University of Torino

Doctoral School on Science
and High Technology

Computer Science Department

Doctoral Thesis

PiCo:
A Domain-Specific Language for Data Analytics Pipelines

Author:
Claudia Misale
Cycle XVIII

Supervisor:
Prof. Marco Aldinucci
Co-Supervisor:
Prof. Guy Tremblay

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy

"*Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it.*
And to make matters worse: complexity sells better. "

Edsger W. Dijkstra, *"On the nature of Computing Science"*
Marktoberdorf, 10 August 1984

UNIVERSITY OF TORINO

Abstract

Computer Science Department

Doctor of Philosophy

PiCo:
A Domain-Specific Language for Data Analytics Pipelines

by Claudia Misale

In the world of Big Data analytics, there is a series of tools aiming at simplifying programming applications to be executed on clusters. Although each tool claims to provide better programming, data and execution models—for which only informal (and often confusing) semantics is generally provided—all share a common underlying model, namely, the Dataflow model. Using this model as a starting point, it is possible to categorize and analyze almost all aspects about Big Data analytics tools from a high level perspective. This analysis can be considered as a first step toward a formal model to be exploited in the design of a (new) framework for Big Data analytics. By putting clear separations between all levels of abstraction (i.e., from the runtime to the user API), it is easier for a programmer or software designer to avoid mixing low level with high level aspects, as we are often used to see in state-of-the-art Big Data analytics frameworks.

From the user-level perspective, we think that a clearer and simple semantics is preferable, together with a strong separation of concerns. For this reason, we use the Dataflow model as a starting point to build a programming environment with a simplified programming model implemented as a Domain-Specific Language, that is on top of a stack of layers that build a prototypical framework for Big Data analytics.

The contribution of this thesis is twofold: first, we show that the proposed model is (at least) as general as existing batch and streaming frameworks (e.g., Spark, Flink, Storm, Google Dataflow), thus making it easier to understand high-level data-processing applications written in such frameworks. As result of this analysis, we provide a layered model that can represent tools and applications following the Dataflow paradigm and we show how the analyzed tools fit in each level.

Second, we propose a programming environment based on such layered model in the form of a Domain-Specific Language (DSL) for processing data collections, called PiCo (***Pi***peline ***Co***mposition). The main entity of this programming model is the Pipeline, basically a DAG-composition of processing elements. This model is intended to give the user an unique interface for both stream and batch processing, hiding completely data management and focusing only on operations, which are represented by Pipeline stages. Our DSL will be built on top of the FastFlow library, exploiting both shared and distributed parallelism, and implemented in C++11/14 with the aim of porting C++ into the Big Data world.

Acknowledgements

Questi quattro anni sono stati veramente brevi. Praticamente non me ne sono accorta. Ho imparato tante cose, molte ancora non ho capito come si fanno (ad esempio non so fare le revisioni in maniera educata ed elegante, non ho imparato a parlare in pubblico, non so riassumere i meeting..) e faccio ancora arrabbiare tantissimo il mio supervisore, il Professor Marco Aldinucci. A lui, infatti, dire semplicemente "grazie" non è abbastanza. Specie per la pazienza che ha avuto con me, da tutti i punti di vista.

These four years were really short. I did not notice they are gone. I learned a lot, I still do not understand how to do a lot of things (for instance I am not able to review papers in a polite and elegant way, I did not learn to talk in public, I am not able to summarize meetings..) and I still make my supervisor very angry with me, Professor Marco Aldinucci. It is not enough to say "thank you" to him. Especially for the patience he had with me, from all points of view.

Sono arrivata a Torino senza conoscerlo, chiedendogli di essere il mio supervisore per il dottorato. Non potevo fare scelta migliore. Tutto quello che so lo devo a lui. Le ore passate ad ascoltarlo spiegarmi i mille aspetti di quest'area di ricerca (anche se per lui, di aspetti, ce ne sono sempre e solo due), mi hanno aperto la mente e mi hanno formato tantissimo, anche se ancora ho tutto da imparare. Quattro anni, infatti, non bastano. Mi ha sempre spronato a fare meglio, a guardare dentro le cose, e cercherò di portarmi dietro i suoi insegnamenti ovunque la vita mi porterà. Inoltre la sua simpatia e i suoi modi di fare, sono stati ottimi compagni in questi anni di dottorato. Sì, anche gli insulti lo sono stati! Grazie, theboss. Di cuore.

I arrived in Turin without knowing that much about Prof. Aldinucci and I asked him to be my supervisor. I could not make a better choice. Everything I know is thanks to him. I spent a lot of hours listening to him explaining all aspects about this research area (even if, from his point of view, there are always only two aspects), this opened my mind and made me grow up a lot, even if I still have to learn. Four years are not enough, actually. He always pushed me to do my best, to look inside things, and I will always try to bring his teachings with me, wherever I will go. Furthermore, his pleasantness and his way to do things, have been perfect companionship in these years. Thank you, theboss. Thank you so much.

Ci sono alcune persone che voglio ringraziare, perché tanto mi hanno dato e sarò sempre in debito con loro per questo.

There are a few people I want to thank, because they gave me so much and I will always be in debt with them.

Vorrei iniziare con il mio co-supervisore, il Professor Guy Tremblay. Grazie di avermi seguito e di avermi aiutato così tanto nella stesura della tesi e nel capire tanti argomenti che, per entrambi, erano nuovi: la tua grande esperienza è stata illuminante, sei stato una guida importantissima. Grazie ancor di più per i tanti consigli per la presentazione che mi hai dato, ho cercato di seguirli al meglio e spero di esserne stata all'altezza.

I would like to start with my co-supervisor, professor Guy Tremblay. Thank you so much for helping me so much while writing this thesis and thank you for helping me in understanding topics that, for both, were new: your great experience has been enlightening, you have been a very important guide. Thank you even more for all the advices about the presentation, I tried to follow all of them the best I can do and I hope I have been able to do that.

Vorrei ringraziare i revisori per il lavoro di revisione che hanno fatto: i loro commenti sono stati preziosissimi e molto dettagliati. Mi hanno permesso di migliorare il manoscritto e soprattutto di rivedere molti aspetti che l'inesperienza non mi permette ancora di vedere.

I would like to thank reviewers for the great job they did, their impressive knowledge and experience, and the time the dedicated to my work: their comments have been precious and very detailed. Their advices made me improve the manuscript and, mostly, to review a lot of aspects that the inexperience makes me not able to see them yet.

Un profondo grazie va ai miei colleghi: una cricca di persone uniche con cui ho condiviso tanto e tanto mi hanno dato. Le nostre serate "non facciamo tardi" che finivano alle 4 o 5 del mattino, le mangiate, le uscite, i giochi, i mille discorsi, le Gossip Girlz.. Grazie di tutto ciò che mi avete dato.

Grazie a Caterina, la mia secolare amica. Tu sai tutto di me, che altro ti devo dire. Siamo cresciute insieme nei cinque anni di università all'UniCal in cui ne abbiamo combinate tante e mi hai assistito durante questi anni: sei anche tu una colonna portante di questo dottorato e una portante nella mia vita.

Quasi tutte le pause pranzo del mio dottorato le ho trascorse in palestra a "picchiare la gente", come dicono i miei colleghi. Le lezioni di kick-boxing/muay-thai sono state assolutamente una droga e a renderle tali sono state innanzitutto le persone che ho conosciuto lì. Grazie al Maestro Beppe e ai ragazzi, ho imparato ad apprezzare questo magnifico sport fatto di disciplina, rispetto, autocontrollo, che mi ha permesso di spingermi oltre i ciò che pensavo fossero i miei limiti fisici e mentali. Grazie, Maestro, per avermi fatto scoprire un lato di me a me sconosciuto e grazie della persona e amico che sei. Grazie a tutti i miei "amici delle botte", grazie delle botte che mi avete dato, della perfetta compagnia che siete sia in palestra che fuori, per avermi aiutato a scaricare l'ansia accumulata ogni giorno.

Grazie a Roberta, la mia amica di botte numero uno in assoluto. Abbiamo condiviso tanto in palestra e siamo cresciute insieme, mi hai fatto crescere e mi hai insegnato tanto. Soprattutto, hai ascoltato e sopportato tutti i miei sfoghi e le mie frustrazioni da dottoranda. Mai ti ringrazierò abbastanza.

E infine, i pezzi grossi. La mia famiglia, a cui tutto devo. Soprattutto per la pazienza che avete avuto con me: so che è stata dura, infatti vi chiedo scusa per questo. Grazie per l'avermi sempre appoggiato in tutte le mie scelte e per avermi permesso di raggiungere questo traguardo. Vorrei sempre rendervi orgogliosi di me e spero che questo dottorato mi aiuti in questo obiettivo. Perché siete il mio esempio di correttezza e forza.

Maurizio. Ti ho conosciuto durante il nostro dottorato, sei diventato parte della mia vita dal primo istante. Abbiamo condiviso questo percorso insieme: i momenti belli, quelli un po' critici, la fatica mentale e fisica. La tua intelligenza e le tue intuizioni mi hanno sempre affascinato e ispirato. Spero sarai orgoglioso, almeno un po', di me alla fine di questa storia. Lo sai che.

Contents

Abstract iii

Acknowledgements v

1 Introduction 1
1.1 Results and Contributions . . . 1
1.2 Limitations and Future Work . . . 4
1.3 Plan of the Thesis . . . 5
1.4 List of Papers . . . 5
1.5 Funding . . . 7

2 Technical Background 9
2.1 Parallel Computing . . . 9
2.2 Platforms . . . 9
2.2.1 SIMD computers . . . 10
Symmetric shared-memory multiprocessors . . . 10
Memory consistency model . . . 11
Cache coherence and false sharing . . . 12
2.2.2 Manycore processors . . . 12
2.2.3 Distributed Systems, Clusters and Clouds . . . 14
2.3 Parallel Programming Models . . . 14
2.3.1 Types of parallelism . . . 15
Data parallelism . . . 16
Map . . . 16
Reduce . . . 17
2.3.2 The Dataflow Model . . . 18
Actors . . . 18
Input channels . . . 19
Output channels . . . 19
Stateful actors . . . 19
2.3.3 Low-level approaches . . . 19
2.3.4 High-level approaches . . . 21
2.3.5 Skeleton-based approaches . . . 22
Literature review of skeleton-based approaches . . . 22
2.3.6 Skeletons for stream parallelism . . . 23
2.4 Programming multicore clusters . . . 25
2.4.1 FastFlow . . . 25
Distributed FastFlow . . . 28
2.5 Summary . . . 29

3 Overview of Big Data Analytics Tools 31
3.1 A Definition for Big Data . . . 31
3.2 Big Data Management . . . 32
3.3 Tools for Big Data Analytics . . . 33
3.3.1 Google MapReduce . . . 33
The five steps of a MapReduce job . . . 34
3.3.2 Microsoft Dryad . . . 36
A Dryad application DAG . . . 36
3.3.3 Microsoft Naiad . . . 37

Timely Dataflow and Naiad programming model 37
3.3.4 Apache Spark 38
Resilient Distributed Datasets 39
Spark Streaming 40
3.3.5 Apache Flink 41
Flink Programming and Execution Model 41
3.3.6 Apache Storm 42
Tasks and Grouping 43
3.3.7 FlumeJava 44
Data Model and Transformations 44
3.3.8 Google Dataflow 45
Data Model and Transformations 46
3.3.9 Thrill 47
Distributed Immutable Arrays 48
3.3.10 Kafka 49
Producer-Consumer Distributed Coordination 49
3.3.11 Google TensorFlow 49
A TensorFlow application 50
3.3.12 Machine Learning and Deep Learning Frameworks 51
3.4 Fault Tolerance 51
3.5 Summary 52

4 High-Level Model for Big Data Frameworks 53
4.1 The Dataflow Layered Model 53
4.1.1 The Dataflow Stack 54
4.2 Programming Models 54
4.2.1 Declarative Data Processing 55
4.2.2 Topological Data Processing 56
4.2.3 State, Windowing and Iterative Computations 56
4.3 Program Semantics Dataflow 58
4.3.1 Semantic Dataflow Graphs 58
4.3.2 Tokens and Actors Semantics 59
4.3.3 Semantics of State, Windowing and Iterations 60
4.4 Parallel Execution Dataflow 60
4.5 Execution Models 64
4.5.1 Scheduling-based Execution 64
4.5.2 Process-based Execution 66
4.6 Limitations of the Dataflow Model 66
4.7 Summary 67

5 PiCo Programming Model 69
5.1 Syntax 69
5.1.1 Pipelines 71
5.1.2 Operators 71
Data-Parallel Operators 72
Pairing 73
Sources and Sinks 73
Windowing 74
Partitioning 74
5.1.3 Running Example: The WORD-COUNT Pipeline 75
5.2 Type System 75
5.2.1 Collection Types 75
5.2.2 Operator Types 76
5.2.3 Pipeline Types 77
5.3 Semantics 78
5.3.1 Semantic Collections 78
Partitioned Collections 79
Windowed Collections 79

5.3.2 Semantic Operators . . . 80
Semantic Core Operators . . . 80
Semantic Decomposition . . . 81
Unbounded Operators . . . 82
Semantic Sources and Sinks . . . 82
5.3.3 Semantic Pipelines . . . 82
5.4 Programming Model Expressiveness . . . 84
5.4.1 Use Cases: Stock Market . . . 84
5.5 Summary . . . 86

6 PiCo Parallel Execution Graph 87
6.1 Compilation . . . 87
6.1.1 Operators . . . 89
Fine-grained PE graphs . . . 90
Batching PE graphs . . . 91
Compilation environments . . . 92
6.1.2 Pipelines . . . 92
Merging Pipelines . . . 93
Connecting Pipelines . . . 93
6.2 Compilation optimizations . . . 94
6.2.1 Composition and Shuffle . . . 94
6.2.2 Common patterns . . . 94
6.2.3 Operators compositions . . . 96
Composition of `map` and `flatmap` . . . 96
Composition of `map` and `reduce` . . . 97
Composition of `map` and `p-reduce` . . . 98
Composition of `map` and `w-reduce` . . . 99
Composition of `map` and `w-p-reduce` . . . 100
6.3 Stream Processing . . . 101
6.4 Summary . . . 101

7 PiCo API and Implementation 103
7.1 C++ API . . . 103
7.1.1 Pipe . . . 104
7.1.2 Operators . . . 105
The `map` family . . . 106
The `combine` family . . . 107
Sources and Sinks . . . 108
7.1.3 Polymorphism . . . 109
7.1.4 Running Example: Word Count in C++ PiCo . . . 111
7.2 Runtime Implementation . . . 111
7.3 Anatomy of a PiCo Application . . . 113
7.3.1 User level . . . 113
7.3.2 Semantics dataflow . . . 114
7.3.3 Parallel execution dataflow . . . 114
7.3.4 FastFlow network execution . . . 115
7.4 Summary . . . 117

