Author's copy (postprint) of C. Misale, M. Drocco, G. Tremblay, A.R. Martinelli, M. Aldinucci, PiCo: High-performance data analytics pipelines in modern C++, Future Generation Computer Systems (2018), https://doi.org/10.1016/j.future.2018.05.030

PiCo: High-Performance Data Analytics Pipelines in Modern C++

Claudia Misale[a], Maurizio Drocco[b,*], Guy Tremblay[c], Alberto R. Martinelli[b], Marco Aldinucci[b]

[a] *Cognitive and Cloud, Data-Centric Solutions, IBM T.J. Watson Research Center. Yorktown Heights, New York, USA*
[b] *Computer Science Department, University of Torino. Torino, Italy*
[c] *Département d'Informatique, Université du Québec à Montréal. Montréal, QC, Canada*

Abstract

In this paper, we present a new C++ API with a fluent interface called PiCo (Pipeline Composition). PiCo's programming model aims at making easier the programming of data analytics applications while preserving or enhancing their performance. This is attained through three key design choices: 1) unifying batch and stream data access models, 2) decoupling processing from data layout, and 3) exploiting a stream-oriented, scalable, efficient C++11 runtime system. PiCo proposes a programming model based on pipelines and operators that are polymorphic with respect to data types in the sense that it is possible to reuse the same algorithms and pipelines on different data models (e.g., streams, lists, sets, etc.). Preliminary results show that PiCo, when compared to Spark and Flink, can attain better performances in terms of execution times and can hugely improve memory utilization, both for batch and stream processing.

Keywords: Big Data, High Performance Data Analytics, Domain Specific Language, C++, Stream Computing, Fog Computing, Edge Computing.

1. Introduction

The importance of Big Data has been strongly assessed in the last years, and their role is crucial to companies institutions, and research centres, which keep incorporating their analysis into their strategic vision, using them to make better and faster decisions. In the next years, most of those Big Data will be produced by the *Internet of Things* (IoT). By 2020, Cisco expects 50 billion of connected devices [1] with an average of almost 7 per person. According to the current *IoT+Cloud* paradigm for Big Data Analytics (BDA), data sensed from distributed devices ought to be stored in cloud data centres to be analysed. Whilst data-processing speeds have increased rapidly, bandwidth to carry data to/from data centres has not increased equally fast [2]. For this, supporting the transfer of data from billions of IoT devices to cloud to be analysed is becoming increasingly unrealistic due to the volume and geo-distribution of those devices. Moreover, the need to reduce latency to enable applications to react promptly to events is impelling, and it is common to push computation and storage closer to where data is continuously generated [3].

Fog (or *Edge*) computing actually aims at exploiting many distributed edge nodes (e.g., routers, set-top boxes, mobile devices, micro data centres) to selectively support distributed latency- or bandwidth-sensitive applications also implementing near-data analytics [3]. Fog configures as a powerful enabling complement to the *IoT+Cloud* scenario, featuring a new layer of cooperating devices that can run services and analytic tasks, independently from and contiguously with existing Cloud systems [4].

Distributing BDA pipelines into Fog systems poses new challenges, stemming from the dynamic and open nature of these systems. For these systems we envision the following desiderata:

1. The ability to process partial data, either a batch's partition or a stream's time window. Neither edge devices nor the Cloud will have a global vision of the whole data (no data lake).
2. The ability to switch from stream to batch data model (and vice versa), as well as reusing code and mixing up the two models in large systems. Data will be subject to different degrees of aggregation or coalescing depending on the system condition.
3. The capacity to deploy on heterogeneous computing devices, possibly low-power and with a lightweight framework stack.
4. The availability of a model for users with a clear and compositional semantics. Compositionality at the language level should be reflected at run-time level. An update in the BDA pipeline should not require to recompile and redeploy the whole system.

From the programmability perspective, a common aim in

*Corresponding author
Email addresses: c.misale@ibm.com (Claudia Misale), drocco@di.unito.it (Maurizio Drocco), tremblay.guy@uqam.ca (Guy Tremblay), alberto.martinelli@edu.unito.it (Alberto R. Martinelli), aldinuc@di.unito.it (Marco Aldinucci)

BDA tools is to allow ease of programming by providing a programming model independent from the *data model*, resulting in uniform interfaces for *batch* and *stream* processing. For instance, such uniformity is the cornerstone of Google Dataflow [5]. Similarly, modifying a batch-oriented Spark [6] program to perform stream processing amounts to wrapping the existing code with stream-oriented constructs.

In this context, we advocate PiCo (**Pi**peline **Co**mposition), a new C++ framework with a fluent API, based on method chaining [7], designed over a conceptual framework of layered Dataflow. PiCo's programming model aims at *facilitating the programming* and *enhancing the performance* of BDA through a unified batch and stream data model (cf. desideratum 2), coupled with a stream-oriented, scalable, efficient C++11 run-time system (cf. desideratum 3).

Moreover, PiCo stands on a *functional abstract model*, mapping each program to its unambiguous semantics (cf. desideratum 4). This was an explicit design choice, that distinguishes PiCo from most mainstream BDA tools. For instance, Spark, Storm [8], and Flink [9] typically require specializing the algorithm to match the data access and layout. Specifically, data transformation functions exhibit different functional types when accessing data in different ways. For this reason, the source code must often be revised when switching from one data model to another. Some of them, such as Spark, provide the runtime with a module to convert streams into micro-batches (Spark Streaming, a library running on Spark core), but still, different code needs to be written at the user-level.

PiCo is designed as a headers-only C++11 framework and supports headers-only analytics applications. This means that a PiCo application can be compiled to any target platform supporting a modern C++ compiler. Since the PiCo run-time exhibits no dependencies from third-party libraries, this makes it easy to port PiCo applications, and even deploy them on specialised hardware platforms and appliances, e.g., to run high-frequency stream analytics for edge computing. A PiCo pipeline could also be directly compiled into a FPGA bitstream.

PiCo has been designed under a philosophy that can be summarized as "high performance at low cost," based on a lightweight C++ runtime, in which data collections are *streamed* among computational nodes to minimize the memory footprint. In particular, PiCo relies on FastFlow [10], a parallel programming framework designed to support streaming applications on cache-coherent multicore platforms. As confirmed by the results presented below, PiCo steadily exhibits low memory consumption with no compromises on the performance side.

The present paper is an extension of a previous workshop paper [11]. In particular, the PiCo API has been extended to include *iterative* and *binary* pipelines, making it possible to implement join-based applications like Page Rank. The experimental evaluation has also been enriched with the use of larger batch and stream datasets, along with Page Rank. Additionally, the PiCo abstract model is now presented—viz., formal syntax and semantics.

This paper thus proceeds as follows. In Section 2, we provide a brief survey over the state-of-the-art data analytics frameworks. In Sections 3 and 4, we introduce the abstract programming model and its current implementation, thus discussing the C++ API exposed by PiCo. In Section 5, we provide some experimental evaluation. Finally, we conclude the paper in Section 6 and present some future work.

2. Related Work

In recent years, a plethora of BDA tools has been proposed. Although each tool claims to provide better programming, data, and execution models—for which only informal semantics are generally provided—all share some characteristics at different levels [12].

Apache Spark design is intended to address iterative computations by reusing the working dataset by keeping it in memory [13, 6]. For this reason, Spark represents a landmark in Big Data tools history, having a strong success in the community. The overall framework and parallel computing model of Spark is similar to MapReduce, while the innovation is in the data model, represented by the *Resilient Distributed Dataset* (RDD). An RDD is a read-only collection of objects partitioned across a cluster of computers that can be operated on in parallel. A Spark program can be characterized by the two kinds of operations applicable to RDDs: *transformations* and *actions*. Those transformations and actions compose the directed acyclic graph (DAG) representing the application. For stream processing, Spark implements an extension through the Spark Streaming module, providing a high-level abstraction called *discretized stream* or *DStream* [13]. Such streams represent results in continuous sequences of RDDs of the same type, called *micro-batches*. Operations over DStreams are "forwarded" to each RDD in the DStream, thus the semantics of operations over streams is defined in terms of batch processing according to the following simple translation: Given a DStream $a = [a_1, a_2, \ldots]$—i.e., a possibly unbounded ordered sequence of items, where each item a_i is a micro-batch of type RDD—then $\mathtt{op}(a) = [\mathtt{op}(a_1), \mathtt{op}(a_2), \ldots]$. All RDDs in a DStream are processed in order, whereas data items inside an RDD are processed in parallel without any ordering guarantees.

