a Dynamic Pipeline. A connected component of an undirected graph is a subgraph in which any two vertices are connected each other by paths. Finding connected components of an undirected graph in our case is to obtain the minimal partition of the set of nodes, such that nodes in one set are connected i.e., there is a path between each pair of nodes. The DP input data is a sequence of edges, pairs of the form $(a, b)$ where $a$ and $b$ are nodes of the graph. The output of the DP is a sequence of sets of nodes. Each set of nodes in the output corresponds to nodes of a connected component. The sequence is a partition of

the set of nodes of the input graph[1]. The underlying idea of this algorithm is that each filter maintains a connected component of the graph. Thus, if we take a snapshot during the execution, the filters in the DP will correspond to a (partial) partition of the input graph according to the relationship *connected* over the set of nodes. The dynamic pipeline $\mathsf{DP}_{\mathsf{cc}} = (IC, \mathsf{I}, \mathsf{SF}, \mathsf{G}, \mathsf{O}, RC)$ for solving the problem of identifying connected components is described as follows:

### 4.1 Input

An *Input* component $\mathsf{I} = (IC, code_{\mathsf{I}}, C_1)$ is defined as follows:

- $IC = \langle C_e \rangle$ is a sequence of only one channels $C_e$ that comprises the edges in the input data.
- $C_1 = \langle C_{1_e}, C_{1_s} \rangle$ is a sequence of two channels. $C_{1_e}$ is a channel of edges and $C_{1_s}$ of sets of nodes.
- $code_{\mathsf{I}}$ corresponds to the behavior of the input. It receives the input sequence of edges, applies the identity transformation to it and outputs these data on $C_{1_e}$. When receiving eof on C1 outputs $eof$ on both outputs, $C_{1_e}$ and $C_{1_s}$, and dies.

### 4.2 The Generator

A generator $\mathsf{G} = (\mathsf{F}, code_{\mathsf{G}}, CT_{\mathsf{G}}, O_{\mathsf{G}})$ is defined as follows.

1. The sequence $CT_{\mathsf{G}}$ is composed by two channels, $\langle CT_e, CT_s \rangle$. The first input channel $CT_e$ comprises a sequence of edges, i.e., pairs of nodes. The second input channel $CT_s$ comprises a sequence of set of nodes having the property of being connected.
2. The sequence $O_{\mathsf{G}} = \langle O_s \rangle$. The channel $O_s$ is a sequence of set of vertices having the property of being connected.
3. The behavior, given by $code_{\mathsf{G}}$ is the following: If an edge $e$=$(a,b)$ arrives to first channel of the *Generator*, then $e$ is not a member of any of the connected components previously identified. Therefore, $e$ is a new connected component and hence, a new instance of the filter is spawned. The memory of this new instance of the filter will hold all the vertices of the identified connected component, i.e., the set of nodes $\{a,b\}$[2].When an $eof$ arrives on the first input channel $CT_e$, the generator $\mathsf{G}$ connects its input channels $CT_s$ to $O_s$ and then, it dies. No special parameter is required by the filters.

### 4.3 Filter

A Filter $\mathsf{F}_i = (\emptyset, \langle \mathsf{A}_1, \mathsf{A}_2 \rangle, CT, M_i)$ for the problem of connected components is as follows:

- There is no parameter;
- $CT$ corresponds to the concatenation of the sequence of channels $CT_{\mathsf{A}_1}$ and $CT_{\mathsf{A}_2}$ of the actors $\mathsf{A}_1$ and $\mathsf{A}_2$
  $CT_{\mathsf{A}_1} = \langle C_{\mathsf{A}_1} \rangle$ and
  $CT_{\mathsf{A}_2} = \langle C_{\mathsf{A}_2} \rangle$
- $\langle \mathsf{A}_1, \mathsf{A}_2 \rangle$ is a stack of actors and $\mathsf{A}_1$ is on the top of this stack.

[1] To effectively compute the subgraphs, an additional process is required

[2] Only vertices are recorded, thus, for identifying a connected sub-graph an additional process is required

1. Find connected components $\mathsf{A}_1 = (code_{\mathsf{A}_1}, CT_{\mathsf{A}_1})$.
   The channel $C_{\mathsf{A}_1}$ carries edges. If the edge in $C_{\mathsf{A}_1}$ is incident to any node present in the memory, add the other node (if not already) to the memory $M_i$. Otherwise, passes this edge to its neighbor in $\mathsf{DP}_{\mathsf{cc}}$ through $C_{\mathsf{A}_1}$. When receiving an $eof$ on $C_{\mathsf{A}_1}$, the actor $\mathsf{A}_1$ passes this mark to its neighbor in $\mathsf{DP}_{\mathsf{cc}}$ through $C_{\mathsf{A}_1}$ and deactivates itself. Now the second actor $\mathsf{A}_2$ is on the top of the stack of actors.
2. Try to enlarge connected components $\mathsf{A}_2 = (code_{\mathsf{A}_2}, CT_{\mathsf{A}_2})$.
   The channel $C_{\mathsf{A}_2}$ carries sets of nodes representing connected components. If the set of nodes in $C_{\mathsf{A}_2}$ intersects the set in $M_i$, the memory $M_i$ is updated with the union of these two sets of nodes and no output is produced. Otherwise, the actor $\mathsf{A}_2$ passes this set of nodes to its neighbor. When receiving the $eof$, the actor A2 outputs the set of nodes in $M_i$, connects its input channel to its output channel and deactivates itself. Hence the stack of actors of $\mathsf{F}_i$ becomes empty and it dies.