8 Case Studies and Experiments 119
8.1 Use Cases . . . 119
8.1.1 Word Count . . . 119
8.1.2 Stock Market . . . 119
8.2 Experiments . . . 120
8.2.1 Batch Applications . . . 120
PiCo . . . 121
Comparison with other frameworks . . . 123
8.2.2 Stream Applications . . . 127
PiCo . . . 128

Comparison with other tools 129
8.3 Summary 132

9 Conclusions **133**

A Source Code **137**

List of Figures

2.1 Structure of an SMP. . . . 11
2.2 *Map* pattern . . . 16
2.3 *Reduce* pattern . . . 17
2.4 *MapReduce* pattern . . . 18
2.5 A general *Pipeline* pattern with three stages. . . . 24
2.6 *Farm* pattern . . . 24
2.7 *Pipeline of Farm* . . . 25
2.8 Layered FastFlow design . . . 27

3.1 An example of a MapReduce Dataflow . . . 35
3.2 A TensorFlow application graph . . . 50

4.1 Layered model representing the levels of abstractions provided by the frameworks that were analyzed. . . . 54
4.2 Functional Map and Reduce dataflow expressing data dependencies. 58
4.3 A Flink JobGraph (4.3a). Spark DAG of the WordCount application (4.3b). . . . 58
4.4 Example of a Google Dataflow Pipeline merging two PCollections. . 59
4.5 MapReduce execution dataflow with maximum level of parallelism reached by eight *map* instances. . . . 61
4.6 Spark and Flink execution DAGs. . . . 61
4.7 Master-Workers structure of the Spark runtime (a) and Worker hierarchy example in Storm (b). . . . 65

5.1 Graphical representation of PiCo Pipelines . . . 70
5.2 Unbounded extension provided by windowing . . . 76
5.3 Pipeline typing . . . 77

6.1 Grammar of PE graphs . . . 88
6.2 Operators compilation in the target language. . . . 89
6.3 Compilation of a MERGE Pipeline . . . 93
6.4 Compilation of TO Pipelines . . . 93
6.5 Forwarding Simplification . . . 94
6.6 Compilation of `map`-to-`map` and `flatmap`-to-`flatmap` composition. . . 96
6.7 Compilation of `map-reduce` composition. . . . 97
6.8 Compilation of `map`-to-`p-reduce` composition. . . . 98
6.9 Compilation of `map`-to-`w-reduce` composition. Centralization in `w-`F_1 for data reordering before `w-reduce` workers. . . . 99
6.10 Compilation of `map`-to-`w-p-reduce` composition. . . . 100

7.1 Inheritance class diagram for `map` and `flatmap`. . . . 107
7.2 Inheritance class diagram for `reduce` and `p-reduce`. . . . 107
7.3 Inheritance class diagram for `ReadFromFile` and `ReadFromSocket`. . 108
7.4 Inheritance class diagram for `WriteToDisk` and `WriteToStdOut`. . . 109
7.5 Semantics DAG resulting from merging three Pipes. . . . 112
7.6 Semantics DAG resulting from the application in listing 7.3. . . . 114
7.7 Semantics DAG resulting from connecting multiple Pipes. . . . 114
7.8 Parallel execution DAG resulting from the application in listing 7.3. 115

8.1 Scalability and execution times for Word Count application in PiCo. 121
8.2 Scalability and execution times for Stock Pricing application in PiCo. 122
8.3 Comparison on best execution times for Word Count and Stock Pricing reached by Spark, Flink and Pico. 124
8.4 Scalability and execution times for Stream Stock Pricing application in PiCo. 128

List of Tables

2.1 Collective communication patterns among `ff_dnodes`. 28

4.1 Batch processing. 62
4.2 Stream processing comparison between Google Dataflow, Storm, Spark and Flink. 63

5.1 Pipelines 70
5.2 Core operator families. 72
5.3 Operator types. 76

7.1 the `Pipe` class API. 104
7.2 Operator constructors. 106

8.1 Decomposition of execution times and scalability highlighting the bottleneck on ReadFromFile operator in the Word Count benchmark. 123
8.2 Decomposition of execution times and scalability highlighting the bottleneck on ReadFromFile operator in the Stock Pricing benchmark. 123
8.3 Execution configurations for tested tools. 124
8.4 Average, standard deviation and coefficient of variation on 20 runs for each benchmark. Best execution times are highlighted. 126
8.5 User's percentage usage of all CPUs and RAM used in MB, referred to best execution times. 127
8.6 Decomposition of execution times and scalability highlighting the bottleneck on ReadFromSocket operator. 129
8.7 Flink, Spark and PiCo best average execution times, showing also the scalability with respect to the average execution time with one thread. 130
8.8 Average, standard deviation and coefficient of variation on 20 runs of the stream Stock Pricing benchmark. Best execution times are highlighted. 131
8.9 User's percentage usage of all CPUs and RAM used in MB, referred to best execution times. 131
8.10 Stream Stock Pricing: Throughput values computed as the number of input stock options with respect to the best execution time. . . . 131

List of Abbreviations

ACID	**A**tomicity, **C**onsistency, **I**solation, **D**urability
API	**A**pplication **P**rogramming **I**nterface
AVX	**A**dvanced **V**ector **E**xtensions
CSP	**C**ommunicating **S**equential **P**rocesses
DSL	**D**omain **S**pecific **L**anguage
DSM	**D**istributed **S**hared **M**emory
DSP	**D**igital **S**ignal **P**rocessor
EOS	**E**nd **O**f **S**tream
FPGA	**F**ield-**P**rogrammable **G**ate **A**rray
GPU	**G**raphics **P**rocessing **U**nit
JIT	**J**ust **I**n **T**ime
JVM	**J**ava **V**irtual **M**achine
MIC	**M**any **I**ntegrated **C**ore
MPI	**M**essage **P**assing **I**nterface
NGS	**N**ext **G**eneration **S**equencing
PGAS	**P**artitioned **G**lobal **A**ddress **S**pace
PiCo	**Pi**peline **Co**mposition
RDD	**R**esilient **D**istributed **D**atasets
RDMA	**R**emote **D**irect **M**emory **A**ccess
RMI	**R**emote **M**ethod **I**nvocation
SPMD	**S**ingle **P**rogram **M**ultiple **D**ata
STL	**S**tandard **T**emplate **L**ibrary
TBB	**T**hreading **B**uilding **B**locks

Alla mia famiglia

Chapter 1

Introduction

Big Data is becoming one of the most (ab)used buzzword of our times. In companies, industries, academia, the interest is dramatically increasing and everyone wants to "do Big Data", even though its definition or role in analytics is not completely clear. Big Data has been defined as the "3Vs" model, an informal definition proposed by Beyer and Laney [32, 90] that has been widely accepted by the community:

> *"Big data is high-Volume, high-Velocity and/or high-Variety information assets that demand cost-effective, innovative forms of information processing that enable enhanced insight, decision making, and process automation."*

In this definition, *Volume* refers to the amount of data that is generated and stored. The size of the data determines whether it can be considered big data or not. *Velocity* is about the speed at which the data is generated and processed. Finally, *Variety* pertains to the type and nature of the data being collected and processed (typically unstructured data).

The "3Vs" model has been further extended by adding two more V-features: *Variability*, since data not only are unstructured, but can also be of different types (e.g., text, images) or even inconsistent (i.e., corrupted), and *Veracity*, in the sense that the quality and accuracy of the data may vary.

From a high-level perspective, Big Data is about extracting knowledge from both structured and unstructured data. This is a useful process for big companies such as banks, insurance, telecommunication, public institutions, and so on, as well as for business in general.

Extracting knowledge from Big Data requires tools satisfying strong requirements with respect to programmability — that is, allowing to easily write programs and algorithms to analyze data — and performance, ensuring scalability when running analysis and queries on multicore or cluster of multicore nodes. Furthermore, they need to cope with input data in different formats, e.g. batch from data marts, live stream from the Internet or very high-frequency sources.

1.1 Results and Contributions

By studying and analyzing a large number of Big Data analytics tools, we identified the most representative ones: Spark [131], Storm [97], Flink [67] and Google Dataflow [5], that we surveyed in Sect. 3. The three major contributions from this study are the following:

An unifying semantics for Big Data analytics frameworks. Our focus is on understanding the expressiveness of their programming and execution

models, trying to come out with a set of common aspects that constitute the main ingredients of existing tools for Big Data analytics. In Chapter. 4, they will be systematically arranged in a single Dataflow-based formalism organized as a stack of layers. This will serve a number of objectives, such as the definition of semantics of existing Big Data frameworks, their precise comparison in terms of their expressiveness, and eventually the design of new frameworks. As we shall see, this approach also makes it possible to uniformly capture different data access mechanisms, such as batch and stream, that are typically claimed as distinguished features, thus making it possible to directly compare data-processing applications written in all the mainstream Big Data frameworks, including Spark, Flink, Storm, Google Dataflow.

As result of this analysis, we provide a layered model that can represent tools and applications following the Dataflow paradigm and we show how the analyzed tools fit in each level. As far as we know, no previous attempt has been made to compare different Big Data processing tools, at multiple levels of abstraction, under a common formalism.

A new data-model agnostic DSL for Big Data. We advocate a new Domain Specific Language (DSL), called **Pi**peline **Co**mposition (PiCo), designed over the presented layered Dataflow conceptual framework (see Chapter 4). PiCo programming model aims at *easing the programming* and *enhancing the performance* of Big Data applications by two design routes: 1) unifying data access model, and 2) decoupling processing from data layout.

Simplified programming. Both design routes undertake the same goal, which is the raising of the level of abstraction in the programming and the execution model with respect to mainstream approaches in tools for Big Data analytics, which typically force the specialization of the algorithm to match the data access and layout. Specifically, data transformation functions (called *operators* in PiCo) exhibit a different functional types when accessing data in different ways. For this reason, the source code should be revised when switching from one data model to the next. This happens in all the above mentioned frameworks and also in the abstract Big Data architectures, such as the Lambda [86] and Kappa architectures [84]. Some of them, such as the Spark framework, provide the runtime with a module to convert streams into micro-batches (Spark Streaming, a library running on Spark core), but still a different code should be written at user-level. The Kappa architecture advocates the opposite approach, i.e., to "streamize" batch processing, but the streamizing proxy has to be coded. The Lambda architecture requires the implementation of both a batch-oriented and a stream-oriented algorithm, which means coding and maintaining two codebases per algorithm.

PiCo fully decouples algorithm design from data model and layout. Code is designed in a fully functional style by composing stateless *operators* (i.e., transformations in Spark terminology). As discussed in Chapter 5, all PiCo operators are polymorphic with respect to data types. This makes it possible to 1) re-use the same algorithms and pipelines on different data models (e.g., streams, lists, sets, etc); 2) reuse the same operators in different contexts, and 3) update operators without affecting the calling context, i.e., the previous and following stages in the pipeline. Notice that in other mainstream frameworks, such as Spark, the update of a pipeline by changing a transformation with another is not necessarily trivial, since it may require the development of an input and output proxy to adapt the new transformation for the calling context.

Enhance performance. PiCo has the ambition of exploring a new direction in the relationship between processing and data. Two successful programming paradigms are object-oriented and task-based approaches: they have been mainstream in sequential and parallel computing, respectively. They both tightly couple data and computing. This coupling supports a number of features at both language and

run-time level, e.g., data locality and encapsulation. At the runtime support level, they might be turned into crucial features for performance, such as increasing the likely access to the closest and fastest memory and the load balancing onto multiple executors. In distributed computing, the quest for methods to distribute data structures onto multiple executors is long standing. In this regard, the *MapReduce* paradigm [61] introduced some freshness in the research area by exploiting the idea of processing data partitions in the machine where they are stored together with a relaxed data parallel execution model. Data partitions are initially randomized across a distributed filesystem for load balancing, computed (Map phase), then shuffled to form other partitions with respect to a key and computed again (Reduce phase). The underneath assumption is that the computing platform is a cluster. All data is managed by way of distributed objects.

As mentioned, PiCo aims at fully abstracting data layout from computing. Beyond programmability, we believe the approach could help in enhancing performance for two reasons. First, because the principal overhead of modern computing is data movement across the memory hierarchy and the described exploitation of locality is really effective for embarrassingly parallel computation only—being an instance of the *Owner-Computes Rule* (in the case of output data decomposition, the Owner-Computes Rule implies that the output is computed by the process to which the output data is assigned.). This argument, however, falls short in supporting locality for streaming, even when streaming is arranged in sliding windows or micro-batches and processed in a data-parallel fashion. The problem is that the Owner-Computes Rule does not help in minimizing data transfers, since anyway data flows from one of more sources to one or more sinks. For this, a scalable runtime support for stream and batch computing cannot be simply organized as a monolithic master-worker network, as it happen for mainstream Big Data frameworks. As described in Chapter 6, the network of processes and threads implementing the runtime support of PiCo is generated from the application by way of structural induction on operators, exploiting a set of basic, compositional runtime support patterns, namely FastFlow patterns (see Sect. 2.4.1). FastFlow [56] is an open source, structured parallel programming framework supporting highly efficient stream parallel computation while targeting shared memory multicores. Its efficiency mainly comes from the optimized implementation of the base communication mechanisms and from its layered design. It provides a set of ready-to-use, parametric algorithmic skeletons modeling the most common parallelism exploitation patterns, which may be freely nested to model more complex parallelism exploitation patterns. It is realized as a C++ pattern-based parallel programming framework aimed at simplifying the development of applications for (shared-memory) multi-core and GPGPU platforms. Moreover, PiCo relies on the FastFlow programming model, based on the decoupling of data from synchronizations, where only pointers to data are effectively moved through the network. The very same runtime can be used on a shared memory model (as in the current PiCo implementation) as well as in distributed memory or on Partitioned Global Address Space (PGAS) or Distributed Shared Memory (DSM), thus allowing a high portability at the cost of just managing communication among actors, which is left to the runtime.

A fluent C++14 DSL To the best of our knowledge, the only other ongoing attempt to address Big Data analytics in C++ is Thrill (see Sect. 3.3), which is basically a C++ Spark clone with a high number of primitives in contrast with PiCo, in which we used a RISC approach. The Grappa framework [98] is a platform for data-intensive applications implemented in C++ and providing a Distributed Shared Memory (DSM) as the underlying memory model, which provides fine-grain access to data anywhere in the system with strong consistency guarantees. It is no more active under development.

PiCo is specifically designed to exhibit a functional style over C++14 standard by defining a library of purely functional data transformation operators exhibiting

1) a well-defined functional and parallel semantics, 2) a fluent interface based on method chaining to relay the instruction context to a subsequent call [70]. In this way, the calling context is defined through the return value of a called method and is self-referential, where the new context is equivalent to the last context. In PiCo, the chaining is terminated by a special method, namely the `run()` method, which effectively executes the pipeline. The fluent interface style is inspired by C++'s *iostream* >> operator for passing data across a chain of operators.