Formerly known as Stratosphere [14], *Apache Flink* [9] focuses on stream programming. The abstraction used is the *DataStream*, a representation of a stream as a single object. Operations are composed (i.e., pipelined) by calling operators on DataStream objects. Flink also provides

the *DataSet* type for batch applications, that identifies a single immutable multi-set—a stream of one element. A Flink program, either for stream or batch processing, is a term from an algebra of operators over DataStreams or DataSets, respectively. Flink, differently from Spark, is a stream processing framework, meaning that both batch and stream processing are based on a streaming runtime. It can be considered one of the most advanced stream processors as many of its core features were already considered in the initial design [9].

Apache Storm is a framework targeting only stream processing [8]. It is perhaps the first widely used large-scale stream processing framework in the open source world. Storm's programming model is based on three key notions: *Spouts*, *Bolts*, and *Topologies*. A Spout is a source of a stream, that is (typically) connected to a data source or that can generate its own stream. A Bolt is a processing element, so it processes any number of input streams and produces any number of new output streams. A topology is a composition of Spout and Bolts.

Google Dataflow SDK [5] is part of the Google Cloud Platform. Here, the term "Dataflow" refers to the "Dataflow model," to describe the processing and programming model of the Cloud Platform. This framework aims at providing a unified model for stream, batch, and micro-batch processing. The base entity is the *Pipeline*, representing a data processing job consisting of a set of operations that can read a source of input data, transform that data, and write out the resulting output. The data model in Google Dataflow is represented by *PCollection*s, representing potentially large, immutable bags of elements, that can be either bounded or unbounded. The bounded (or unbounded) nature of a PCollection affects how Dataflow processes the data. Bounded PCollections can be processed using batch jobs, that might read the entire data set once and perform processing in a finite job. Unbounded PCollections must be processed using streaming jobs, as the entire collection may never be available for processing at any one time, and they can be grouped by using windowing to create logical windows of finite size.

Thrill [15] is a prototype of a general purpose big data batch processing framework with a dataflow style programming interface implemented in C++ and exploiting template meta-programming. Thrill's data model is the *Distributed Immutable Array* (DIA), an array of items distributed over the cluster, to which no direct access to elements is permitted—i.e., it is only possible to apply operations to the array as a whole. A DIA remains an abstract entity flowing between two concrete DIA operations, allowing to apply optimizations such as pipelining or chaining, combining the logic of multiple functions into a single one (called pipeline). A consequence of using C++ is that memory has to be managed explicitly, although memory management in modern C++11 has been considerably simplified—for instance, Thrill uses reference counting extensively. Thrill provides a SPMD (Single Program, Multiple Data) execution model, similar to MPI, where the same program is run on different machines.

StreamBox [16] is another recent stream processing engine, also in C++, specifically designed to exploit the parallelism of multicore machines. StreamBox executes a pipeline of transforms over records, expressed using a model similar to Google Dataflow (viz., `PCollection`, `ParDo`, etc.). Records can be grouped into epochs delineated by watermarks. To enhance parallelism, records as well as epochs may be processed out-of-order. However, all records within an epoch are ensured to be consumed before the watermarks, thus ensuring an appropriate ordering between epochs.

Despite the fact that the design of both the C++ API and the underlying abstract model focuses on minimality (*à la* RISC), PiCo is expressive enough to almost subsume all the mentioned frameworks. Thus, starting from a program expressed in one of those tools' API, any function can easily be mapped to PiCo functions, yielding a PiCo Pipeline with a similar but more abstract semantics.

From the implementation perspective, along with the same line as StreamBox, PiCo is based on a C++ stream-based runtime, thus we expect PiCo and StreamBox to exhibit similar performance. Unfortunately, it was not possible to deploy StreamBox on our testing platform, therefore an effective comparison has to be relegated to future work.

3. PiCo Abstract Model

In this section, we present the abstract model underlying PiCo [17, 18]. We remark that providing the possibility of reasoning about programs in abstract—rather than operational—terms is a primary goal of PiCo, in particular with respect to the Fog context, where compositionality is a key requirement (cf. desideratum 4).

The building blocks of the PiCo syntax are *Pipelines* (Sect. 3.4), that are composed by *Operators* (Sect. 3.3) according to a *type system* supporting partial polymorphism. Conversely, *Collections* (Sect. 3.2) are considered only at the semantic level, where a PiCo Pipeline is mapped to a *Dataflow graph*, with tokens, nodes, and edges representing, respectively, Collections, Operators, and data dependencies. But first, we present a brief overview of dataflow concepts.

3.1. The Dataflow Model

Dataflow is a model of computation where a program is described as a set of concurrent processes—aka. *actors*—communicating with each other by sending/receiving data *tokens* through channels [19].

A set of *firing rules* is associated with each actor. Processing then consists of "repeated firings of actors" based on the availability of tokens. The Dataflow model is inherently parallel as all actors whose data tokens are available can be fired simultaneously.

More precisely, a Dataflow actor consumes input tokens when it *fires* and then produces output tokens. The function mapping input to output tokens is called the *kernel*. Typically, kernels are *purely functional*, meaning that firings have no side effects and that output tokens are pure functions of the input tokens. However, the model can be extended to allow stateful actors. As for output tokens, they can be replicated and placed onto all output channels (i.e., broadcasting) or sent to specific channels (e.g., switch, scatter). Extensions to the basic model also allow to express arbitrary stream consuming policies (e.g., gathering from any channel).

3.2. Collections

In PiCo, collections are semantic entities that flow across semantic graphs, in the form of tokens. A collection is either *bounded* or *unbounded*; moreover, it is also either *ordered* or *unordered*. A combination of the mentioned characteristics defines the *structure type* of a collection. We refer to the key structure types with a mnemonic name:[1]

- a bounded, ordered collection is a *list*;
- a bounded, unordered collection is a (bounded) *bag*;
- an unbounded, ordered collection is a *stream*.

A collection type is characterized by a structure type and a *data type*, namely the type of the collection elements. Formally, a collection type has form T_σ where $\sigma \in \Sigma$ is the structure type, T is the data type, and where $\Sigma = \{\text{bag}, \text{list}, \text{stream}\}$ is the set of all structure types. We also partition Σ into Σ_b and Σ_u, defined as the sets of bounded and unbounded structure types. Moreover, we define Σ_o as the set of ordered structure types, thus $\Sigma_b \cap \Sigma_o = \{\text{list}\}$ and $\Sigma_u \cap \Sigma_o = \{\text{stream}\}$. Finally, we allow the void type $\emptyset$.

At the semantic level, collection are Dataflow tokens. Unordered collections are mapped to multi-sets, whereas ordered collections are mapped to sequences, in which each item is associated with a numeric *timestamp*, representing its temporal coordinate. In the following, for the sake of simplicity, we restrict the presentation to unordered, bounded collections (i.e., multi-sets), although the model also covers the other collection types [17].

[1] We do not deal with unbounded, unordered collections: unbounded collections typically represent *data streams*, which are *sequences* of items, thus are ordered.

3.3. Operators

Operators are the building blocks composing a Pipeline. They are categorized according to the following grammar of core operator families:

⟨*core-operator*⟩ ::= ⟨*core-unary-operator*⟩
| ⟨*core-binary-operator*⟩

⟨*core-unary-operator*⟩ ::= ⟨*map* ⟩ | ⟨*combine*⟩
| ⟨*emit*⟩ | ⟨*collect*⟩

⟨*core-binary-operator*⟩ ::= ⟨*b-map* ⟩

In addition to core operators, generalized operators can *decompose* their input collections by:

- partitioning the input collection according to a user-defined grouping policy (e.g., group by key);
- windowing the *ordered* input collection according to a user-defined windowing policy (e.g., sliding windows).

The complete grammar of operators follows:

⟨*operator*⟩ ::= ⟨*core-operator*⟩
| ⟨*w-operator*⟩ | ⟨*p-operator*⟩ | ⟨*w-p-operator*⟩

where `w-` and `p-` denote decomposition by windowing and partitioning.