### 4.4 Output

An *Output* component $\mathsf{O} = (O, code_{\mathsf{O}}, RC_{\mathsf{O}})$ is defined as follows:

- $O = \langle O_s \rangle$, where the channel $O_s$ comprises sets nodes (i.e., the connected components).
- $RC = \langle RC_s \rangle$ is a sequence of one channel of sets of nodes.
- $code_{\mathsf{O}}$ applies the identity transformation to the data on $O_s$ and outputs them on $RC_s$. When receiving an $eof$ on the $O_s$, it dies.

## 5 Related Work

In the literature, the notion of pipeline is the construction of a pipe, i.e., a sequence of sequential processes, communicating only with its neighbors. In [7], an analytical model for pipeline parallelism based on the queuing theory is developed. In this work Navarro et al. claim that they model is useful for the design of new pipeline algorithms. They identified two strong limitations: First, the ability of specifying stages that produce a different number of outputs and second, the possibility of arriving to an I/O bottleneck. These issues are overcome by the DP, due to the fact that most of the stages have a similar behavior. Benoit et al. [2] present a very interesting survey on the technique called *pipelined workflow scheduling*. The survey focuses son the scheduling of applications that continuously process a stream of datasets using a given workflow (i.e., a fixed net of channels without loops). This kind of pipelines lacks of the possibility of evolving according to the incoming data. Lee et al. in [5] present a programming model where pipelines are not static (i.e., pipelines are not specified *a priori*). Instead, the structure of the pipeline emerges (on-the-fly pipelines) as the program executes. The difference with Dynamic Pipelines is that on-the-fly pipelines allow for the use of fork and join operations, stages need not be similar and thus, the pipeline's topologies could be non-linear. DP is closer to MapReduce, in the sense is a simple schema in which very few elements are necessary to describe a solution.

## 6 Concluding Remarks

In this paper, the problem of flexible parallel computation is tackled and the Dynamic Pipeline is presented as a novel computational structure to address this problem. A Dynamic Pipeline enables the

specification of data-driven algorithms that follow a one-dimensional and unidirectional chain of stages. The computational method employed by the Dynamic Pipeline enables to define algorithms capable of stretching and shrinking stages in the pipe according to the conditions of the input and the lifetime of the stages. Moreover, the DP-based solutions to the well-known problems of multiset to set, sorting, and connected components, provided evidence of the flexibility and adaptability of the proposed computational structure. Thus, our work broadens the repertoire of computational frameworks for structured parallelism that are suitable for the implementation of data-driven algorithms. We hope that our proposed computational structure enables the definition of more flexible and non-blocking solutions where results are output incrementally, as soon as, they are generated by the Dynamic Pipeline. In the future, we will plan to formalize the semantics of the computational method of the Dynamic Pipeline, as well as to state the main properties of this formalism. Furthermore, the implementation and empirical evaluation of well-known data-driven problems are part of our future agenda.

## References

[1] Mikhail J. Atallah & Marina Blanton, editors (2010): *Algorithms and Theory of Computation Handbook: General Concepts and Techniques*, 2 edition. Chapman & Hall/CRC.

[2] Anne Benoit, Ümit V Çatalyürek, Yves Robert & Erik Saule (2013): *A survey of pipelined workflow scheduling: Models and algorithms*. *ACM Computing Surveys (CSUR)* 45(4), p. 50.

[3] Michael I Gordon, William Thies & Saman Amarasinghe (2006): *Exploiting coarse-grained task, data, and pipeline parallelism in stream programs*. *ACM SIGOPS Operating Systems Review* 40(5), pp. 151–162.

[4] Waqar Hasan & Rajeev Motwani (1994): *Optimization Algorithms for Exploiting the Parallelism-Communication Tradeoff in Pipelined Parallelism*. In Jorge B. Bocca, Matthias Jarke & Carlo Zaniolo, editors: *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, Morgan Kaufmann, pp. 36–47. Available at `http://www.vldb.org/conf/1994/P036.PDF`.

[5] I Lee, Ting Angelina, Charles E Leiserson, Tao B Schardl, Zhunping Zhang & Jim Sukha (2015): *On-the-fly pipeline parallelism*. *ACM Transactions on Parallel Computing* 2(3), p. 17.

[6] Donald Miner & Adam Shook (2013): *MapReduce Design Patterns : [building effective algorithms and analytics for Hadoop and other systems]*. O'Reilly, Beijing, Kln, u.a. Available at `http://opac.inria.fr/record=b1134500`. DEBSZ.

[7] Angeles Navarro, Rafael Asenjo, Siham Tabik & Calin Cascaval (2009): *Analytical modeling of pipeline parallelism*. In: *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*, IEEE, pp. 281–290.

[8] Ana Edelmira Pasarella Sánchez, Maria-Esther Vidal & Ana Cristina Zoltan Torres (2017): *Comparing MapReduce and pipeline implementations for counting triangles*. *Electronic proceedings in theoretical computer science* 237, pp. 20–33.