Many of the mainstream frameworks for Big Data analytics are designed on top of Scala/Java, which simplifies the distribution and execution of remote objects, thus the mobility of code in favor of data locality. This choice has its own drawbacks, consider for instance a lower stability caused by Java's dynamic nature. By using C++, it is possible to directly manage memory as well as data movement, which represents one of the major performance issues because of data copying. In PiCo, no unnecessary copy is done and, thanks to the FastFlow runtime and the FastFlow allocator, it is possible to optimize communications (i.e., only pointers are moved) as well as manage allocations in order to reuse as much memory slots as possible. Moreover, the FastFlow allocator can be used also in a distributed memory scenario, thus becoming fundamental for streaming applications.

A last consideration about the advantage of using C++ instead of Java/Scala is that C++ allows a direct offloading of user-defined kernels to GPUs, thus leaning also towards specialized computing with DSPs, reconfigurable computing with FPGA, but also specialized networking co-processors, such as Infiniband RDMA verbs (which is a C/C++ API). Furthermore, Kernel offloading to GPUs is a feature already present in FastFlow that can be done with no need by the user to care about data management from/to the device [12].

1.2 Limitations and Future Work

PiCo has been designed to be implemented on cache-coherent multicores, GPUs, and distributed memory platforms. Since it is still in a prototype phase, only a shared memory implementation is provided but, thanks to the FastFlow runtime, it will be easy to 1) port it to run on distributed environment and 2) make PiCo exploit GPUs, since both features are already supported by FastFlow. PiCo is also still missing binary operator implementations, which are a work in progress. From the actual performance viewpoint, we aim to solve dynamic allocation contention problems we are facing in input generation nodes, as showed in Chapter 8, which limits PiCo scalability. In PiCo, we rely on the stability of a lightweight C++ runtime, in contrast to Java. We measured RAM and CPU utilization with the *sar* tool, which confirmed a lower memory consumption by PiCo with respect to the other frameworks when compared on batch application (Word Count and Stock Pricing) and stream application (Stock Pricing streaming). As another future work, we will provide PiCo with fault tolerance capabilities for automatic restore in case of failures. Another improvement for PiCo implementation on distributed systems would be to exploit the very same runtime on PGAS or DSMs, in order to still be able to use FastFlow's characteristic of moving pointers instead of data, thus allowing a high portability at the cost of just managing communication among actors in a different memory model, which is left to the runtime.

1.3 Plan of the Thesis

The thesis will proceed as follows:

- Chapter 2 provides technical background by reviewing the most common parallel computing platforms and then by introducing the problem of effective programming of such platforms exploiting high-level skeleton-based approaches on multicore and multicore cluster platforms; in particular we present FastFlow, which we use in this work to implement the PiCo runtime.
- Chapter 3 defines the Big Data terminology. It then continues with a survey of state-of-the-art of Big Data Analytics tools.
- Chapter 4 analyzes some well-known tools — Spark, Storm, Flink and Google Dataflow — by providing a common structure based on the *Dataflow model* describing all levels of abstraction, from the user-level API to the execution model. This Dataflow model is proposed as a stack of layers where each layer represents a dataflow graph/model with a different meaning, describing a program from what is exposed to the programmer down to the underlying execution model layer.
- Chapter 5 proposes a new programming model based on Pipelines and operators, which are the building blocks of PiCo programs, first defining the syntax of programs, then providing a formalization of the type system and semantics. The chapter also presents a set of use cases. The aim is to show the expressiveness of the proposed model, without using the current specific API in order to demonstrate that the model is independent from the implementation.
- Chapter 6 discusses how a PiCo program is compiled into a directed acyclic graph representing the parallelization of a given application, namely the parallel execution dataflow. This chapter shows how each operator is compiled into its corresponding parallel version, providing a set of optimization applicable when composing parallel operators.
- Chapter 7 provides a comprehensive description of the actual PiCo implementation, both at user API level and at runtime level. A complete source code example (Word Count) is used to describe how a PiCo program is compiled and executed.
- Chapter 8 provides a set of experiments based on examples defined in Section 5.4. We compare PiCo to Spark and Flink, focusing on expressiveness of the programming model and on performances in shared memory.
- Chapter 9 concludes the thesis.

1.4 List of Papers

This section provides the complete list of my publications, in reverse chronological order. Papers (i)–(vi) are directly related to this thesis. Papers (i) and (ii) introduces the *Dataflow Layered Model* presented in Chapter 4. Paper (iii) and (iv) are results of part of the work I did as an intern in the High Performance System Software research group in the Data Centric Systems Department at IBM T.J. Watson research center. During this period, the task I accomplished was to modify the Spark network layer in order to use the RDMA facilities provided by the IBM JVM. These papers refer to the optimization of the Spark shuffle strategy, in which a novel shuffle data transfer strategy is proposed, which dynamically adapts the prefetching to the computation.

Paper (v) introduces the *Loop-of-stencil-reduce* paradigm that generalizes the iterative application of a Map-Reduce pattern and discusses its GPU implementation in FastFlow. As already done in this work, it will be possible to apply the same techniques in the offloading of some PiCo operators to GPU (see Sect. 1.2). Paper (vi) presents a novel approach for functional-style programming of distributed-memory clusters with C++11, targeting data-centric applications. The proposed programming model explores the usage of MPI as communication library for building the runtime support of a non-SPMD programming model, as PiCo is. The paper itself is a preliminary study for the PiCo DSL.

Papers (vii)–(xiv) are not directly related to this thesis. Most of them are related to the design and optimization of parallel algorithms for Next Generation Sequencing (NGS) and Systems Biology, which is an application domain of growing interest for parallel computing. Also, most the algorithms in this domain are memory-bound, as are many problems in Big Data analytics; moreover, they make a massive use of pipelines to process NGS data, aspect that is reflected also in Big Data analytics. Even if not directly related to this thesis, working on the parallel implementation of NGS algorithms has definitely been an excellent motivation for this work.

Paper's list is divided into journal and conference papers list.

Journal Papers

(i) **Claudia Misale**, M. Drocco, M. Aldinucci, and G. Tremblay. A comparison of big data frameworks on a layered dataflow model. *Parallel Processing Letters*, 27(01):1740003, 2017

(ii) B. Nicolae, C. H. A. Costa, **Claudia Misale**, K. Katrinis, and Y. Park. Leveraging adaptative I/O to optimize collective data shuffling patterns for big data analytics. *IEEE Transactions on Parallel and Distributed Systems*, PP(99), 2016

(iii) M. Aldinucci, M. Danelutto, M. Drocco, P. Kilpatrick, **Claudia Misale**, G. Peretti Pezzi, and M. Torquati. A parallel pattern for iterative stencil + reduce. *Journal of Supercomputing*, pages 1–16, 2016

(iv) F. Tordini, M. Drocco, **Claudia Misale**, L. Milanesi, P. Liò, I. Merelli, M. Torquati, and M. Aldinucci. NuChart-II: the road to a fast and scalable tool for Hi-C data analysis. *International Journal of High Performance Computing Applications (IJHPCA)*, 2016

(v) **Claudia Misale**, G. Ferrero, M. Torquati, and M. Aldinucci. Sequence alignment tools: one parallel pattern to rule them all? *BioMed Research International*, 2014

(vi) M. Aldinucci, M. Torquati, C. Spampinato, M. Drocco, **Claudia Misale**, C. Calcagno, and M. Coppo. Parallel stochastic systems biology in the cloud. *Briefings in Bioinformatics*, 15(5):798–813, 2014

Conference Papers

(i) B. Nicolae, C. H. A. Costa, **Claudia Misale**, K. Katrinis, and Y. Park. Towards memory-optimized data shuffling patterns for big data analytics. In *IEEE/ACM 16th Intl. Symposium on Cluster, Cloud and Grid Computing, CCGrid 2016*, Cartagena, Colombia, 2016. IEEE

(ii) M. Drocco, **Claudia Misale**, and M. Aldinucci. A cluster-as-accelerator approach for SPMD-free data parallelism. In *Proc. of Intl. Euromicro PDP 2016: Parallel Distributed and network-based Processing*, pages 350–353, Crete, Greece, 2016. IEEE

(iii) F. Tordini, M. Drocco, **Claudia Misale**, L. Milanesi, P. Liò, I. Merelli, and M. Aldinucci. Parallel exploration of the nuclear chromosome conformation with NuChart-II. In *Proc. of Intl. Euromicro PDP 2015: Parallel Distributed and network-based Processing*. IEEE, Mar. 2015

(iv) M. Drocco, **Claudia Misale**, G. Peretti Pezzi, F. Tordini, and M. Aldinucci. Memory-optimised parallel processing of Hi-C data. In *Proc. of Intl. Euromicro PDP 2015: Parallel Distributed and network-based Processing*, pages 1–8. IEEE, Mar. 2015

(v) M. Aldinucci, M. Drocco, G. Peretti Pezzi, **Claudia Misale**, F. Tordini, and M. Torquati. Exercising high-level parallel programming on streams: a systems biology use case. In *Proc. of the 2014 IEEE 34th Intl. Conference on Distributed Computing Systems Workshops (ICDCS)*, Madrid, Spain, 2014. IEEE

(vi) **Claudia Misale**. Accelerating bowtie2 with a lock-less concurrency approach and memory affinity. In *Proc. of Intl. Euromicro PDP 2014: Parallel Distributed and network-based Processing*, Torino, Italy, 2014. IEEE. (Best paper award)

(vii) **Claudia Misale**, M. Aldinucci, and M. Torquati. Memory affinity in multithreading: the bowtie2 case study. In *Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES) – Poster Abstracts*, Fiuggi, Italy, 2013. HiPEAC

1.5 Funding

This work has been partially supported by the Italian Ministry of Education and Research (MIUR), by the EU-H2020 RIA project “Toreador” (no. 688797), the EU-H2020 RIA project “Rephrase” (no. 644235), the EU-FP7 STREP project “REPARA” (no. 609666), the EU-FP7 STREP project “Paraphrase” (no. 288570), and the 2015-2016 IBM Ph.D. Scholarship program.

Chapter 2

Technical Background

In this chapter, we provide a technical background to help the reader go through the topics of this thesis. We first review the most common parallel computing platforms (Sect. 2.2); then we introduce the problem of effective programming of such platforms exploiting high-level skeleton-based approaches on multicore and multicore cluster platforms; in particular we present FastFlow, which we use in this work to implement the PiCo runtime.

2.1 Parallel Computing

Computing hardware has evolved to sustain the demand for high-end performance along two basic ways. On the one hand, the increase of clock frequency and the exploitation of instruction-level parallelism boosted the computing power of the single processor. On the other hand, many processors have been arranged in multi-processors, multi-computers, and networks of geographically distributed machines.

Nowadays, after years of continual improvement of single core chips trying to increase instruction-level parallelism, hardware manufacturers realized that the effort required for further improvements is no longer worth the benefits eventually achieved. Microprocessor vendors have shifted their attention to thread-level parallelism by designing chips with multiple internal cores, known as multicores (or chip multiprocessors).

More generally, *parallelism* at multiple levels is now the driving force of computer design across all classes of computers, from small desktop workstations to large warehouse-scale computers.

2.2 Platforms

We briefly recap the review of existing parallel computing platforms from Hennessy and Patterson [73].

Following Flynn's taxonomy [69], we can define two main classes of architectures supporting parallel computing:

- Single Instruction stream, Multiple Data streams (SIMD): the same instruction is executed by multiple processors on different data streams. SIMD computers support *data-level parallelism* by applying the same operations to multiple items of data in parallel.
- Multiple Instruction streams, Multiple Data streams (MIMD): each processor fetches its own instructions and operates on its own data, and it targets task-level parallelism. In general, MIMD is more flexible than SIMD and thus more generally applicable, but it is inherently more expensive than SIMD.

We can further subdivide MIMD into two subclasses:

- Tightly coupled MIMD architectures, which exploit thread-level parallelism since multiple cooperating threads operate in parallel on the same execution context;
- Loosely coupled MIMD architectures, which exploit request-level parallelism, where many independent tasks can proceed in parallel "naturally" with little need for communication or synchronization.

Although it is a very common classification, this model is becoming more and more coarse, as many processors are nowadays "hybrids" of the classes above (e.g., GPU).

2.2.1 SIMD computers

The first use of SIMD instructions was in 1970s with the *vector supercomputers* such as the CDC Star-100 and the Texas Instruments ASC. Vector-processing architectures are now considered separate from SIMD machines: vector machines were processing the vectors one word at a time exploiting pipelined processors (though still based on a single instruction), whereas modern SIMD machines process all elements of the vector simultaneously [107].

Simple examples of SIMD computers are Intel Streaming SIMD Extensions (SSE) [76] for the x86 architecture. Processors implementing SSE (with a dedicated unit) can perform simultaneous operations on multiple operands in a single register. For example, SSE instructions can simultaneously perform eight 16-bit operations on 128-bit registers.

AVX-512 are 512-bit extensions to the 256-bit Advanced Vector Extensions SIMD instructions for x86 instruction set architecture (ISA) proposed by Intel in July 2013, and is supported in Intel's Xeon Phi x200 (Knights Landing) processor [77]. Programs can pack eight double precision or sixteen single precision floating-point numbers, or eight 64-bit integers, or sixteen 32-bit integers within the 512-bit vectors. This enables processing of twice the number of data elements that AVX/AVX2 can process with a single instruction and four times that of SSE.

Advantages of such approach are almost negligible overhead and little hardware cost; however, they are difficult to integrate into existing code, as this actually requires writing in assembly language.

One of the most popular platform specifically targeting data-level parallelism consists in the use of Graphics Processing Units (GPU) for general-purpose computing, known as General-Purpose computing on Graphics Processing Units (GPGPU). Moreover, using specific programming languages and frameworks (e.g., NVIDIA CUDA [102], OpenCL [85]) partially reduces the gap between high computational power and easiness of programming, though it still remains a low-level parallel programming approach, since the user has to deal with close-to-metal aspects like memory allocation and data movement between the GPU and the host platform.

Typical GPU architectures are not strictly SIMD architectures. For example, NVIDIA CUDA-enabled GPUs are based on multithreading, thus they support all types of parallelism; however, control hardware in these platforms is limited (e.g., no global thread synchronization), making GPGPU more suitable for data-level parallelism.

Symmetric shared-memory multiprocessors

Thread-level parallelism implies the existence of multiple program counters, hence being exploited primarily through MIMDs. Threads can also be used to support

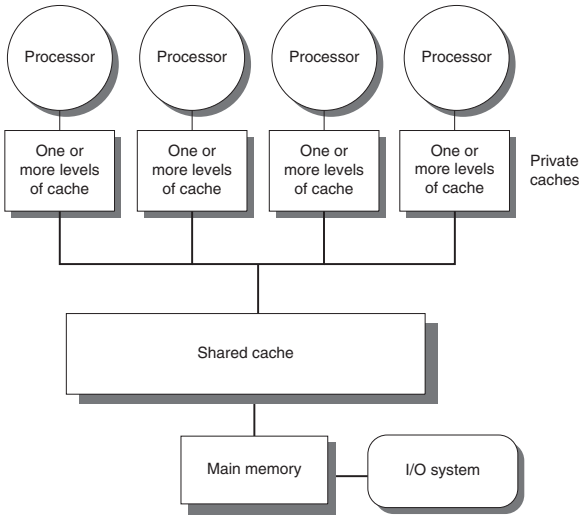


FIGURE 2.1: Structure of an SMP.

data-level parallelism, but some overhead is introduced at least by thread communication and synchronization. This overhead means the *grain size* (i.e., the ratio of computation to the amount of communication), which is a key factor for efficient exploitation of thread-level parallelism.