Operator types have form $T_\sigma \to U_\sigma$, where T_σ and U_σ are collection types. All operators are polymorphic with respect to data types. Moreover, all operators but `emit` and `collect` are polymorphic with respect to structure types. Conversely, each `emit` and `collect` operator deals with one specific structure type. For example, an `emit` for a finite text file (i.e., a `from-file` Operator) would generate a bounded collection of strings, whereas an `emit` for a stream of tweets would generate an unbounded collection of tweet objects. The depicted polymorphic type system is the essence of the unified processing model for batch and streaming, that we identified as a primary requirement for the Fog context (cf. desideratum 2).

3.3.1. Data-Parallel Operators

Operators in the `map` family process data collections on a per-element basis, according to a *kernel* function. They are defined by the following grammar:

⟨*map* ⟩ ::= `map` f | `flatmap` f

where f is the *kernel* function. The former produces exactly one output element from each input element, whereas the latter produces a (possibly empty) bounded set of output elements for each input element and the output collection is the merging of the output sets. Formally:

$$\texttt{map}\ f\ m = \{f(m_i) \bullet m_i \in m\}$$
$$\texttt{flatmap}\ f\ m = \bigcup \{f(m_i) \bullet m_i \in m\}$$

Operators in the `map` family have type $T_\sigma \to U_\sigma$, for $\sigma \in \Sigma$.

Operators in the `combine` family synthesize the elements from an input collection into a single value. They are defined by the following grammar:

$\langle$ *combine* $\rangle$ ::= `reduce` $\oplus$ | `fold+reduce` $\oplus_1$ z $\oplus_2$

The former corresponds to a classical reduction with a binary operator, whereas the latter is a two-phase aggregation that uses $\oplus_2$ to reduce a set of partial accumulated states obtained using z and $\oplus_1$.[2] Formally, for the `reduce` operator—we do not discuss further the `fold+reduce` for the sake of simplicity:

$$\texttt{reduce}\ \oplus\ m = \left\{\bigoplus \{m_i \in m\}\right\}$$

Operators in the `combine` family deal only with bounded collections, thus they have type $T_\sigma \to U_\sigma$, for $\sigma \in \Sigma_b$.

3.3.2. Pairing Operators

Operators in the `b-map` family are intended to be the binary counterparts of `map` operators. They are defined by the following grammar:

$\langle$ *b-map* $\rangle$ ::= `zip-map` f | `join-map` f
| `zip-flatmap` f | `join-flatmap` f

The binary kernel f takes as input pairs of elements, one from each of the input collections. Variants `zip-` and `join-` correspond to the following pairing policies, with some restrictions:

- Zipping is restricted to ordered collections and produces the pairs of elements with the same position within the order of respective collections;
- Joining is restricted to bounded collections and produces the Cartesian product of the input collections.

In the following, since our presentation is restricted to unordered collections, we only consider `join-` operators, that have type $T_\sigma \times {T'}_\sigma \to U_\sigma$, for $\sigma \in \Sigma_b$.

The `join-map` and `join-flatmap` operators have the following semantics, where m_1 and m_2 are input multi-sets:

$$\texttt{join-map}\ f\ (m_1, m_2) = \{f(x,y) \bullet (x,y) \in m_1 \times m_2\}$$
$$\texttt{join-flatmap}\ f\ (m_1, m_2) = \bigcup \{f(x,y) \bullet (x,y) \in m_1 \times m_2\}$$

3.3.3. Windowing Operators

Windowing operators are defined according to the following grammar, where ω is the windowing policy:

$\langle$ *w-operator* $\rangle$ ::= `w-`$\langle$ *core-operator* $\rangle$ ω

In semantic terms, a windowing operator takes an ordered collection, produces a collection (with the same structure type as the input one) of windows (i.e., lists), according to the windowing policy ω, and applies the subsequent operation to each window.

We also rely on windowing to extend bounded operators[3] and have them deal with unbounded collections; for instance, given a (bounded) windowing `combine` operator *op*, the semantics of its unbounded variant is a direct extension of the bounded one. Formally, using + as an infix operator for list concatenation, where $s^{(\omega)}$ denotes the result of windowing the stream s according to policy ω:

$$\texttt{w-}op\ \omega\ s = op\ s_0^{(\omega)} + \ldots + \ op\ s_i^{(\omega)} + \ldots$$

For the sake of simplicity, we omit some parts of the formalization, including windowed collections and unbounded `map` operators. We refer to the complete PiCo formalization [17] for further details.

3.3.4. Partitioning Operators

Partitioning operators are defined according to the following grammar, where π is the partitioning policy:

$\langle$ *p-operator* $\rangle$ ::= `p-`$\langle$ *core-operator* $\rangle$ π

Operators in the `combine` and `b-map` families support partitioning, so, for instance, a `p-combine` produces a multi-set, in which each value is the synthesis of one group; also the natural join operator from the relational algebra is a particular case of per-group joining.

In semantic terms, a partitioning policy $\pi : T \to K$ defines how to group the elements from a T-typed multi-set m, such that each element m_i is mapped to a sub-collection, depending on the corresponding key $\pi(m_i) \in K$. The sub-collection containing all the elements from m mapped to k by π is denoted by $\sigma_k^\pi(m)$, whereas the collection containing all such sub-collections is denoted by $c^{(\pi)}$.

Given a `combine` operator *op* and a partitioning policy π, the semantics of operator `p-`*op* π is as follows:

$$\texttt{p-}op\ \pi\ c = \left\{op\ c' \bullet c' \in c^{(\pi)}\right\}$$

As for binary operators, given an operator *op* in the `b-map` family and a partitioning policy π, the semantics of `p-`*op* π is defined as follows, for all sub-collections mapped to the same key $k \in K$:

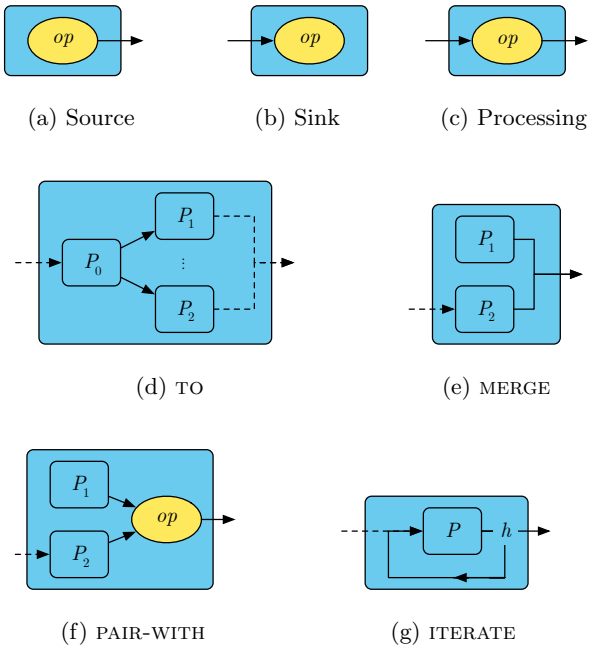
$$\texttt{p-}op\ \pi\ (m_1, m_2) = \bigcup \{op\ (\sigma_k^\pi(m_1), \sigma_k^\pi(m_2))\}$$

The common *group-by-key* partitioning, with π_1 being the left projection,[4] can be identified by collections with data type $K \times V$ and $\pi = \pi_1$.

[2] More precisely, $z \in S$ is the initial value, $\oplus_1 : S \times T \to S$ is the fold (how each input item affects the accumulated state) and $\oplus_2 : S \times S \to S$ is the reduce (how the various states are combined into a final value).

[3] An operator is *bounded* if it only deals with bounded collections.

[4] $\pi_1(x, y) = x$



(a) Source (b) Sink (c) Processing

(d) TO (e) MERGE

(f) PAIR-WITH (g) ITERATE

Figure 1: Graphical representation of PiCo Pipelines.

3.4. Pipelines

PiCo's cornerstone concept is the *Pipeline*. The grammar for Pipelines is essentially a graph-composition of Operators (Sect. 3.4.1), that allows to map each Pipeline to a corresponding Dataflow graph, representing its semantic counterpart (Sect. 3.4.2).