The most common MIMD computers are multiprocessors, defined as computers consisting of *tightly coupled* processors that share memory through a shared address space. Single-chip systems with multiple cores are known as *multicores*. *Symmetric* (shared-memory) multiprocessors (SMPs) typically feature small numbers of cores (nowadays from 12 to 24), where processors can share a single *centralized* memory, to which they all have equal access to (Fig. 2.1). In multicore chips, the memory is effectively centralized, and all existing multicores are SMPs. SMP architectures are also sometimes called uniform memory access (UMA) multiprocessors, arising from the fact that all processors have a uniform latency from memory, even if the memory is organized into multiple banks.

The alternative design approach consists of multiprocessors with physically distributed memory, called Distributed-Memory Systems (DSM). To support larger numbers of processors, memory must be distributed rather than centralized; otherwise, the memory system would not be able to support the bandwidth demands of processors without incurring excessively long access latency. Such architectures are known as nonuniform memory access (NUMA), since the access time depends on the location of a data word in memory.

Memory consistency model

The memory model, or memory consistency model, specifies the values that a shared variable read in a multithreaded program is allowed to return. The memory model affects programmability, performance and portability by constraining the transformations that any part of the system may perform. Moreover, any part of the system (hardware or software) that transforms the program must specify a memory model, and the models at the different system levels must be compatible [[37, 115]]. The most basic model is the *Sequential Consistency*, in which all instructions executed by threads in a multithreaded program appear in a total order that is consistent with the program order on each thread [88]. Some programming languages offer sequentially consistency in a multiprocessor environment: in C++11, all shared variables can be declared as atomic types with default memory ordering constraints. In Java, all shared variables can be marked as volatile.

For performance gains, modern CPUs often execute instructions out of order to fully utilize resources. Since the hardware enforces instructions integrity, it can be noticed in a single thread execution. However, in a multithreaded execution, reordering may lead to unpredictable behaviors.

Different CPU families have different memory models, thus different rules concerning memory ordering, but also the compiler optimizes the code by reordering instructions. To guarantee Sequential Consistency, you must consider how to prevent memory reordering. Ways can be lightweight synchronizations or fences, full fence, or acquire/release semantics. Sequential Consistency restricts many common compiler and hardware optimizations and to overcome the performance limitations of this memory model, hardware vendors and researchers have proposed several relaxed memory models reported in [4].

In lock-free programming for multicore (or any symmetric multiprocessor), sequential consistency must be ensured by preventing memory reordering. Herlihy and Shavit [74] provide the following definition of lock-free programs: "In an infinite execution, infinitely often some method call finishes". In other words, as long as the program is able to keep calling lock-free operations, the number of completed calls keeps increasing. It is algorithmically impossible for the system to lock up during those operations. C/C++ and Java provide portable ways of writing lock-free programs with sequential consistency or even weaker consistency models.

For such purpose of writing sequentially consistent multithreaded programs, C++ introduced the atomics standard library, which provides fine-grained atomic operations [37] that guarantee Sequential Consistency only for data race free programs, hence allowing also lockless concurrent programming. Each atomic operation is indivisible with regards to any other atomic operation that involves the same object. Atomic objects are free of data races.

Cache coherence and false sharing

SMP machines usually support the caching of both shared and private data, reducing the average access time as well as the required memory bandwidth. Unfortunately, caching shared data introduces a new problem because the view of memory held by two different processors is through their individual caches, which could end up seeing two different values. This problem is generally referred to as the *cache coherence problem* and several protocols have been designed to guarantee cache coherence. In cache-coherent SMP machines, *false sharing* is a subtle source of cache miss, which arises from the use of an invalidation-based coherence algorithm. False sharing occurs when a block is invalidated (and a subsequent reference causes a miss) because some word in the block, other than the one being read, is written into. In a false sharing miss, the block is shared, but no word in the cache is actually shared, and the miss would not occur if the block size were a single word.

2.2.2 Manycore processors

Manycore processors are specialized multi-core processors designed to exploit a high degree of parallelism, containing a large number of simpler, independent processor cores, often called *Hardware accelerators*. A manycore processor contains at least tens of cores and usually distributed memory, which are connected (but physically separated) by an interconnect that has a communication latency of multiple clock cycles [110]. A multicore architecture equipped with hardware accelerators is a form of *heterogeneous architecture*. Comparing multicore to manycore processors, we can characterize them as follows:

- Multicore: a symmetric multiprocessing (SMP) architecture containing tightly coupled identical cores that (usually) share all memory, where caches are hardware cache coherent.
- Manycore: are specialized multicore processors designed for a high degree of parallel processing, containing a large number of simpler, independent processor cores (e.g., tens up to thousands) with a reduced cache coherency to increase performance. A manycore architecture may run a specific OS, as for Intel MIC coprocessor Xeon Phi.

As the core count increases, hardware cache coherency is unlikely to be sustained [45]. Accelerators share some features: they are typically slower with respect to multicore CPUs, the high performance is obtained by high level of parallelism, data transfer from host to device is slower than memory bandwidth in host processors and need data locality to obtain good performance. Following we provide a description of some of those accelerators, used regularly in High-Performance Computing (HPC).

- GPUs: Graphics Processing Units include a large number of small processing cores (from hundreds to thousands) in an architecture optimized for highly parallel workloads, paired with dedicated high performance memory. They are accelerators, used from a general purpose CPU, that can deliver high performance for some classes of algorithms. Programming GPUs requires using the Nvidia CUDA programming model or the OpenCL cross-platforms programming model.
- Intel Xeon Phi: Xeon Phi is a brand name given to a series of massively-parallel multicore co-processors designed and manufactured by Intel, targeted at HPC. An important component of the Intel Xeon Phi coprocessor's core is its Vector Processing Unit (VPU). In the second generation MIC architecture product from Intel, Knights Landing, each core have two 512-bit vector units and will support AVX-512 SIMD instructions, specifically the Intel AVX-512 Foundational Instructions (AVX-512F) with Intel AVX-512 Conflict Detection Instructions (AVX-512CD), Intel AVX-512 Exponential and Reciprocal Instructions (AVX-512ER), and Intel AVX-512 Prefetch Instructions (AVX-512PF) [77]. The Xeon Phi co-processor family can be programmed with OpenMP, OpenCL, MPI and TBB using offloading directives, Cilk/Cilk Plus and specialised versions of Intel's Fortran, C++ and math libraries.

  Differently from GPUs, which exist in different models with typically thousands of cores, the co-processor comprises up to sixty-one Intel cores that can run at most 4 hyper threads per core, and are connected by a high performance on-die bidirectional interconnect [80].

  Intel Knights Landing co-processor belongs to the Xeon family and is Xeon Phi's successor, increasing the total number of cores from 61 to 64, always exploiting 4 hyper threads per core. Knights Landing is made up of 36 tiles. Each tile contains two CPU cores and two VPUs (vector processing units) per core (total of four per tile). Unlike GPUs and previous Xeon Phi manycore cards, which functioned solely as co-processors, the new Knights Landing is more than just an accelerator: it is designed to self-boot and can control an operating system as the native processor.
- FPGA: Field Programmable Gate Arrays (FPGAs) are semiconductor devices based around a matrix of configurable logic blocks (CLBs) connected via programmable interconnects. FPGAs can be reprogrammed to desired application or functionality requirements after manufacturing. The FPGA configuration is generally specified using a hardware description language (HDL), similar to that used for an application-specific integrated circuit (ASIC).

2.2.3 Distributed Systems, Clusters and Clouds

A distributed system is a model in which components located on networked computers communicate and coordinate their actions by passing messages. It can be composed by various hardware and software components, thus exploiting homogeneous and heterogeneous architectures. An important aspect of distributed computing architectures is the method of communicating and coordinating work among concurrent processes. Through various message passing protocols, processes may communicate directly with one another, typically in a master-slave relationship, operating to fulfill the same objective. This master-slave relationship reflects the typical execution model of Big Data analytics tools on distributed systems: the master process is assigned to coordinate slaves execution (typically called workers) and coordination. Slaves may execute the same program or part of the computation, and they communicate with each other and with the master by message passing. Master and slaves may also exchange data via serialization. From the implementation viewpoint, those architectures are typically programmed by exploiting the Message Passing Interface (MPI) [105], a language-independent communication protocol used for programming parallel computers, as well as a message-passing API that supports point-to-point and collective communication by means of directly callable routines. Many general-purpose programming languages have bindings to MPI's functionalities, among which: C, C++, Fortran, Java and Python. Moving to the Big Data world, tools are often implemented in Java in order to easily exploit facilities such as Java RMI API [103], which performs remote method invocation supporting direct transfer of serialized Java classes and distributed garbage collection. These models for communication will be further investigated in Section 2.3.3.

In contrast with shared-memory architectures, *clusters* look like individual computers connected by a network. Since each processor has its own address space, the memory of one processor cannot be accessed by another processor without the assistance of software protocols running on both processors. In such design, message-passing protocols are used to communicate data among processors. Clusters are examples of loosely coupled MIMDs. These large-scale systems are typically used for *cloud computing* with a model that assumes either massive numbers of independent requests or highly parallel intensive compute tasks.

There are two classes of large-scale distributed systems:

1) *Private* clouds, in particular multicore clusters, which are inter networked — possibly heterogeneous — multicore devices.

2) *Public* clouds, which are (physical or virtual) infrastructures offered by providers in the form of inter networked clusters. In the most basic public cloud model, providers of IaaS (Infrastructures-as-a-Service) offer computers — physical or (more often) virtual machines — and other resources on-demand. Public IaaS clouds can be regarded as *virtual* multicore clusters. The public cloud model encompasses a *pay-per-use* business model. End users are not required to take care of hardware, power consumption, reliability, robustness, security, and the problems related to the deployment of a physical computing infrastructure.

2.3 Parallel Programming Models

Shifting from sequential to parallel computing, a trend largely motivated by the advent of multicore platforms, does not always translate into greater CPU performance: multicores are small-scale but full-fledged parallel machines and they retain many of their usage problems. In particular, sequential code will get no performance benefits from them: a workstation equipped with a quad-core CPU but running sequential code is wasting $\frac{3}{4}$ of its computational power. Developers are then facing

the challenge of achieving a trade-off between performance and human productivity (total cost and time to solution) in developing and porting applications to multicore and parallel platforms in general.

Therefore, effective parallel programming happens to be a key factor for efficient parallel computing, but efficiency is not the only issue faced by parallel programmers: writing parallel code that is portable on different platforms and maintainable are tasks that programming models should address.

2.3.1 Types of parallelism

Types of parallelisms can be categorized in four main classes:

- *Task Parallelism* consists of running the same or different code (task) on different executors (cores, processors, etc.). Tasks are concretely performed by threads or processes, which may communicate with one another as they execute. Communication takes place usually to pass data from one thread to the next as part of a graph. Task parallelism does not necessarily concern stream parallelism, but there might cases in which the computation of each single item in a input stream embeds an independent (thus parallel) task, that can efficiently be exploited to speedup the application. The *farm* pattern is a typical representation of such class of patterns, as we will describe in next sections.
- *Data Parallelism* is a method for parallelizing a single task by processing independent data elements in parallel. The flexibility of the technique relies upon stateless processing routines implying that the data elements must be fully independent. Data Parallelism also supports *Loop-level Parallelism* where successive iterations of a loop working on independent or read-only data are parallelized in different flows-of-control (according to the model *co-begin/co-end*) and concurrently executed.
- *Stream Parallelism* is a method for parallelizing the execution (aka. filtering) of a stream of tasks by segmenting each task into a series of *sequential*[1] or *parallel* stages. This method can be also applied when there exists a *total* or *partial* order, respectively, in a computation preventing the use of data or task parallelism. This might also come from the successive availability of input data along time (e.g., data flowing from a device). By processing data elements in order, local state may be either maintained in each stage or distributed (replicated, scattered, etc.) along streams. Parallelism is achieved by running each stage simultaneously on *subsequent* or *independent* data elements.
- *DataFlow Parallelism* is a programming paradigm modeling a (parallel) program as a directed graph where operations are represented by nodes, and edges model data dependencies. Nodes in this graph represent functional unit of computation over data item (tokens) flowing on edges. In contrast with procedural (imperative) programming model, a DataFlow program is described as a set of operations and connections among them, defined as a set of input and output edges. Operations execute as soon as all their input edges have an incoming token available and operations without a direct dependence can be run in parallel. This model, formalized by Kahn [83], is one of the main building blocks of the model proposed in this thesis. A further description is provided in subsequent sections.

Since Data Parallelism and Dataflow Parallelism are important aspects in this work, we explore them further in the next two paragraphs.

[1] In the case of total sequential stages, the method is also known as *Pipeline Parallelism*.

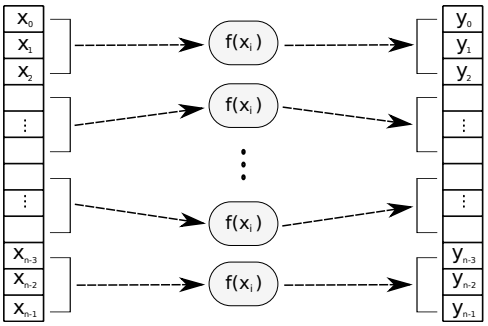


FIGURE 2.2: A general representation of a *Map* data parallel pattern. Input data structure is partitioned according to the number of workers. Business logic function is replicated among each worker.

Data parallelism

A data-parallel computation performs the same operation on different items of a given data structure at the same time. Opposite to task parallelism, which emphasizes the parallel nature of the computation, data parallelism stresses the parallel nature of the data[2].

Formally, a data parallel computation is characterized by partitioning data structures and function replication: a common partitioning approach divides input data between the available processors. Depending on the algorithm, there might be cases in which data dependencies exist among partitioned subtasks: many data-parallel programs may suffer from bad performance and poor scalability because of a high number of data dependencies or a low amount of inherent parallelism.

Here we present two widely used instances of data parallel patterns, namely the *map* and the *reduce* patterns, that are also the most often used patterns in Big Data scenarios. Other data parallel patterns exist, which basically permit to apply higher-order functions to all the elements of a data structure. Among them we can mention the *fold* pattern and the *scan* (or prefix sum) pattern. There is also the *stencil* pattern, which is a generalization of the *map* pattern, and under a functional perspective both patterns are similar, but the stencil encompasses all those situations that require data exchange among workers [12].

Map

The map pattern is a straightforward case of data parallel paradigm: given a function f that expresses an application's behavior, and a data collection X of known size (e.g., a vector with n elements), a map pattern will apply the function f to all the elements $x_i \in X$:

$$y_i = f(x_i) \qquad \forall i = 0, \ldots, n-1 \ .$$

Each element y_i of the resulting output vector Y is the result of a parallel computation. A pictorial representation of the map pattern is presented in Fig. 2.2.

Notable examples that naturally fit this paradigm include some vector operations (*scalar-vector* multiplication, vector sum, etc.), *matrix-matrix* multiplication (in

[2]We denote this difference by using a zero-based numbering when indexing data structures in data parallel patterns.

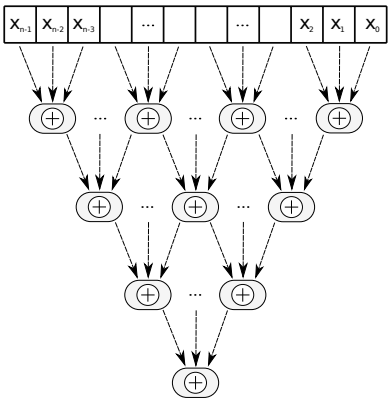


FIGURE 2.3: A general *Reduce* pattern: the reduction is performed in parallel organizing workers in a tree-like structure: each worker computes the function $\oplus$ on the results communicated by the son worker. The root worker delivers the reduce result.

which the result matrix is partitioned, the input matrices replicated), the *Mandelbrot set* calculation, and many others.

Reduce

A reduce pattern applies an *associative* binary function ($\oplus$) to all the elements of a data structure. Given a vector X of length n, the reduce pattern computes the following result:

$$x_0 \oplus x_1 \oplus \ldots \oplus x_{n-1} \quad .$$

In Figure 2.3, we report an example of a possible implementation of a reduce pattern, in which the reduce operator is applied over the assigned partition of the vector in parallel, and each single result is then used to compute a global reduce. In a tree-like organization, leaf nodes compute their local reduce and then propagate results to parent nodes; the root node delivers the reduce result.

While the reduce pattern is straightforward for associative and commutative operators (e.g., addition and multiplication of real numbers is associative), this is no more true for floating point operators. Floating-point operations, as defined in the IEEE-754 standard, are not associative [1]. Studies demonstrated how, on massively multi-threaded systems, the non-deterministic nature of how machine floating-point operations are interleaved, combined with the fact that intermediate values have to be rounded or truncated to fit in the available precision leads to non-deterministic numerical error propagation [129].

The composition of a *map* step and a *reduce* step generates the *Map+Reduce* pattern, in which a function is first applied, in parallel, to all elements of a data structure, and then the results from the map phase are merged using some reduction function (see Figure 2.4). This is an important example of the *composability* allowed by parallel patterns, which permits to define a whole class of algorithms for data-parallel computation.

The functional composition of these two parallel patterns is the basis of Google's *MapReduce* distributed programming model and framework [61], which exploits *key-value* pairs to compute problems that can be parallelized by mapping a function

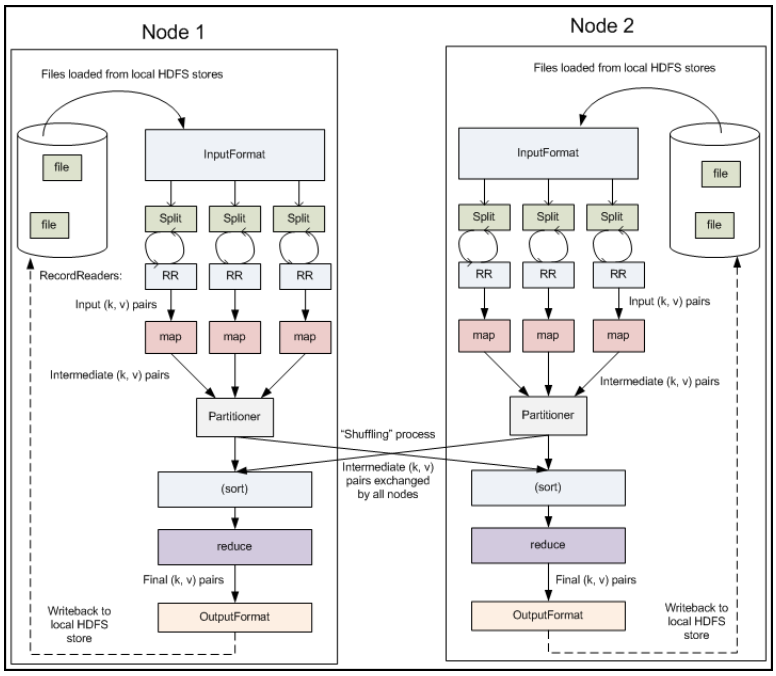


FIGURE 2.4: A *Map+Reduce* pattern.

over a given dataset and then combining the results. Likely, it is the largest pattern framework in use, and spun off different open-source implementations, such as Hadoop [130] and Phoenix [109].

Both Map and Reduce patterns are exploited in PiCo and they are further described in Chapter 5.

2.3.2 The Dataflow Model

Dataflow Process Networks are a special case of *Kahn Process Networks*, a model of computation that describes a program as a set of concurrent processes communicating with each other via FIFO channels, where reads are blocking and writes are non-blocking [83]. In a Dataflow process network, a set of *firing rules* is associated with each process, called *actor*. Processing then consists of "repeated firings of actors", where an actor represents a *functional* unit of computation over *tokens*. For an actor, to be *functional* means that firings have no side effects—thus actors are stateless—and the output tokens are pure functions of the input tokens. The model can also be extended to allow stateful actors.

A Dataflow network can be executed mainly by two classes of execution, namely *process-based* and *scheduling-based*—other models are flavors of these two. The process-based model is straightforward: each actor is represented by a process that communicates via FIFO channels. In the scheduling-based model—also known as dynamic scheduling—a scheduler tracks the availability of tokens in input to actors and schedules enabled actors for execution; the atomic unit being scheduled is referred as a *task* and represents the computation performed by an actor over a single set of input tokens.

Actors

A Dataflow actor consumes input tokens when it *fires* and then produces output tokens; thus it repeatedly fires on tokens arriving from one or more streams. The

function mapping input to output tokens is called the *kernel* of an actor.[3] A *firing rule* defines when an actor can fire. Each rule defines what tokens have to be available for the actor to fire. Multiple rules can be combined to program arbitrarily complex firing logics (e.g., the *If* node).

Input channels

The kernel function takes as input one or more tokens from one or more input channels when a firing rule is activated. The basic model can be extended to allow for testing input channels for emptiness, to express arbitrary stream consuming policies (e.g., gathering from any channel).

Output channels

The kernel function places one or more tokens into one or more output channels when a firing rule is activated. Each output token produced by a firing can be replicated and placed onto each output channel (i.e., broadcasting) or sent to specific channels, in order to model arbitrarily producing policies (e.g., switch, scatter).

Stateful actors

Actors with state can be considered like objects (instead of functions) with methods used to modify the object's internal state. Stateful actors is an extension that allows side effects over *local* (i.e., internal to each actor) states. It was shown by Lee and Parks [91] that stateful actors can be emulated in the stateless Dataflow model by adding an extra feedback channel carrying the value of the state to the next execution of the kernel function on the next element of the stream and by defining appropriate firing rules.

In the next sections, we provide a description of various other low-level and high-level programming models and techniques.

2.3.3 Low-level approaches

Typically, low-level approaches provide the programmers only with primitives for flow-of-control management (creation, destruction), their synchronization and data sharing, which are usually accomplished in critical regions accessed in mutual exclusion (mutex). As an example, POSIX thread library can be used for this purpose. Parallel programming languages are usually extensions to well-established sequential languages, such as C/C++, Java or Fortran, where the coordination of multiple execution flows is either obtained by means of external libraries, linked at compile time to the application source code (e.g., *Pthreads*, *OpenMP*, *MPI*), or enriched with specific constructs useful to orchestrate the parallel computation, for instance consider the Java Concurrency API and the C++11 standard published in 2011, which introduced multithreaded programming. The well known report, authored by H. Boehm [36], provides specific arguments that a pure library approach, in which the compiler is designed independently of threading issues, cannot guarantee correctness of the resulting code. It is illustrated that there are very simple cases (concurrent modification, adjacent data rewriting and register promotion) in which

[3] The Dataflow Process Network model also seamlessly comprehends the Macro Dataflow parallel execution model, in which each process executes arbitrary code. Conversely, an actor's code in a classical Dataflow *architecture* model is typically a single machine instruction. In the following, we consider Dataflow and Macro Dataflow to be equivalent models.

a pure library-based approach seems incapable of expressing an efficient parallel algorithm.

Programming parallel complex applications in this way is certainly hard; tuning them for performance is often even harder due to the non-trivial effects induced by memory fences (used to implement mutex) on data replicated in core's caches.

Indeed, memory fences are one of the key sources of performance degradation in communication intensive (e.g., streaming) parallel applications. Avoiding memory fences means not only avoiding locks but also any kind of atomic operation in memory (e.g., Compare-And-Swap, Fetch-and-Add). While there exists several assessed fence-free solutions for *asynchronous symmetric* communications,[4] these results cannot be easily extended to *asynchronous asymmetric* communications[5], which are necessary to support arbitrary streaming networks. This ability is one of the core features of FastFlow (2.4.1).

A first way to ease the programmer's task and improve program efficiency consists in raising the level of abstraction of concurrency management primitives. As an example, threads might be abstracted out in higher-level entities that can be pooled and scheduled in user space possibly according to specific strategies to minimize cache flushing or maximize load balancing of cores. Synchronization primitives can be also abstracted out and associated to semantically meaningful points of the code, such as function calls and returns, loops, etc. Intel *Threading Building Block* (TBB) [78], *OpenMP* [106], and *Cilk* [47] all provide this kind of abstraction — each of them in its own way. This kind of abstraction significantly simplifies the hand-coding of applications but it is still too low-level to effectively (semi-)automatize the optimization of the parallel code. In the following paragraphs, we describe a bit more in detail a list of low level parallel programming frameworks and APIs.

POSIX Threads (or Pthreads) [40] are one of the most famous low-level parallel programming APIs for shared-memory environments, defined by the standard POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995). They are present in every Unix-like operating system (Linux, Solaris, Mac OS X, etc.) and other POSIX systems, giving access to OS-level threads and synchronization mechanisms. Since Pthreads is a C library, it can be used in C++ programs as well, though there have been much improvements in the C++11 standard aimed at facilitating shared-memory parallel programming from a low-level to a high-level parallel programming. We remark that ISO C++ is completely independent from POSIX and it is provided also in non-POSIX platforms.

Message Passing Interface (MPI) [105] is a language-independent communication protocol used for programming parallel computers, as well as a message-passing API that supports point-to-point and collective communication by means of directly callable routines. Many general-purpose programming languages have bindings to MPI's functionalities, among which: C, C++, Fortran, Java and Python.

Mainly targeted at distributed architectures, MPI offers specific implementations for almost any high-performance interconnection network. At the same time, shared-memory implementations exist, that allow the use of MPI even on NUMA and multi-processors systems.

MPI allows to manage synchronization and communication functionalities among a set of processes, and provides mechanisms to deploy a virtual topology of the system upon which the program is executing. These features, supported by a rich set of capabilities and functions, clearly require high programming and networking skills: MPI has long been the *lingua franca* of HPC, supporting the majority of all supercomputing work scientists and engineers have relied upon for the past two decades. Nonetheless, MPI is at a low level of abstraction for application writers:

[4] Single-Producer-Single-Consumer (SPSC) queues [89].
[5] Multiple-Producer-Multiple-Consumer queues (MPMC).

using MPI means programming at the *transport* layer, where every exchange of data has to be implemented through *sends* and *receives*, data structures must be manually decomposed across processors and every update of the data structure needs to be recast into a flurry of messages, synchronizations, and data exchange.

OpenMP [106] is considered by many the *de facto* standard API for shared-memory parallel programming. OpenMP is an extension that can be supported by C, C++ and Fortran compilers and defines an "accelerator-style" programming, where the main program is run sequentially while code is accelerated in specific points, running in parallel, using special preprocessor instructions known as `pragma`s. Compilers that do not support specific `pragma`s can ignore them, making an OpenMP program compilable and runnable on every system with a generic sequential compiler.

While Pthreads are low-level and require the programmer to specify every detail of the behavior of each thread, OpenMP allows to simply state which block of code should be executed in parallel, leaving to compiler and run-time system the responsibility to determine the details of the thread behavior. This feature makes OpenMP programs simpler to code, with a risk of unpredictable performance that strongly depends on compiler implementations and optimizations. For instance, in [111] OpenMP limitations have been demonstrated such as its inability to perform reductions on complex data types as well as scheduling issues on applications with irregular computations (such as the Mandelbrot benchmark).

OpenCL [85] is an API designed to write parallel programs that execute across heterogeneous architectures, and allows the users to exploit GPUs for general purpose tasks that can be parallelized. It is implemented by different hardware vendors such as Intel, AMD and NVIDIA, making it highly portable and allowing OpenCL code to be run on different hardware accelerators. OpenCL allows the implementation of applications onto FPGAs, allowing software programmers to write hardware-accelerated kernel functions in OpenCL C, an ANSI C-based language with additional OpenCL constructs. OpenCL represents an extension to C/C++ but must be considered a low-level language, focusing on low-level features management rather than high-level parallelism exploitation patterns. It has the capability to revert to the CPU for execution when there is no GPU in the system, and its portability makes it suitable for hybrid (CPU/GPU) or cloud based environments.

Java provides multi-threading and RPC support (Java RMI API) that can be used to write parallel applications for both shared memory and distributed memory architectures [103]. The original implementation depends on Java Virtual Machine (JVM) class representation mechanisms and it only supports making calls from one JVM to another. However, while being a high-level sequential language, parallel support is provided in a low-level fashion, possibly lower than OpenMP.

2.3.4 High-level approaches

Parallel programming has always been related to HPC environments, where programmers write parallel code by mean of low-level libraries that give complete control over the parallel application, allowing them to manually optimize the code in order to exploit at best the parallel architecture. This programming methodology has become unsuitable with the fast move to heterogeneous architectures, that encompass hardware accelerators, distributed shared-memory systems and cloud infrastructures, highlighting the need for proper tools to easily implement parallel applications.

It is widely acknowledged that the main problem to be addressed by a parallel programming model is *portability*: the ability to not only compile and execute the same code on different architectures and obtain the same top performance [58], but also — and even more complex — the challenge of performance portability, that is, implementing applications that scale on different architectures. A high-level

approach to parallel programming is a better way to go if we want to address this problem, so that programmers can build parallel applications and be sure that they will perform reasonably well on the wide choice of parallel architectures available today [114].