3.4.1. Pipeline Syntax

Fig. 1 presents the grammar for Pipelines. Pipelines in Figs. 1a, 1b, and 1c, respectively, produce, consume, and process data, through the operator *op*. Fig. 1d is the TO composition, in which the output from Pipeline p is sent as input to every downstream Pipeline—a particular case is when $n = 1$, which then represents basic Pipeline chaining. Fig. 1e is the MERGE composition, in which the outputs from two Pipelines are merged. Fig. 1f is the PAIR-WITH composition, in which the input is paired with the output of an internal Pipeline P, through the binary Operator *op*. Finally, Fig. 1g is the ITERATE constructor, that represents iterative processing by feeding the output of the internal Pipeline P back as input to P itself, until some termination condition (denoted by h in Fig. 1g) holds.

Note that in Fig. 1, a dashed line means the path may be void. We refer to such input- or output-less Pipelines as, respectively, *sources* and *sinks*. In particular, a Pipeline is *executable* only if it is both input-less and output-less. The type system for Pipelines, which we omit for the sake of simplicity, defines legal compositions with respect to both data and structure types—see the whole formalization for further details [17]. We remark that, to ease the implementation of composition and typing, we prohibit legal pipelines to have multiple input or output paths.

3.4.2. Pipeline Semantics

The semantics of a Pipeline is obtained by mapping its syntactic tree to a Dataflow graph, such that Operators are mapped to Dataflow *nodes*, Dataflow *edges* are inferred from the Pipeline structure, and Dataflow *tokens* represent either synchronization signals (for sources, e.g., a firing signal for an `emit` Operator that allows to read from a file: cf. sect. 3.3) or whole collections. Therefore, the proposed mapping is *denotational* rather than operational, in the sense that it models the (abstract) functional transformations applied to (possibly unbounded) data collections, rather than any specific data processing.

The most basic mapping is for a single-Operator Pipeline (Figs. 1a, 1b, and 1c), resulting in a Dataflow graph with a single node that, when fired upon the arrival of an input token, produces and emits a token according to its Operator semantics; moreover, if the Operator is input-less, a firing (synchronization) token is preloaded on the incoming edge, so that the node is activated at the beginning of the (logical) execution.

The semantic mapping for compound Pipelines is defined by structural induction. Some mappings introduce additional Dataflow nodes, with respect to syntactic nodes; for instance, mapping TO and MERGE Pipelines introduces gathering nodes at the downstream end of the resulting Dataflow graph. Moreover, some mappings require to modify the internal behavior of Dataflow nodes; for instance, mapping a TO Pipeline requires modifying the most upstream node, so that it emits n copies of the produced tokens, one for each downstream sub-graph (resulting from mapping each of the downstream Pipeline in Fig. 1d).

Finally, mapping ITERATE Pipelines is the most complex case, since it requires to both: 1. introduce an additional upstream (resp. downstream) *switch* node to control the input source (resp. output destination) for the iteration "box"; 2. introduce additional edges from the downstream switch to each input-less node within the iteration box, so that firing tokens are emitted to trigger the next iteration.

4. PiCo Implementation

In this section, we present an implementation of the model discussed in Section 3. The presented implementation is available as open-source code.[5]

The logical path from an (abstract) PiCo Pipeline to the corresponding runtime-level support is illustrated in

[5] `https://github.com/alpha-unito/PiCo`

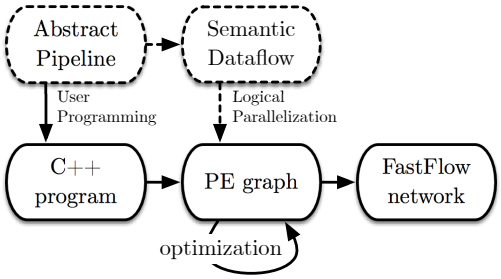


Figure 2: PiCo implementation schema.

Fig. 2. First, the user translates the abstract Pipeline into a C++ program, using a *fluent-style C++ API* (Sect. 4.1). Then, the program is mapped to a *parallel execution (PE) graph*, in which each operator is converted into a multi-node Dataflow network, by introducing as much parallelism as possible and performing network-to-network optimizations (Sect. 4.2). Finally, the PE graph is mapped to an actual thread network, on top of the FastFlow parallel programming framework (Sect. 4.3).

Let us stress, as highlighted in Fig. 2, that every PiCo program is endowed with an abstract semantics that allows the user to reason about data transformations (i.e., business logic) without dealing with any operational aspect (related with program execution).

Finally, we remark that the choice of implementing PiCo as a C++ stack was driven by the aim of minimising both the run-time overhead and the resource consumption. This is desirable from a high-performance perspective, but also a fundamental aspect in the Fog context, where dealing with limited resources is paramount (cf. desideratum 3). The experimental evaluation in Sect. 5 provides some quantitative assessment of this aim.

4.1. C++ API

In this section, we provide a C++ API for expressing PiCo Pipelines (Sect. 4.1.1) and Operators (Sect. 4.1.2). As a fundamental design choice, we adopted a functional fluent-style interface, exploiting method cascading (aka. method chaining) to relay the instruction context to a subsequent call. According to the functional design, each function call returns a fresh object (either Pipeline or Operator[6]), so that natural composition is supported. Moreover, the API is endowed with a type system that reflects the polymorphism provided by the abstract model, relying on both compile-time and run-time type polymorphism (Sect. 4.1.3). Finally, in Sect. 4.1.4, we show the complete source code for a PiCo implementation of the PageRank algorithm.

[6]Throughout the present paper, we seamlessly abuse the terms Pipeline and Operator to denote both abstract entities and C++ objects of the `Pipe` and `Operator` classes.

4.1.1. Pipes

Table 1 summarizes the most relevant functions in the API for the `Pipe` class, whose instances represent PiCo Pipelines. For each function, the corresponding abstract meaning is described.

The first function is the default constructor, that produces an empty Pipeline, whereas the second constructor produces a single stage Pipeline, containing only the argument Operator. The `add` function produces a Pipeline by adding the argument Operator to the subject Pipeline `p`. Similarly, the two flavors of the `to` function attach to `p`, respectively, a single Pipeline and a series of independent Pipelines, each receiving as input the output of `p`. It can be easily observed that `add(op)` is just syntactic sugar for `to(Pipe{op})`.

The `pairWith` function is similar to the single-argument `to` function, but it routes the output of both the subject and the argument Pipelines into a binary Operator. Similarly, the `merge` function merges the outputs from the two Pipelines. Finally, the `iterate` function produces an iterative version of the subject Pipeline, taking as input a generic termination condition. In the current implementation, only definite (fixed-length) iterations are provided, but the API is general enough to support arbitrarily complex conditions.

The above functions allow to define the structure of a Pipeline, indeed they correspond to the grammar rules illustrated in Fig. 1. In addition to the structural functions, the `run` function triggers the execution of a Pipeline, provided the Pipeline is executable (cf. Sect. 3.4.1).

4.1.2. Operators

The second part of the C++ PiCo API represents the PiCo operators. Following the grammar in Sect. 3.3, we organize the API in a hierarchical structure of unary and binary operator classes. The design of the operators API is based on inheritance to easily follow the grammar describing all operators; nevertheless, we recognize that using generic programming (e.g., based on templates) would lead to safer code, in which composition errors could be detected at compile-time rather than run-time.

`UnaryOperator` is the base class representing PiCo unary operators, i.e., those with no more than one input or output collection. For instance, a `Map` object takes a C++ callable value (i.e., a kernel) as parameter and represents a PiCo operator `map`, which processes a collection by applying the kernel to each item. Also, `ReadFromFile` is a subclass of `UnaryOperator` which represents those PiCo operators that produce a (bounded) unordered collection of text lines, read from an input file.