Attempts to raise the level of abstraction and reduce the programming effort date back to at least three decades. Notable results have been achieved by the *algorithmic skeleton* approach [48] (aka. *pattern-based* parallel programming), that has gained popularity after being revamped by several successful parallel programming frameworks. Despite some criticisms — mostly related to the limited amount of patterns that might not be sufficient to allow a decent parallelization of most algorithms — algorithmic skeletons success has been determined by the numerous advantages it has compared to traditional parallel programming frameworks.

2.3.5 Skeleton-based approaches

Algorithmic skeletons have been initially proposed by Cole [49] to provide predefined parallel computation and communication patterns, hiding parallelism management from the user. *Algorithmic skeletons* capture common parallel programming paradigms (e.g., Map+Reduce, ForAll, Divide&Conquer, etc.) and make them available to the programmer as high-level programming constructs equipped with well-defined functional and extra-functional semantics [11]. Ideally, algorithmic skeletons address the difficulties of parallel programming (i.e., concurrency exploitation, orchestration, mapping, tuning) moving them from the application design to development tools, by capturing and abstracting common paradigms of parallel programming and providing them with efficient implementations. This idea can be considered at the core of *structured parallel programming*: expressing the parallel code as a composition of simple "building blocks".

Literature review of skeleton-based approaches

Many skeletons have been proposed in literature in the last two decades covering many different usage schema of the three classes of parallelism, on top of both the message passing [50, 57, 113, 9, 16, 35, 108, 13, 61, 10] and shared memory [7, 78] programming models.

Here, we discuss some skeleton libraries that target C/C++ as their execution language, and mostly focus on parallel programming for multi-core architectures. For a broader survey of algorithmic skeletons, we refer to González-Vélez and Leyton's survey [71].

P^{3}L is one of the earliest proposal for pattern-based parallel programming [53]. P^3L is a skeleton-based coordination language that manages the parallel or sequential execution of C code. It comes with a proper compiler for the language, and uses implementation templates to compile the code into a target architecture. P^3L provides patterns for both stream parallelism and data parallelism.

SKELib [54] builds upon the contributions of P^3L by inheriting, among other features, the template system. It differs from P^3L because a coordination language is no longer used, and skeletons are provided as a C library. It only offers stream-based skeletons (namely farm and pipe patterns).

SkeTo [95] is a C++ library based on MPI that provides skeletons for distributed data structures, such as arrays, matrices, and trees. The current version is based on C++ expression templates, used to represent part of an expression where the template represents the operation and parameters represent the operands to which the operation applies.

SkePU [66] is an open-source skeleton programming framework for multi-core CPUs and multi-GPU systems. It is a C++ template library with data-parallel and task-parallel skeletons (map, reduce, map-reduce, farm) that also provides generic container types and support for execution on multi-GPU systems, both with CUDA and OpenCL.

SkelCL [116] is a skeleton library targeting OpenCL. It allows the declaration of skeleton-based applications hiding all the low-level details of OpenCL. The set of skeletons is currently limited to data-parallel patterns: map, zip, reduce and scan, and it is unclear whether skeleton nesting is allowed. A limitation might come from the library's target, which is restricted to the OpenCL language: it likely benefits from the possibility to run OpenCL code both on multi-core and on many-core architectures, but the window for tunings and optimizations is thus quite restricted.

Muesli — Muenster Skeleton Library [46] — is a C++ template library that supports shared-memory multi-processor and distributed architectures using MPI and OpenMP as underlying parallel engines. It provides data parallel patterns such as map, fold (i.e., reduce), scan (i.e., prefix sum), and distributed data structures like distributed arrays, matrices and sparse matrices. Skeleton functions are passed to distributed objects as pointers, since each distributed object has skeleton functions as internal member of the class itself. The programmer must explicitly indicate whether GPUs are to be used for data parallel skeletons, if available.

Intel Threading Building Blocks (**TBB**) [78] defines a set of high-level parallel patterns that permit to exploit parallelism independently from the underlying platform details and threading mechanisms. It targets shared-memory multi-core architectures, and exposes parallel patterns for exploiting both stream parallelism and data parallelism. Among them, the `parallel_for` and `parallel_foreach` methods may be used to parallelize independent invocation of the function body of a for loop, whose number of iterations is known in advance. C++11 *lambda* expression can be used as arguments to these calls, so that the loop body function can be described as part of the call, rather than being separately declared. The `parallel_for` uses a divide-and-conquer approach, where a range $[0, num\_iter)$ is splitted into sub-ranges and each sub-range r can be processed as a separate task using a serial for loop.

GrPPI [62] is a generic high-level pattern interface for stream-based C++ applications. Thanks to its high-level C++ API, this interface allows users to easily expose parallelism in sequential applications using already existing parallel frameworks, such as C++ threads, OpenMP and Intel TBB. It is implemented using C++ template meta-programming techniques to provide interfaces of a generic, reusable set of parallel patter patterns without incurring in runtime overheads. GrPPI targets the following stream parallel processing patterns: *Pipeline*, *Farm*, *Filter* and *Stream-Reduce*. Parallel versions and to implement the proposed interfaces are implemented by we leveraging C++11 threads and OpenMP, and the pattern-based parallel framework Intel TBB. These patterns are parametrized by user defined lambdas and they are nestable.

FastFlow [15] is a parallel programming framework originally designed to support streaming applications on cache-coherent multicore platforms. It will be described more in detail in Section 2.4.1, since it is part of the PiCo runtime.

2.3.6 Skeletons for stream parallelism

A stream-parallel program can be naturally represented as a graph of independent *stages* (kernels or filters) which communicate over data channels. Conceptually, a streaming computation represents a sequence of transformations on the data streams

in the program. Parallelism is achieved by running each stage of the graph simultaneously on *subsequent* or *independent* data elements. Several skeletons exist that support stream-parallel programming.

The *Pipeline* skeleton is one of the most widely-known pattern (see Fig. 2.5). Parallelism is achieved by running each stage simultaneously on subsequent data elements, with the pipeline's throughput being limited by the throughput of the slowest stage. It is typically used to model computations expressed in *stages*, and in the general case a pipeline has at least two stages. Given a sequence $x_1, \ldots, x_k$ of input tasks and the simplest form of a pipeline with three stages, the computation on each single task x_i is expressed as the composition of three functions f, z and g, where the second stage (function z) executes on the results of the application of the first stage, $z(f(x_i))$, and third stage applies the function z on the output of the second stage: $g(z(f(x_i)))$.

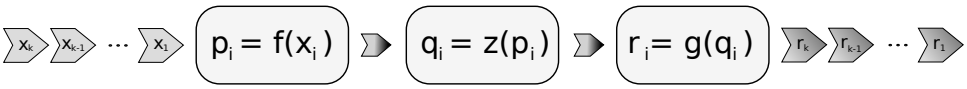


FIGURE 2.5: A general *Pipeline* pattern with three stages.

The parallelization is obtained by concurrently executing all the stages onto different consecutive items of the input stream. In the general form, a pipeline with stages $s_1, \ldots, s_m$ computes the output stream

$$s_m(s_{m-1}(\ldots s_2(s_1(x_k))\ldots)), \ldots, s_m(s_{m-1}(\ldots s_2(s_1(x_1))\ldots)) \ .$$

The sequential code has to be described in terms of function composition, where the output of each stage is sent to the next one, respecting the function ordering. The semantic of the pipeline paradigm ensures that all stages will execute in parallel.

The *Farm* skeleton models functional replication and consists in running multiple independent stages in parallel, each operating on different tasks of the input stream, thus it is often used to model *embarrassingly* parallel computations.

The *Farm* skeleton can be better understood as a three stage — *emitter*, *workers*, *collector* — pipeline (see Fig. 2.6). The emitter dispatches stream items to a set of workers, which independently elaborate different items. The output of the workers is then gathered by the collector into a single stream. More complex combinations of both patterns are possible, such as a *pipeline of farms*, where each stage of the pipeline is, in fact, a farm (as in Figure 2.7).

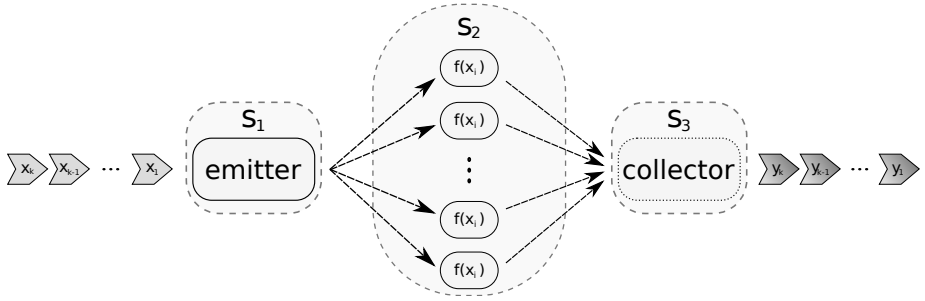


FIGURE 2.6: A simple *Farm* pattern with an *optional* collector stage, whose stroke is dashed. Background light-gray boxes show that a farm pattern can be embedded into a three-stages pipeline.

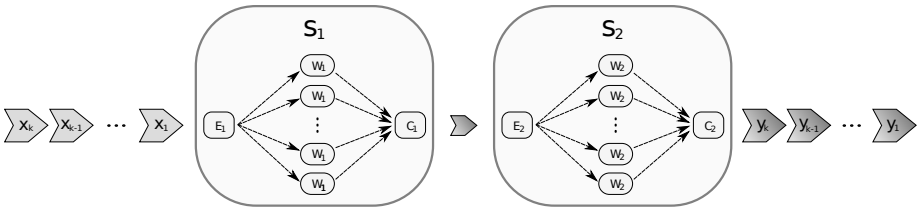


FIGURE 2.7: A combination of pipeline and farm patterns, where each stage of a two-stage pipeline is a farm. In this figure, subscripts in farms components are used to distinguish the two farms.

The *Loop* skeleton (also known as *feedback*), provides a way to generate cycles in a stream graph. By using a feedback channel, it is possible to route back results in order to implement iterative computations. This skeleton is typically used together with the farm skeleton to model recursive and Divide&Conquer computations, in which an input task is recursively sub-divided in sub-tasks until a condition is met, then each sub-task is executed and results are merged. For instance, in a Farm pattern as in Fig. 2.6 we can connect the Collector to the Emitter with a *feedback channel*. In general it is possible to connect Farm, Pipeline and feedback channels to build more complex networks.

2.4 Programming multicore clusters

Programming tools and frameworks are needed to efficiently target architectures hosting inter networked — possibly heterogeneous — multicore devices, which appear to be *the* reference architecture ferrying programmers from the mainly sequential to mainly parallel programming era [29].

Shared-memory multicores and clusters/networks of processing elements require quite different techniques and tools to support efficient parallelism exploitation. The *de facto* standard tools in both cases are OpenMP [106] and MPI [105], used either alone or in conjunction. Despite being efficient on some classes of applications, OpenMP and MPI share a common set of problems: poor separation of concerns among application and system aspects, a rather low level of abstraction presented to the application programmers, and poor support for really fine grained applications.

At the moment, it is not clear if the mixed MPI/OpenMP programming model always offers the most effective mechanisms for programming clusters of SMP systems [41]. Furthermore, when directly using communication libraries such as MPI, the abstraction level is rather low and the programmer has to think about decomposing the problem, integrating the partial solutions, and bother with communication problems such as deadlocks and starvation.

Therefore, we advocate the use of high-level parallel programming frameworks targeting hybrid multicore and distributed platforms as a vehicle for building efficient parallel software — possibly derived semi-automatically from sequential code — featuring performance portability over heterogeneous parallel platforms.

2.4.1 FastFlow

FastFlow [56] is an open source, structured parallel programming framework supporting highly efficient stream parallel computation while targeting shared memory multicores. Its efficiency mainly comes from the optimized implementation of the base communication mechanisms and from its layered design. It provides a set of

ready-to-use, parametric algorithmic skeletons modeling the most common parallelism exploitation patterns, which may be freely nested to model more complex parallelism exploitation patterns. It is realized as a C++ pattern-based parallel programming framework aimed at simplifying the development of applications for (shared-memory) multi-core and GPGPU platforms.

The key vision of FastFlow is that ease-of-development and runtime efficiency can both be achieved by raising the abstraction level of the design phase. FastFlow provides a set of algorithmic skeletons addressing both stream parallelism — with the farm and pipeline patterns — and data parallelism — providing stencil, map, reduce pattern, and their arbitrary nesting and composition [19]. Map, reduce and stencil patterns can be run both on multi-cores and offloaded onto GPUs. In the latter case, the user code can include GPU-specific statements (i.e., CUDA or OpenCL statements).

Leveraging the farm skeleton, FastFlow exposes a *ParallelFor* pattern [55], where chunks of a loop iterations having the form `for(idx=start;idx<stop;idx+=step)` are executed by the farm workers. Just like TBB, FastFlow's ParallelFor pattern uses C++11 *lambda* expression as a concise and elegant way to create function objects: lambdas can "capture" the state of non-local variables by value or by reference and allow functions to be syntactically defined when needed.

From the performance viewpoint, one distinguishing feature at the core of FastFlow is that it supports lock-free (fence-free) Multiple-Producer-Multiple-Consumer (MPMC) queues [18] that can support low-overhead high-bandwidth multi-party communications on multicore architectures, i.e., any *streaming network*, including cyclic graphs of threads. The key intuition underneath FastFlow is to provide the programmer with lock-free MP queues and MC queues (that can be used in pipeline to build MPMC queues) to support fast streaming networks.

Traditionally, MPMC queues are built as passive entities: threads concurrently synchronize (according to some protocol) to access data; these synchronizations are usually supported by one or more atomic operations (e.g., Compare-And-Swap) that behave as memory fences. FastFlow design follows a different approach: to avoid any memory fence, the synchronizations among queue readers or writers are arbitrated by an active entity (e.g., a thread). We call these entities *Emitter* (E) or *Collector* (C) according to their role; they actually read an item from one or more lock-free SPSC queues and write onto one or more lock-free SPSC queues. This requires a memory copy but no atomic operations.

The performance advantage of this solution comes from the higher speed of the copy operation compared with the memory fence; this advantage is further increased by avoiding cache invalidation triggered by fences. This also depends on the size and the memory layout of copied data. The former point is addressed using data pointers instead of data, and ensuring that the data is not concurrently written: in many cases this can be derived by the semantics of the skeleton that has been implemented using MPMC queues — as an example this is guaranteed in a stateless farm and many other cases.

FastFlow design is layered (see Fig. 2.8). The lower layer, called *simple streaming networks*, basically provides two basic abstractions:

- *process-component*, i.e., a control flow realized with POSIX threads and processes, for multicore and distributed platforms respectively.
- *1-1 channel*, i.e., a communication channel between two components, realized with wait-free single-producer/single-consumer queues (FF-SPSC) [14] or zero-copy *ZeroMQ* channels [134], for multicore and distributed platforms, respectively.

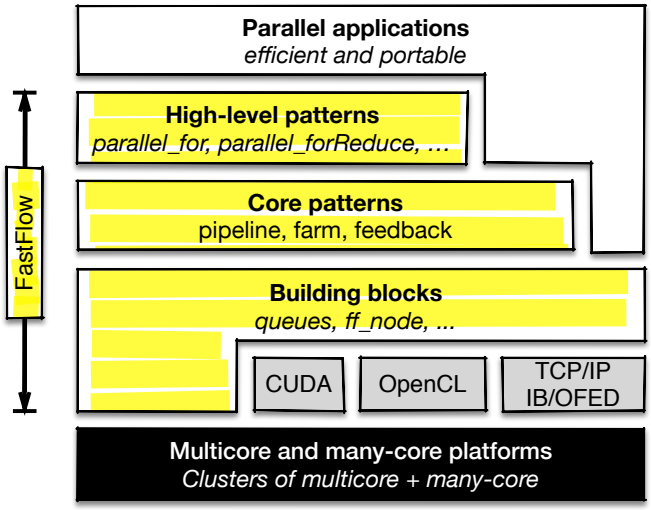


FIGURE 2.8: Layered FastFlow design .

Both realizations of the 1-1 channel are top state-of-the-art in their classes, in terms of latency and bandwidth. As an example, FF-SPSC exhibits a latency down to 10 nanoseconds per message on a standard Intel Xeon @2.0GHz [14].

```
1 class ff_node {
2  protected:
3   virtual bool push(void* data) { return qout->push(data); }
4   virtual bool pop(void** data) { return qin->pop(data); }
5  public:
6   virtual void* svc(void * task) = 0;
7   virtual int   svc_init() { return 0; };
8   virtual void  svc_end()  {}
9   ...
10 private:
11  SPSC* qin;
12  SPSC* qout;
13 };
```

LISTING 2.1: FastFlow `ff_node` class schema.

Above this mechanism, the second layer — called *arbitrary streaming networks* — further generalizes the two concepts, providing:

- *FastFlow node*, i.e., the basic unit of parallelism that is typically identified with a node in a streaming network. It is used to encapsulate sequential portions of code implementing functions, as well as high-level parallel patterns such as pipelines and farms. From the implementation viewpoint, the `ff_node` C++ class realizes a node in the shared-memory scenario and the `ff_dnode` extends it in the distributed memory setting (see Listing 2.1).
- *Collective channel*, i.e., a collective communication channel, either among `ff_node`s or many `ff_dnode`s.

Eventually, the third layer provides the farm, pipeline and other parallel patterns as C++ classes.

Each `ff_node` is used to run a concurrent activity in a component, and it has two associated channels: one used to receive input data (pointers) to be processed and one to deliver the (pointers to the) computed results. Representing communication and synchronization as a channel ensures that synchronisation is tied to communication and allows layers of abstraction at higher levels to compose parallel programs where synchronization is implicit. The `svc` method encapsulates the computation

to be performed on each input datum to obtain the output result. `svc_init` and `svc_end` methods are executed when the application is started and before it is terminated. The three methods should be provided by the programmer in order to instantiate an `ff_node`.

Distributed FastFlow

FastFlow also provides an implementation running on distributed systems, based on the ZeroMQ library. Briefly, ZeroMQ is an LGPL open-source communication library [134] providing the user with a socket layer that carries whole messages across various transports: inter-thread communications, inter-process communications, TCP/IP, and multicast sockets. ZeroMQ offers an asynchronous communication model, which provides a quick construction of complex asynchronous message-passing networks, with reasonable performance.

A `ff_dnode` (distributed `ff_node`) provides an external channel specialized to provide different patterns of communication. The set of communication collectives allows one to provide exchange of messages among a set of distributed nodes, using well-known predefined patterns. The semantics of each communication pattern currently implemented are summarized in Table 2.1. In the current version (see Fig. 2.8), which supports distributed platforms, many graphs of `ff_node`s can be connected by way of `ff_dnode`s (which support network collective channels), thus providing a useful abstraction for effective programming of hybrid multicore and distributed platforms.

Pattern	Description
unicast	unidirectional point-to-point communication between two peers
broadcast	sends the same input data to all connected peers
scatter	sends different parts of the input data, typically partitions, to all connected peers
onDemand	the input data is sent to one of the connected peers, the choice of which is taken at runtime on the basis of the actual workload
fromAll	aka. *all-gather*, collects different parts of the data from all connected peers combining them in a single data item
fromAny	collects one data item from one of the connected peers

TABLE 2.1: Collective communication patterns among `ff_dnode`s.

The FastFlow programming model is based on streaming of pointers, which are used as synchronization tokens. This abstraction is kept also in the distributed version (i.e., across network channels) by way of two auxiliary methods provided by `ff_dnode` for data *marshalling* and *unmarshalling* according to ZeroMQ specifics. These (virtual) methods provide the programmers with the tools to serialize and de-serialize data flowing across `ff_dnode`s. The hand-made serialization slightly increases the coding complexity (e.g., with respect to Java automatic serialization) but makes it possible to build efficient network channels. While ZeroMQ provides an abstraction to implement a queue-based distributed network, different libraries

can be used for the data serialization step, e.g., Boost.Serialize [93] or Google Protobuf [94].

FastFlow has also been provided with a minimal message passing layer implemented on top of InfiniBand RDMA features [112]. The proposed RDMA-based communication channel implementation achieves comparable performance with highly optimised MPI/InfiniBand implementations. The results obtained demonstrate that the RDMA-based library can improve application performance by more than 30% with a consistent reduction of CPU time utilization with respect to the original TCP/IP implementation.

2.5 Summary

In this chapter we provided a review of the most common parallel computing platforms, programming models for multicore and cluster of multicores. From a hardware perspective, we presented multicore, manycore processors such as accelerators and distributed systems. From a high-level programming model perspective, we listed the four main classes of parallelism (task, data, stream and dataflow parallelism) followed by some insights about the Dataflow model, a model of computation describing a program as a set of concurrent processes extensively used in this thesis. We also provided a review of low-level approaches in parallel programming, such as POSIX programming model, MPI, TBB and OpenMP. In particular, we reviewed programming model for the described platforms exploiting high-level skeleton-based approaches, focusing on the FastFlow library, which we use in this work to implement the PiCo runtime.

Chapter 3

Overview of Big Data Analytics Tools

Big Data is a term used to identify data sets that are very large and/or complex (i.e., unstructured) so that traditional data processing applications are inadequate to process them. In this chapter we will describe what Big Data is starting by its "formal" definition. We then continue with a survey of the state-of-the-art of Big Data Analytics tools.

3.1 A Definition for Big Data

Big Data has been defined as the "3Vs" model, an informal definition proposed by Beyer and Laney [32, 90] that has been widely accepted by the community:

> *"Big data is high-Volume, high-Velocity and/or high-Variety information assets that demand cost-effective, innovative forms of information processing that enable enhanced insight, decision making, and process automation."*

More in detail:

- **Volume**: The amount of data that is generated and stored. The size of the data determines whether it can be considered big data or not
- **Velocity**: The speed at which the data is generated and processed
- **Variety**: The type and nature of the data that, when collected, are typically unstructured

The "3Vs" model can be extended by adding two more V-features: 1) **Variability**, since data not only are unstructured, but can also be of different types (i.e. text, images) or even inconsistent and, 2) **Veracity**, in the sense that the quality and accuracy of the data may vary.

In 2013, IBM tried to quantify the amount of data being produced: "Every day, we create 2.5 quintillion bytes of data — so much that 90% of the data in the world today has been created in the last two years alone." [75]

There is no clear-cut definition on how "big" the data should be to be considered Big Data. Data is said to be *Big* if it becomes difficult to be stored, analyzed and searched using traditional database systems, since it is large, unstructured and hard to be organized. Consider, for instance, large organizations like Facebook: having more than 950 millions users, it pulls in 500 Terabytes per day, into a 100 Petabytes

warehouse, and runs 70 000 queries per day on this data as of 2012.[1] Here are some of the statistics provided by the company:

- 2.5 billion content items shared per day
- 2.7 billion Likes per day
- 300 million photos uploaded per day
- 100+ Petabytes of disk space in one of their largest Hadoop (HDFS) clusters
- 105 Terabytes of data scanned every 30 minutes
- 70 000 queries executed on these databases per day
- 500+ Terabytes of new data stored into the databases every day

Of course, Facebook is not the unique entity producing Big Data. There is a large set of use cases in which Big Data analysis can produce knowledge. For instance, the *IoT* (Internet of Things) is the interconnection of uniquely identifiable devices connected (also among each other) to the Internet. Those devices are referred to as "connected" or "smart" devices. Big Data and IoT are strictly connected, since billions of devices produce massive amount of data, and companies can benefit from analyzing all that information in order to produce new knowledge. Another use case example comes from healthcare and genomics, where the amount of data being produced by sequencing, mapping, and analyzing genomes takes those areas into the realm of Big Data. Big Data can improve operational efficiencies, help predict and plan responses to disease epidemics, improve the quality of monitoring of clinical trials, and optimize healthcare spending at all levels from patients to hospital systems to governments. Another key area is genomics sequencing which is expected to be the future of healthcare [33]. Big Data can also be not that big. Companies not as big as Facebook or Twitter may be interested in analyzing their data, as well as institutions, public offices, banks, etc.

Extracting knowledge from Big Data is also related to programmability, that is, how to easily write programs and algorithms to analyze data, and performance issues, such as scalability when running analysis and queries on multicore or cluster of multicores. For this reason, starting from the Google MapReduce paper [61], a constantly increasing number of frameworks for Big Data processing has been implemented. In the next section, we provide an overview of such frameworks, starting from an sum-up about data management.

3.2 Big Data Management

Big Data management is more complex than simple database management. It can be seen as the process of capturing, delivering, operating, protecting, enhancing, and disposing of the data cost-effectively, which needs the ever-going reinforcement of plans, policies, programs, and practices [92].

HDFS The Hadoop Distributed File System (HDFS) [23] is a distributed file system designed to run on commodity hardware. It is highly fault-tolerant and designed to be deployed on low-cost hardware. It is part of the Hadoop Ecosystem [22], consisting of: 1) the high-throughput Hadoop Distributed File System *HDFS* [23], 2) MapReduce engine for data processing and, 3) Hadoop *YARN* [26] software for resource management and job scheduling and monitoring. In recent years, Hadoop has been detached from the MapReduce processing engine, so that

[1] https://gigaom.com/2012/08/22/facebook-is-collecting-your-data-500-terabytes-a-day/

it became possible to use the HDFS+YARN layer as the base ecosystem for other frameworks, such as Apache Spark, Apache Storm, etc.

HDFS provides high throughput access to application data and is suitable for applications that have large data sets. Being a software distributed file system, HDFS runs on top of the local file systems but it appears to the user as a single namespace accessible via HDFS URI.

An HDFS instance typically run on hundred or thousands of commodity components that may fail, implying that, at a given time, there can be some components not (virtually) working or unable to recover from their failure. Therefore, the system should support constant monitoring, error detection, fault tolerance, in order to automatically recover from failure. HDFS is tuned to support a modest number (tens of millions) of large files, which are typically Gigabytes to Terabytes in size. In its first implementation, HDFS assumed a write-once-read-many access model for files, assumption that simplified the data coherency problem and enabled high throughput data access. The append operation was added later (single appender only).[2] HDFS is mainly designed for batch processing, since it is specialized for high throughput of data access rather than low latency.

NoSQL The *Not Only SQL* model [104] is a distributed database whose data model is not strictly relational. The data structures used by NoSQL databases (e.g. key-value, plain document, graph) are different from those used by default in relational databases, making some operations faster in NoSQL. These databases have to abide by the CAP theorem (Consistency-Availability-Partition tolerance), proposed by Brewer in [39]. This theorem asserts that a distributed system cannot provide simultaneously all the following properties:

1. Consistency: all nodes see the same data at the same moment (single up-to-date copy of the data);
2. Availability: all data should always be available by all entities requesting. If a node is not responsive, any data request must terminate;
3. Partition tolerance: the system continues to operate despite arbitrary message loss or failure of part of the system.

By playing with the CAP theorem, many NoSQL stores compromise consistency in favor of availability and partition tolerance, which also brings design simplicity. A distributed database system does not have to drop consistency and are ACID compliant but, on the other hand, these systems are much more complex and slow with respect to NoSQL. Frameworks using the NoSQL paradigm are, for instance, ZooKeeper [27], HBase [25], Riak [31] or Cassandra [24].

3.3 Tools for Big Data Analytics

In this section, we provide the background related to Big Data analytics tools, starting from the Google MapReduce model to the most modern frameworks.

3.3.1 Google MapReduce

Programming for Big Data poses many challenges, first of all ease of programming and high performance. Google can be considered the pioneer of Big Data processing,

[2] https://issues.apache.org/jira/browse/HDFS-265

as the publication of the MapReduce framework paper [61] made this model "popular" — almost "mainstream". Inspired by the map and reduce functions commonly used in functional programming, a MapReduce program is composed of a `map` and a `reduce` procedures: more specifically, the user-defined `map` function processes a key-value pair to generate a set of intermediate key-value pairs, and the `reduce` function aggregates all intermediate values associated with the same intermediate key.

The contribution of the MapReduce programming model is not the composition of `map` and `reduce` functions, but in the utilization of a key-value model in this process and the repartitioning step, known as shuffle. One aspect often stressed underlying this model is the data locality exploitation during the map phase, following the idea of moving the computation to the data. That is, the runtime exploits the natural data partitioning on a distributed file systems by forcing operations to be computed using only local data — of course, during the shuffle phase, some data will be copied to other nodes of the cluster.

We now explain in more detail how a MapReduce computation is characterized.

The five steps of a MapReduce job

From a high-level perspective, a MapReduce job can be divided into three macro steps: 1) a *Map* step in which each worker node applies the `map` function to the local data, transforming each datum v into a pair $\langle\ k, v\ \rangle$, and writes the output to temporary storage; 2) a *Shuffle* step where worker nodes redistribute data based on the value of the keys such that all data belonging to one key is located on the same worker node, and 3) a *Reduce* step where each worker computes the `reduce` function on local (key-partitioned) data.

Users write MapReduce applications defining the sequential kernel functions (`map`, `reduce`) that are automatically parallelized by the runtime and executed on a large cluster. During the execution, `map` invocations are distributed across multiple machines, provided each `map` operation can be executed independently on input split on different machines. The same applies for the `reduce` invocations, that are distributed by partitioning the intermediate key space into R sets, provided that all outputs of the `map` operation with the same key are processed by the same reducer at the same time, and that the reduction function is associative and commutative. Besides partitioning the input data and running the various tasks in parallel, the runtime also manages all communications and data transfers, load balance, and fault tolerance.