`BinaryOperator` is the base class representing operators with two input collections and one output collection. For

Function	Abstract Meaning
`Pipe()`	Create an empty Pipeline
`template<typename OpType> Pipe(OpType&)`	Create a Pipeline from an initial operator
`template<typename OpType> Pipe add(OpType&)`	Create a Pipeline by adding a new stage to the subject
`Pipe to(Pipe&)`	Create a Pipeline by appending the argument Pipeline to the subject
`Pipe to(std::vector<Pipe*>&)`	Create a Pipeline by appending a set of Pipelines, all fed by the subject
`template<typename BinaryOpType> Pipe pairWith(Pipe&, BinaryOpType&)`	Create a Pipeline by pairing the subject with an argument Pipeline, through a binary operator
`Pipe merge(Pipe& pipe)`	Create a Pipeline by merging the subject with the argument Pipeline
`template<typename TerminationPolicy> Pipe iterate(TerminationPolicy&)`	Create a Pipeline that iterates the subject until the termination condition is met
`void run()`	Execute the subject Pipeline

Table 1: The `Pipe` class API. Although `const` annotations are omitted for readability, all the arguments are `const` and all the functions are marked as `const`, yielding a purely functional API.

instance, a `BinaryMap` object represents a PiCo Operator in the `b-map` family (e.g., `join-map`), that processes pairs of elements coming from two different input collections and produces a single output for each pair. A `BinaryMap` object is passed as parameter to Pipeline objects built by calling the `pairWith` member function (cf. Table 1). Table 2 summarizes the constructors for the subclasses of the `Operator` class, whose instances represent PiCo Operators. For each constructor, the corresponding abstract Operator is described.

The constructors `Map`, `FlatMap`, and `Reduce` produce core data-parallel Operators (cf. Sect. 3.3.1). The `FlatMap` constructor takes as input a `FlatMapCollector` object, whose `add` method is used to build the output collection to be emitted by a kernel instance. Similarly, the `JoinMap` constructor produces a (core) binary `join-map` Operator (cf. Sect. 3.3.2).

In addition to core Operators, some constructors are provided that produce partitioning Operators (cf. Sect. 3.3.4). For instance, the `ReduceByKey` constructor produces a `p-reduce` Operator, that reduces a key-value collection by applying the kernel function to each value mapped to a given key. Similarly, a `JoinMapByKey` produces a `p-join-map` Operator, that applies the kernel function to each pair generated by joining two key-value collections, on a per-key basis. Moreover, although not shown in Table 2, the `window` function can be invoked on any supported Operator (e.g., a `combine` Operator), by passing as input a windowing policy, yielding a windowing Operator.

Constructors for source and sink Operators are also provided. For instance, the `ReadFromFile` constructor produces a `from-file` Operator, that reads data from an input file on a per-line basis and returns each line as a `std::string`. Dually, the `WriteToFile` constructor produces a `to-file` Operator, that writes data to an output file through the `<<` operator. Finally, `ReadFromSocket` and `WriteToSocket` are the socket-based sources and sinks.

4.1.3. Polymorphism

One distinguishing feature of the PiCo programming model, compared to other state-of-the-art frameworks, is that the same syntactic object (viz., a Pipeline or an Operator) can be used to process data collections having different types. As discussed in Sect. 3, Pipelines and Operators are polymorphic with respect to both data types and structure types (i.e., the "shape" of the collections, such as bag or stream).

In the proposed C++ API, the data type polymorphism is expressed at compile-time by implementing operators as template classes. As shown in Table 2, each operator takes a template parameter representing the data type of the collections processed by the operator. Namely, any *copy-constructible*[7] data type is supported. Moreover, each Pipeline object is decorated by the set of supported structure types.

In the proposed type system, all polymorphism is dropped in executable Pipelines. Therefore, executable `Pipe` objects have a unique type. By construction, source operators (i.e., objects of the `SourceOperator` class) play the role of specifying the unique structure type processed by the (executable) pipe they belong to. For instance, a pipe starting with a `ReadFromFile` type operator will only process multi-sets, whereas a pipe starting with a `ReadFromSocket` type operator will only process streams.

The types supported by a `Pipe` object are not exposed by the C++ API, but they are exploited at run-time to

[7] `http://en.cppreference.com/w/cpp/concept/CopyConstructible`

Constructor	Abstract Operator
`template<typename T, typename U> Map(std::function<U(T&)> f)`	`map` f, with kernel function $f : T \to U$
`template<typename T, typename U> FlatMap(std::function<void(T&, FMapCollector<U>&)> f)`	`flatmap` f, with kernel function $f : T \to \mathcal{P}(U)$
`template<typename T> Reduce(std::function<T(T&, T&)> f)`	`reduce` f, with kernel function $f : T \times T \to T$
`template<typename K, typename V> ReduceByKey(std::function<V(V&, V&)> f)`	`p-reduce` π_1 f', with kernel function $f' : V \times V \to V$, that applies f to the values and leaves the K-typed keys unchanged (a slight variant of the semantics in Sect. 3.3.4)
`template<typename T1, typename T2> JoinMap(std::function<U(T1&, T2&)> f)`	`join-map` f, with kernel $f : T_1 \times T_2 \to U$
`template<typename T1, typename T2> JoinMapByKey(std::function<U(T1&, T2&)> f)`	`p-join-map` π_1 f, with kernel function $f : T_1 \times T_2 \to U$, with $T_1 = K \times T_1'$ and $T_2 = K \times T_2'$
`ReadFromFile(std::string &)`	`from-file`, reading data from the argument filename
`template<typename SocketType> ReadFromSocket(SocketType &, char)`	`from-socket`, reading data from the argument socket, where lines are separated by the argument delimiter
`template<typename T> WriteToFile(std::string&)`	`to-file`, writing data to the argument filename through the `<<` operator for type T
`template<typename SocketType, typename T> WriteToSocket(SocketType &)`	`to-socket`, writing data to the argument socket through the `<<` operator for type T

Table 2: Operator constructors. As for the `Pipe` API in Table 1, `const` annotations are omitted for readability but the API is purely functional.

ensure only legal PiCo Pipelines are built, thus they are part of the API specification. When a member function is called on a `Pipe` object, the runtime: 1. checks the legality of the call by inspecting the type of both the subject and the argument `Pipe` objects (type checking); 2. updates the type of the subject `Pipe` object (type inference).

4.1.4. Example: PageRank

The usage of the proposed API is illustrated in Listing 1 with the source code for the PageRank algorithm.

In the example, the top-level executable Pipeline is the `pageRank` object (line 60). At the input end, the `generateLinks` Pipeline (line 61) builds the input graph by: 1. reading lines from the input text file; 2. parsing each line, by mean of the `map` Operator `parseLinks` (line 14), into a `src-links` pair, where `src` is a page identifier and links is the set of pages cited by `src`. From the generated graph, the `map` Operator `generateInitialRanks` (line 23) generates the initial ranking by associating the default rank 1.0 to each page. The initial approximation is fed as input to the core Pipeline, built by iterating the `improveRanks` Pipeline (line 54) for 20 iterations. Finally, the `WriteToFile` Operator (line 64) writes the computed ranks to the output text file.

The `improveRanks` Pipeline, whose iteration is the core processing in the example, combines a ranking with the input graph (i.e., the output from `generateLinks`), through the binary Operator `computeContributions` (line 29). At the first iteration, the input ranking is the output from `generateInitialRanks`, whereas at iteration $i + 1$ it is the output from the previous i-th iteration. Internally, `computeContributions` joins the input collections (i.e., the input graph and the most recent ranking) based on page identifiers, so that, for a given page p, the pairwise kernel receives as input both p's neighbors (`nl`) and the rank associated to p (`nr`); for each of such pairs, the Operator generates a contribution from each of p's neighbor, according to the PageRank formula, by calling the `add` function on the collector (cf. Sect. 4.1.2). The contributions are reduced on a per-key basis, so that all the contributions for a given page, generated by different instances of the binary Operator, are summed up through the `p-reduce` Operator `sumContributions` (line 36). Finally, each contribution is converted into an updated rank through the normalization provided by the `map` Operator `normalize` (line 42).

Note that, differently for instance from the reference Spark implementation of Page Rank, there is no need to specify which data should be *cached*, since in PiCo only the runtime system is in involved in taking this kind of decisions.

4.2. Intermediate Optimizations

We now show how a PiCo program is mapped to a graph of *parallel processing* nodes. Logically, as shown in Fig. 2, the mapping takes as input a Semantic Dataflow graph representing a PiCo program (cf. 3.4.2) and produces a Parallel Execution (PE) Graph, representing a possible parallelization of the Semantic graph. The resulting PE graph is a classical macro-Dataflow network [19], in which tokens represent portions of data collections and nodes are persistent processing units, mapping input to output tokens, according to a pure functional behavior.

```
#include <pico/pico.hpp>
using namespace pico;

typedef std::string Node;
typedef float Rank;
typedef KeyValue<Node, Rank> NRank;
typedef KeyValue<Node, std::vector<Node>> NLinks;

constexpr float DAMPENING = 0.85;
unsigned VERTICES;

int main(int argc, char** argv) {
  // Map operator parsing lines into node-links pairs
  Map<std::string, NLinks> parseLinks {
    [] (const std::string &adj) {
      Node src;
      std::vector<Node> links;
      // Code omitted: Tokenize adj into src and links...
      return NLinks{src, links};
    }};

  // Map operator generating initial ranks for node-links
  Map<NLinks, NRank> generateInitialRanks {
    [] (const NLinks &nl) {
      return NRank{nl.Key(), 1.0};
    }};

  // By-key join + FlatMap operator, computing ranking updates
  JoinFlatMapByKey<NRank,NLinks,NRank> computeContribs {
    [] (const NRank &nr, const NLinks &nl, FMapCollector<KV> &c) {
      for( auto &dest: nl.Value() )
        c.add( NRank{dest, nr.Value() / nl.Value().size()} );
    }};

  // By-key reduce summing up contributions
  ReduceByKey<NRank> sumContribs {
    [] (Rank r1, Rank r2) {
      return r1 + r2;
    }};

  // Map operator normalizing node-rank pairs
  Map<NRank,NRank> normalize {
    [] (const NRank &nr) {
      float jump = (1 - DAMPENING) / VERTICES;
      return NRank(nr.Key(), nr.Value() * DAMPENING + jump);
    }};

  // The pipe for building the graph to be processed.
  Pipe generateLinks = Pipe{}
    .add(ReadFromFile(argv[1]))
    .add(parseLinks);

  // The pipe that gets iterated to improve the computed ranks
  Pipe improveRanks = Pipe{}
    .pairWith(generateLinks, computeContribs)
    .add(sumContribs)
    .add(normalize);

  // The whole pageRank pipe.
  Pipe pageRank = Pipe{}
    .to(generateLinks)
    .add(generateInitialRanks)
    .to(improveRanks.iterate(FixedLengthIteration<20>))
    .add(WriteToFile<NRank>(argv[2]));

   // Code omitted: count VERTICES...

  pageRank.run();

  return 0;
}
```

Listing 1: PageRank example in PiCo. Input and output filenames are taken as first and second command-line parameters, respectively.

Dataflow networks naturally express some basic forms of parallelism. For instance, non-connected nodes (i.e., independent nodes) may execute independently from each other, exploiting *embarrassing* parallelism. Moreover, connected nodes (i.e., data-dependent nodes) may process different tokens independently, exploiting *pipeline* parallelism (also known as task parallelism). Finally, each PiCo operator is compiled into a Dataflow (sub-)graph of nodes, each processing different portions of the data collection at hand, exploiting *data* parallelism.

We also provide a set of rewriting rules for optimizing PE graphs, similarly to what is done by optimizing compilers over intermediate representations. For instance, Fig. 3 shows the rewriting for a `map`/`p-reduce` composition. This optimization pattern is commonly referred as *shuffle*[8]: between the `map` and `p-reduce` operators, the data is moved from the `map` workers (u_i in the figure) to the `reduce` workers (v_i in the figure) by following a partitioning criterion. By shuffling data, only the data belonging to a given partition can be assigned to each worker, and the `reduce` operator produces a single value for each partition. Typically, data shuffling produces an *all-to-all* communication pattern among `map` and `reduce` workers, highlighted by the dotted box in Fig. 3c. As a further optimization, part of the reducing phase can be moved into the `map` workers, so that each `reduce` worker computes the final result for each key by combining partial results coming from `map` workers.

4.3. FastFlow Runtime

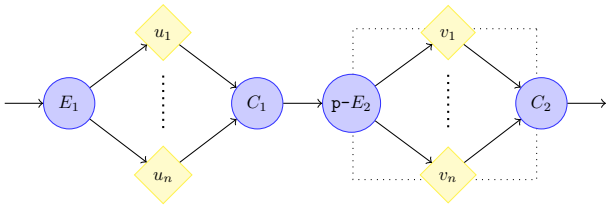
The bottom-level runtime support for PiCo is implemented on top of FastFlow [10], a programming framework in which lock-free parallel applications can be expressed as arbitrary networks of `ff_node` objects (i.e., threads). FastFlow provides a set of preset *core patterns*, capturing some common networks such as `ff_pipeline`, representing the pipeline pattern, and `ff_farm`, representing the farm pattern. We exploit `ff_node`, `ff_pipeline`, and `ff_farm` as building blocks for implementing PE graphs.

The following mapping holds for the API from Sect. 4.1,

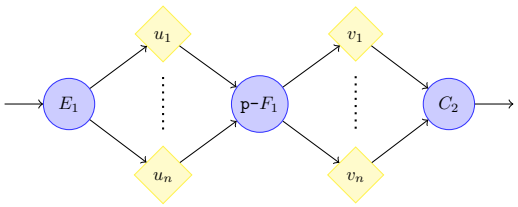
- `Pipe` objects (i.e., PiCo Pipelines) are implemented as `ff_pipeline` objects; in particular, iterative Pipelines are implemented as FastFlow pipelines with a feedback channel;
- All operators but sources and sinks are implemented as `ff_farm` objects, in which multiple workers are spawned to exploit data parallelism;
- Input and output operators are implemented as `ff_node` objects, thus only sequential input/output is provided in the current implementation.

In the prototypical implementation we present, the internal parallelism of each Operator is set manually (i.e., no automatic elasticity is provided). Although omitted for simplicity in Table 2, each Operator constructor takes as

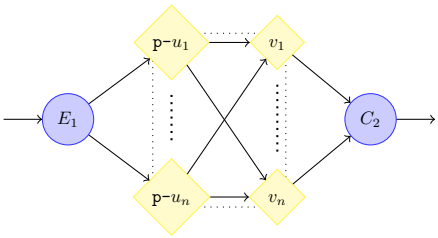
[8] The shuffle pattern is sometimes referred as "parallel-sorting."



(a) $\mathbb{C}\,[\![\texttt{map}\ f]\!] \circ \mathbb{C}\,[\![\texttt{p-(reduce}\ \oplus)\ \pi]\!]$



(b) Result of a *Forwarding Simplification* optimization



(c) Optimized shuffle by *All-to-All* optimization

Figure 3: Rewriting the PE graph for `map`/`p-reduce` composition.

input an optional argument specifying the degree of internal parallelism. A global default degree can also be specified as an environment variable.

In a runtime-level network, data collections are *streamed* across the threads (i.e., `ff_node` objects) in the form of pointers. To reduce the overhead induced by inter-thread communication, data collections are sliced into contiguous non-overlapping *chunks*. The chunk size is an execution parameter that controls the usual trade-off between latency and communication (i.e., grain size). FastFlow has been shown to not suffer even with ultra-fine grain and, indeed, we did not observe any significant effect induced by various chunk sizes. For the following experiments, we arbitrarily set this value to 512.

5. Experiments

In this section, the performance sustained by PiCo in a shared-memory setting is evaluated in comparison to two mainstream tools, Spark (v2.1.0) and Flink (v1.2.0).

The machine used for the experiments is the Occam Supercomputer[9] (Open Computing Cluster for Advanced data Manipulation) [20], designed and managed by the University of Torino and the National Institute for Nuclear Physics. We used one node having the following characteristics: Hardware: 4x Intel®Xeon®Processor E7-4830 v3 12 core/2.1Ghz, 768GB/1666MHz (48 x 16GB) DDR4 RAM, 1x SSD 800GB + 1x HDD 2TB/7200rpm, InfiniBand 56Gb + 2x Ethernet 10Gb. Software: Linux CentOS v7.3 with Linux kernel 3.10, gcc v4.8.5 compiler (PiCo was compiled with `O3` optimization flag), and OpenJDK Server v1.8 Java runtime.

For the sake of fairness, the comparison is carried on the configuration leading to the best performance, for each tool.[10] For Spark and Flink, as both rely on master-workers run-time systems, this means finding the optimal number of workers. For PiCo, each program is mapped to a different network of threads, requiring to explore a broader space of configurations. Although a general solution for automating this is beyond the current scope, some simple strategies are discussed for each benchmark.

The observed results are summarized in Table 3 (p. 12). Each reported measurement is the average of 10 repetitions. Since only negligible deviations were observed, no discussion about variance is included in the analysis.

This section proceeds as follows. Sect. 5.1 discusses two benchmarks conforming to the simple MapReduce pattern, namely Word Count and Stock Pricing. Sect. 5.2 discusses a streaming variant of Stock Pricing. Finally, Sect. 5.3 discusses Page Rank, a more complex application, typical of a real world application.

5.1. MapReduce-like Benchmarks

In Big Data analytics, *Word Count* is considered the "Hello, World!" application. In its batch flavour, Word Counts finds the number of occurrences of each word from a text file. Fig. 4 shows the semantic graph (cf. Sect. 3.4.2) for a PiCo Pipeline implementing Word Count.[11] The input is read line by line from the text file `foo`; each line is tokenized into a sequence of words using a `FlatMap` node (as each line contains a varying number of words) and each word w is mapped to a key-value pair $\langle w, 1\rangle$; the generated pairs are grouped by-word and then the values (i.e., the 1s) are summed up by a `ReduceByKey` node; finally, the result (i.e., one pair per word along with its number of occurrences in the text) is written to the text file `bar`.

The second benchmark is Stock Pricing, an application from the stock market domain that computes the maximum price for each option from a bunch of stock-option

[9] `http://c3s.unito.it/index.php/super-computer`

[10] Initialization/termination excluded.

[11] The semantic graph for a PiCo application can be generated in `dot` format by the `to_dotfile` function from the `Pipe` class API.

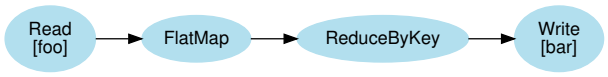


Figure 4: Semantic graph for Word Count in PiCo.

data. Its semantic graph is omitted since it is analogous to the Word Count graph (Fig. 4): each stock-option entry from a text file is parsed and processed by a `Map` node, to compute the stock-price associated to the entry, by means of the Black & Scholes formula; finally, a `ReduceByKey` extracts the maximum price for each stock name and the output is written to a text file.

5.1.1. Settings

For the Word Count benchmark, an input file of 6 GB is considered, formed by text lines of various lengths, each composed by random words from a dictionary of 1024 words. For the Stock Pricing benchmark, an input file of 10 GB is considered, generated by replicating the largest dataset included in the PARSEC[12] suite for the `blackscholes` benchmark.

The parallel configuration for PiCo can be controlled by setting the degree of parallelism for parsing and processing stages. Indeed, in the current implementation, both benchmarks are implemented at the FastFlow level as two pipelined farms, one for reading the input file and parsing lines into strings (ReadFromFile, Fig. 4), the other for applying the map-reduce processing (FlatMap and ReduceByKey, Fig. 4). Therefore, the optimal configuration is observed with the *minimal* parallelism degree for parsing that prevents a bottleneck at parsing side, leaving as many resources as possible to the heavier processing stage.

As we discuss in Sect. 6, more refined solutions can be envisioned based on a larger set of intermediate optimizations (cf. Sect. 4.2), leading to improved FastFlow networks. However, in the current implementation, the simple heuristic already discussed yields good performance, as we show in the following.

5.1.2. Results

Table 3 shows the comparison on minimum execution time, whereas Fig. 5 shows the relative speedup for the Word Count benchmark, with respect to the parallelism degree of the processing stage.

For Word Count, with the optimal parallelism for parsing found at 4, PiCo features the best performance with parallelism for processing at 36, that is also the maximum degree in Fig. 5 not causing overbooking. For the Stock Pricing benchmark, 8 parallel readers are needed to avoid the bottleneck, then lowering the optimal parallelism for processing to 32. With these configurations, PiCo outperforms both Spark (48 workers) and Flink (24 workers).

[12] `http://parsec.cs.princeton.edu/index.htm`

	Spark	Flink	PiCo
Min. Execution Time (s)			
Word Count	10.07	69.29	5.95
Stock Pricing (S.P.)	19.99	42.80	14.20
Page Rank	226.52	82.11	81.08
Max. Throughput (MB/s)			
Streaming S.P.	18.63	37.89	112.39
Additional Memory Footprint vs. PiCo (GB)			
Word Count	+4.45	+2.06	-
Stock Pricing (S.P.)	+16.80	+2.32	-
Page Rank	+38.95	+2.15	-
Streaming S.P.	+45.80	+1.73	-

Table 3: Outcome of the performance experiments. All measurements refer to the best-performing parallel configuration. Execution time and throughput are showed for batch and streaming benchmarks, respectively. The memory footprint for Spark and Flink is represented by the difference in memory usage with respect to PiCo.

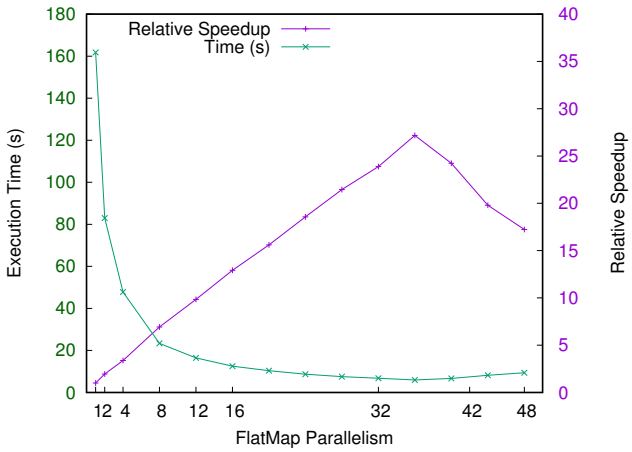


Figure 5: Scalability of Word Count in PiCo.

5.2. Streaming Benchmark

We now consider a *streaming* variant of the Stock Pricing benchmark discussed above. In this variant (referred as "Streaming S.P." in Table 3), the data comes from a socket (rather than a file) and more complex processing is done on each stock-option entry—Binomial Tree and Explicit Finite Difference are computed, in addition to the Black & Scholes formula, and the result is averaged. Finally, a *per-window* reduction is performed under fixed-size windowing with size 8. The semantic graph for this benchmark is again analogous to the graph in Fig. 4.

5.2.1. Settings

The same 10GB input file as in Stock Pricing is used, but streaming it to a network socket using the `netcat` tool.

First, let us note that the we could not do a totally fair comparison with Spark, since Spark does not provide fixed-size windowing.

As for PiCo, although the streaming variant is semantically similar to the Stock Pricing batch benchmark, its parallel configuration is different. First, in the current implementation, no parallelism is exploited when reading from/writing to sockets. Second, no fusion is applied between `Map` (or `FlatMap`) and per-window reductions, when performing intermediate optimizations. Therefore, in the streaming benchmark, the parallelism degrees for the two stages can be set independently.

5.2.2. Results

Table 3 shows the comparison on maximum sustained throughput. In terms of stock-option entries processed per second, PiCo processes more than 1.7M entries per second, outperforming Flink and Spark, that process approx. 600K and 300K entries per second, respectively.

The optimal parallel configuration for PiCo was found when not exploiting any parallelism on the reduce-side while setting the map-side parallelism to 24; Spark and Flink also attain the best performance with 24 workers.

In addition to poor throughput, Table 3 shows a striking *memory explosion* for Spark. To understand this poor performance, an analysis of bottlenecks in streaming platforms would need to be performed, which would also help determine the overhead induced by the JVM [21].

5.3. Page Rank

In this section, we now examine the Page Rank application, described in Sect. 4.1.4. Fig. 6 shows the semantic graph for a PiCo implementation, which is a slight variant of the code in Listing 1. In particular, this variant takes as input the list of pages and the list of links from two separate files (like the Flink implementation) and the adjacency lists are generated by per-page reduction of the links (the leftmost ReduceByKey node in Fig. 6).

5.3.1. Settings

The largest graph from the SNAP repository,[13] representing links between users of the LiveJournal on-line community, is given as input to the Page Rank implementations. The graph has approx. 4.8M nodes and 69M edges (i.e., edge factor is 14.37), requiring approx. 1GB of memory.

The parallel configuration for PiCo is relevant only for the critical portion of the application—the highlighted box in Fig. 6. Indeed, the execution time is dominated by the iterative processing. Within the critical portion, the optimal parallelism degree for the `ReduceByKey` stage was found at 6, whereas various degrees were tested for the (binary) `FlatMap` stage.

In the current implementation, as for the benchmarks in Sect. 5.1, only some map-reduce fusion is performed as intermediate optimizations (e.g., between `FlatMap` and `ReduceByKey` within the critical portion). Therefore, the performance will unavoidably suffer from significant *resource overbooking*, as discussed in more detail in Sect. 6.

5.3.2. Results

Table 3 shows the comparison for minimum execution time. The results show that, although suffering from limited scalability due to the factors discussed above, PiCo performs as well as Flink, reaching the optimal performance with parallelism degree set to 16 for the `FlatMap` stage, whereas Spark and Flink attain the best performance with 32 workers. Again, Spark exhibits the highest execution time and significant memory explosion.

6. Conclusions

In this paper, we presented PiCo, a new C++ API with a fluent interface for generic Big Data Analytics pipelines, based on a streaming runtime that provides high performance at low cost of resource consumption.

PiCo provides an expressive programming model backed by a *functional abstract semantics*. The abstract model is coupled with a concrete API expressed using modern C++, thus ensuring good code portability. One distinguishing feature of PiCo is the *polymorphic pipelines*, that allow uniform programming for different data models (i.e., stream or batch processing).

The experiments performed, where we compared execution times in shared memory for both batch and stream applications, showed that PiCo attained the best execution time when compared to two state-of-the-art frameworks, Spark and Flink. Moreover, the C++ streaming runtime provides efficiency also in terms of low memory footprint, thus showing it is possible to do high performance analytics with low resource consumption.

The presented results allow us to advocate the exploitation of PiCo for emerging Fog scenarios, as it addresses some fundamental challenges inherent in these systems: batch and stream programs can be assembled by composition (enabled by the abstract model) and executed over a lightweight, high-performance runtime. In this respect, we remark that both Spark and Flink are systems supporting distributed memory hardware holistically, and both the higher memory footprint and the longer execution time may be strongly influenced by the support for system-wide fault tolerance and data distribution.

[13] `https://snap.stanford.edu/data/`

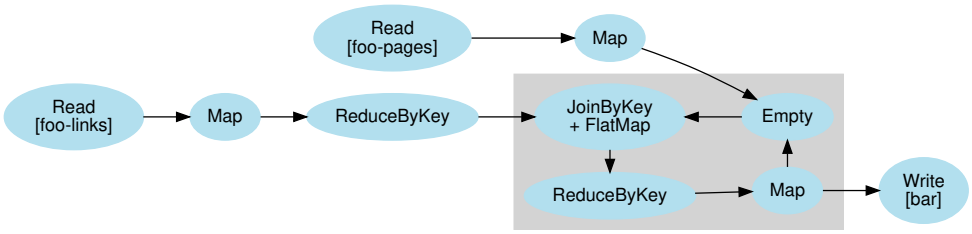


Figure 6: Semantic graph for Page Rank in PiCo.

As future work, we plan to design and implement a broader class of intermediate optimisations for runtime-level networks, with the aim of minimising both the number of threads—thus alleviating the performance penalties due to overbooking—and the amount of inter-thread communication. In the same perspective, we will consider adopting some strategy to automate (at least partially) the process of finding optimal parallel configurations. Finally, we plan to investigate a distributed-memory implementation on top of some distributed-shared memory system. In that context, we will also address the issues of fault-tolerance (e.g., automatic restores in case of failures).

Acknowledgments

We would like to thank the C3S supercomputing facility for the support with the OCCAM SuperComputer. This work has been partially supported by the EU-H2020 RIA project "Toreador" (no. 688797), the EU-H2020 RIA project "Rephrase" (no. 644235), and the 2015-2016 IBM Ph.D. Scholarship program.

References

[1] Cisco (White Paper), Fog computing and the internet of things: Extend the cloud to where the things are, `https://www.cisco.com/c/dam/en_us/solutions/trends/iot/docs/computing-overview.pdf` (2015).

[2] W. Shi, S. Dustdar, The promise of edge computing, IEEE Computer 49 (5) (2016) 78–81.

[3] F. Bonomi, R. A. Milito, P. Natarajan, J. Zhu, Fog computing: A platform for internet of things and analytics, in: N. Bessis, C. Dobre (Eds.), Big Data and Internet of Things: A Roadmap for Smart Environments, Vol. 546 of Studies in Computational Intelligence, Springer, 2014, pp. 169–186.

[4] A. Brogi, S. Forti, QoS-aware deployment of IoT applications through the fog, IEEE Internet of Things Journal 4 (5) (2017) 1185–1192.

[5] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernàndez-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, S. Whittle, The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing, Proceedings of the VLDB Endowment 8 (2015) 1792–1803.

[6] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, I. Stoica, Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing, in: Proc. of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI), 2012.

[7] M. Fowler, Domain-Specific Languages, Addison-Wesley, 2011.

[8] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, D. Ryaboy, Storm@twitter, in: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, 2014, pp. 147–156.

[9] P. Carbone, G. Fóra, S. Ewen, S. Haridi, K. Tzoumas, Lightweight asynchronous snapshots for distributed dataflows, CoRR abs/1506.08603.

[10] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Torquati, Fastflow: high-level and efficient streaming on multi-core, in: Programming Multi-core and Many-core Computing Systems, Parallel and Distributed Computing, Wiley, 2017, Ch. 13.

[11] C. Misale, M. Drocco, G. Tremblay, M. Aldinucci, PiCo: a novel approach to stream data analytics, in: Euro-Par 2017: Parallel Processing Workshops, LNCS, Springer, 2018.

[12] C. Misale, M. Drocco, M. Aldinucci, G. Tremblay, A comparison of big data frameworks on a layered dataflow model, Parallel Processing Letters 27 (01).

[13] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, I. Stoica, Discretized streams: Fault-tolerant streaming computation at scale, in: Proc. of the 24th ACM Symp. on Operating Syst. Principles, ACM, New York, NY, USA, 2013, pp. 423–438.

[14] A. Alexandrov, R.Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, D. Warneke, The stratosphere platform for big data analytics, The VLDB Journal 23 (6) (2014) 939–964.

[15] T. Bingmann, M. Axtmann, E. Jöbstl, S. Lamm, H. C. Nguyen, A. Noe, S. Schlag, M. Stumpp, T. Sturm, P. Sanders, Thrill: High-performance algorithmic distributed batch data processing with C++, CoRR abs/1608.05634.

[16] H. Miao, H. Park, M. Jeon, G. Pekhimenko, K. S. McKinley, F. X. Lin, StreamBox: Modern stream processing on a multi-core machine, in: Proc. of the 2017 Usenix Annual Technical Conference (USENIX), 2017, pp. 617–629.

[17] C. Misale, PiCo: A domain-specific language for data analytics pipelines, Ph.D. thesis, Computer Science Department, University of Torino (May 2017).

[18] M. Drocco, C. Misale, G. Tremblay, M. Aldinucci, A formal semantics for data analytics pipelines, Tech. rep., Computer Science Department, University of Torino (May 2017).

[19] E. A. Lee, T. M. Parks, Dataflow process networks, Proc. of the IEEE 83 (5) (1995) 773–801.

[20] M. Aldinucci, S. Bagnasco, S. Lusso, P. Pasteris, S. Rabellino, The Open Computing Cluster for Advanced data Manipulation (OCCAM), in: Journal of Physics: Conf. Series 898 (CHEP 2016), San Francisco, USA, 2017.

[21] S. Yang, Y. Jeong, C. Hong, H. Jun, B. Burgstaller, Scalability and state: A critical assessment of throughput obtainable on big data streaming frameworks for applications with and without state information, in: Euro-Par 2017: Parallel Processing Workshops, 2018, pp. 141–152.