Looking more deeply into the MapReduce model, it can be described as a 5-step parallel and distributed computation:

1. **Input preparation**: the runtime designates Map processors by assigning the input key value $k1$ that each processor would work on, and provides that processor with all the input data associated with that key value.
2. **User-defined `map` execution**: `map` is run exactly once for each $k1$ key value, generating output organized by key values $k2$. The intermediate key-value pairs produced by the `map` function are buffered in memory. Periodically, the buffered pairs are written to local disk, partitioned into R regions by the partitioning function (called partitioner) that is given the key and the number of reducers R and returns the index of the desired reducer processor.
3. **Shuffle `map` output**: reduce processors are designed by the runtime, by assigning $k2$ key value to a predefined processor. A reducer reads remotely the buffered data from the local disks of the Map workers. When a Reduce worker completes with data reading, it sorts the data by the intermediate keys. Typically many different keys map to the same reduce task.

4. **User-defined `reduce` execution**: `reduce` is run exactly once for each $k2$ produced by the `map` step assigned to each Reduce worker.
5. **Produce output**: the MapReduce system collects all the `reduce` output for each key to produce the final outcome.

These five steps can be logically thought of as running in sequence — each step starts only after the previous step is completed — although in practice they can be interleaved as long as the final result is not affected. Out of these five steps, sorting and partitioning are performed natively by the runtime.

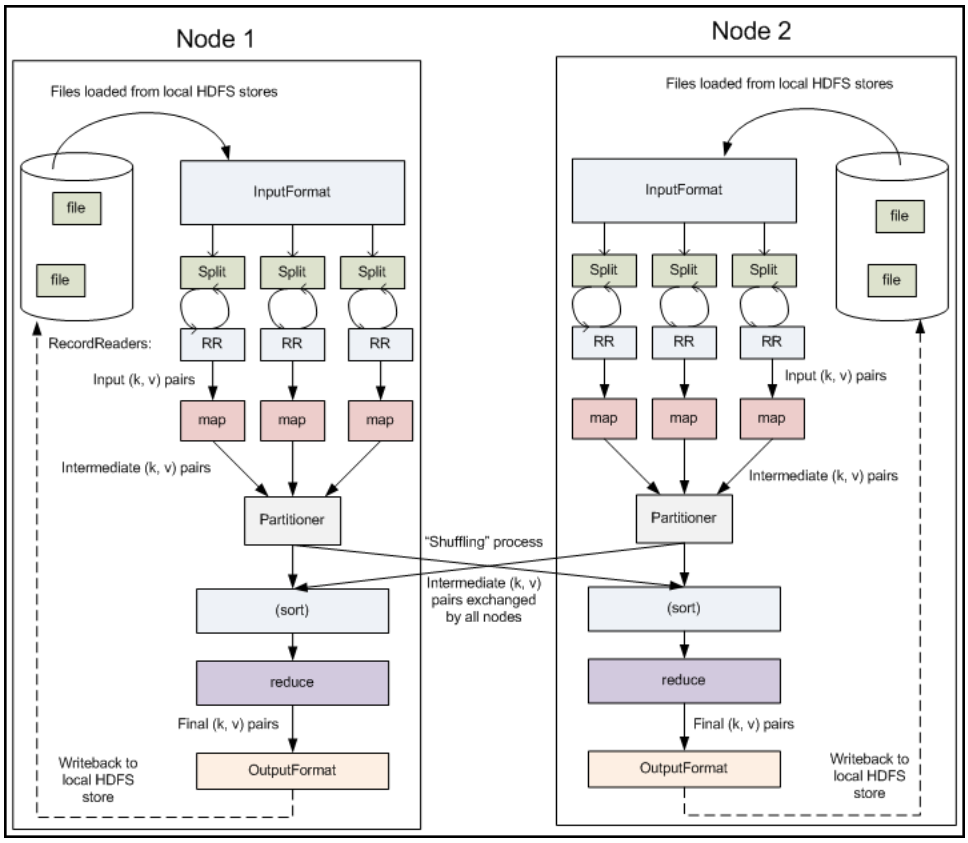


FIGURE 3.1: An example of a MapReduce Dataflow

Figure 3.1[3] shows the dataflow of operations coming from a MapReduce execution. In more detail, it shows the pipeline (input pre-processing – map – sort – shuffle – reduce – output processing) in all of its inner steps. It can be noted that while only two nodes are depicted, the same pipeline is replicated across all nodes, thus making all the worker nodes aware of the whole computation.

MapReduce tasks must be written as acyclic dataflow programs, i.e. a stateless mapper followed by a stateless reducer, putting some limitations in performance when considering iterative computations such as machine learning algorithms [99]. This is especially due to the repeated writes to disk, that represent a bottleneck when revisiting a single working set multiple times is necessary [132].

In Listing 3.1, we show a source code extract of a MapReduce application, in which only the `map` and `reduce` function are presented.

[3]Figure taken from https://developer.yahoo.com/hadoop/tutorial/module4.html

```
1 public class WordCount {
2   public static class TokenizerMapper extends Mapper<Object,Text,Text,
      IntWritable>{
3     private final static IntWritable one = new IntWritable(1);
4     private Text word = new Text();
5
6     public void map(Object key, Text value, Context context) throws IOException,
      InterruptedException{
7       StringTokenizer itr = new StringTokenizer(value.toString());
8       while (itr.hasMoreTokens()) {
9           word.set(itr.nextToken());
10        context.write(word, one);
11      }
12    }
13  }
14 }
15
16 public static class IntSumReducer extends Reducer<Text,IntWritable,Text,
      IntWritable> {
17   private IntWritable result = new IntWritable();
18   public void reduce(Text key, Iterable<IntWritable> values, Context context)
      throws IOException, InterruptedException {
19     int sum = 0;
20     for (IntWritable val : values) {
21       sum += val.get();
22     }
23     result.set(sum);
24     context.write(key, result);
25   }
26  }
```

LISTING 3.1: The `map` and `reduce` functions for a Java Word Count class example in MapReduce.

3.3.2 Microsoft Dryad

Dryad [79] is a Microsoft research framework for distributed processing of coarse-grain data parallel applications. A Dryad application is a Dataflow graph where computational vertexes are connected among each other with communication channels. Dryad runs the application by executing the graph's vertexes on a set of available computers, communicating through files, TCP pipes, and shared-memory FIFO queues. Kernel functions executed by the vertexes are sequential functions (with no threads creation or data contention) provided by the user. Concurrency arises from scheduling vertexes to run simultaneously on shared memory or on distributed systems. A Dryad application developer can specify an arbitrary directed acyclic graph to describe the application's communication patterns, and express the data transport mechanisms (files, TCP pipes, and shared memory FIFO queues) between the computation vertexes. Dryad is notable for allowing graph vertexes to use an arbitrary number of inputs and outputs, while MapReduce restricts all computations to take a single input set and generate a single output set. Although Dryad provides a nice alternative to MapReduce, Microsoft terminated development on Dryad in 2011.

A Dryad application DAG

In Dryad, graphs are created by combining subgraphs or by using simple vertexes. A graph is formally defined as a tuple $G = \langle V_G, E_G, I_G, O_G \rangle$ where V_G and E_G are set of vertexes and edges, while $I_G \subseteq V_G$ and $O_G \subseteq V_G$ are input and output vertexes, respectively. No graph can contain edges entering an input vertex nor can contain edges exiting an output vertex. Edges are created in two ways: 1) by applying a composition of two existing graphs (a single vertex is considered a graph), and 2) by

merging graphs by the union of their vertexes and edges. Edges represent channels. By default, channels are implemented as temporary files where the producer vertex writes to and the producer vertex reads from. Users can even choose to use channel implemented as shared memory FIFO queues or TCP channels. Once created, a Dryad DAG is optimized using various heuristics. The Dryad runtime parallelizes the application graph by distributing the computational vertexes across various execution engines (in shared memory or shared nothing). Note that replicating the whole application graph is also a valid distribution, to exploit data parallelism. Scheduling of the computational vertexes on the available hardware is handled by the runtime.

3.3.3 Microsoft Naiad

Naiad is an investigation of data-parallel dataflow computation, successor of Dryad, focusing on low-latency streaming and cyclic computations [96]. Naiad's authors introduced a new computational model called Timely Dataflow, which enriches dataflow computation with timestamps that represent logical points in the computation. This model provides the user with the possibility of implementing iterative computations, thanks to feedback loops in the dataflow. Stateful nodes in the dataflow can manage a global status for coordination: they asynchronously receive messages and notifications of global progress, used also in loop management.

Timely Dataflow and Naiad programming model

Timely Dataflow is a model of data-parallel computation that extends traditional dataflow by associating each communication event with a *virtual time* that does not need to be ordered. As in the Time Warp mechanism [81], virtual times serve to differentiate between data in different phases or aspects of a computation, for example, data associated with different batches of inputs and different loop iterations. A formal definition of Timely Dataflow can be found in [3]. In this model, each node may request different notifications about completions for loops, for instance, thus allowing asynchronous processing. Timely Dataflow graphs are directed graphs where vertexes are organized into possibly nested loop contexts. Edges entering a loop context must pass through an ingress vertex and edges leaving a loop context must pass through an egress vertex. Additionally, every cycle in the graph must be contained entirely within some loop context, and must include at least one feedback vertex that is not nested within any inner loop context. Every token flowing on edges carries a timestamp of the form $Timestamp := (e \in \mathbb{N}, \langle c_1, \dots c_n \rangle \in \mathbb{N}^k)$. The value e represents an *epoch*, corresponding to integer timestamp representing the id of the item within the stream, while k is the depth of nesting in a timestamp having $k \geq 0$ loop counters.

The programming model is quite straightforward: a Naiad program is implemented by means of Dataflow nodes created by extending the `Vertex<T>` interface. Such vertex is a possibly stateful object that sends and receives messages, and requests and receives notifications. Message exchange and notification delivery are asynchronous. Vertexes can modify arbitrary state in their callbacks, and send messages on any of their outgoing edges. Parallelism is obtained by creating multiple instances of some vertexes, thus exploiting data parallelism in operators, inter-operators parallelism and loop parallelism. A Naiad application may be run on shared memory and on distributed systems. The smallest unit of computation is the Worker, namely a single thread, called a *shard*. Processes are larger unit of computation, which may contain one or more Workers (thus executing one or more vertexes). A single machine may host one or more processes.

Listing 3.2 presents a code fragment implementing a MapReduce style computation.

```
// 1a. Define input stages for the dataflow.
var input = controller.NewInput<string>();

// 1b. Define the timely dataflow graph.
var result = input.SelectMany(y => map(y))
                      .GroupBy(y => key(y), (k, vs) => reduce(k, vs));

// 1c. Define output callbacks for each epoch
result.Subscribe(result => { ... });

// Step 2. Supply input data to the query.
input.OnNext(/* 1st epoch data */);
input.OnNext(/* 2nd epoch data */);
input.OnNext(/* 3rd epoch data */);
input.OnCompleted();
```

LISTING 3.2: A LINQ MapReduce computation example in Naiad (taken from [96]).

Step 1a defines the source of data, and Step 1c defines what to do with output data when produced. Step 1b constructs a Timely Dataflow graph using `SelectMany` and `GroupBy` library calls, which are pre-defined vertexes. `SelectMany` applies its argument function to each message (thus it is a `map` function), and `GroupBy` groups results by a key function before applying its reduction function `reduce`. Once the graph is built, in step 2 `OnNext` supplies the computation with epochs of input data. The `Subscribe` stage applies its callback to each completed epoch of data it observes. Finally, `OnCompleted` indicates that no further epochs of input data exist, so the runtime can drain messages and shut down the computation.

3.3.4 Apache Spark

As mentioned in previous sections, MapReduce is not suitable for iterative algorithms or interactive analytics, not because it is not possible to implement iterative jobs, but because data have to be repeatedly stored and loaded at each iteration. Furthermore, data can be replicated on the distributed file system between successive jobs. Apache Spark [133, 131, 132] design is intended to solve this problem by reusing the working dataset by keeping it in memory, 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), which are immutable multisets. In this Section, we only give an overview of Spark, which will be further investigated in Chapter 4.

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. Transformations are the functional style operations applicable to collections, such as `map`, `reduce`, `flatmap`, that are uniformly applied to whole RDDs [131]. Actions return a value to the user after running a computation on the dataset, thus they effectively start the program execution. Transformations in Spark are lazy, in that they do not compute their results right away. Instead, they just store the transformations applied to some base dataset, and they are computed when an action requires a result to be returned to the driver program, that is, the main program written by the programmer.

Listing 3.3 shows the source code for a simple Word Count application in the Java Spark API.

```
JavaRDD<String> textFile=sc.textFile("hdfs://...");

JavaRDD<String> words =
  textFile.flatMap(new FlatMapFunction<String, String>() {
    public Iterable<String> call(String s) {
    return Arrays.asList(s.split(" "));
    }
  });

JavaPairRDD<String, Integer> pairs =
  words.mapToPair(new PairFunction<String, String, Integer>() {
    public Tuple2<String, Integer> call(String s) {
    return new Tuple2<String, Integer>(s, 1);
  }
});

JavaPairRDD<String, Integer> counts =
  pairs.reduceByKey(new Function2<Integer, Integer, Integer>() {
    public Integer call(Integer a, Integer b) {
    return a + b;
  }
});
counts.saveAsTextFile("hdfs://...");
```

Listing 3.3: A Java Word Count example in Spark.

For stream processing, Spark implements an extension through the Spark Streaming module, providing a high-level abstraction called *discretized stream* or *DStream* [133]. Such streams represent results in continuous sequences of RDDs of the same type, called *micro-batch*. Spark's execution model relies on the Master-Worker model: a cluster manager (e.g., YARN) manages resources and supervises the execution of the program. It manages application scheduling to worker nodes, which execute the application logic (the DAG) that has been serialized and sent by the master.

Resilient Distributed Datasets

An RDD is a read-only collection of objects partitioned across a cluster of computers that can be operated on in parallel. A Spark application consists of a driver program that creates RDDs from, for instance, HDFS files or an existing Scala collection as well as creating RDDs from queries to databases. The driver program may transform an RDD in parallel by invoking supported operations with user-defined functions, which returns another RDD. Since RDDs are read-only collections, each transformation on such collections creates a new RDD containing the result of the transformation. This new collection is not materialized, nor kept in memory: it is the result of an expression rather than a value and it is computed each time the transformation is called. The driver can also persist an RDD in memory, allowing it to be reused efficiently across parallel operations without recomputing it. In fact, the semantics of RDDs have the following properties:

- Abstract: elements of an RDD do not have to exist in physical memory. In this sense, an element of an RDD is an expression rather than a value. The value can be computed by evaluating the expression when necessary (i.e., when executing an *action*).
- Lazy and Ephemeral: RDDs can be created from a file or by transforming an existing RDD such as `map`, `filter`, `groupByKey`, `reduceByKey`, `join`, etc. However, RDDs are materialized on demand when they are used in some operation, and are discarded from memory after use. This thus performs a sort of lazy evaluation.

- Caching and Persistence: a dataset can be cached in memory across operations, which allows future actions to be much faster since they not have to be reconstructed. Caching is a key tool for iterative algorithms and fast interactive use cases and it is actually one special case of persistence that allows different storage levels, e.g. persisting the dataset on disk or in memory but as serialized Java objects (to save space), replicating it across nodes, or storing it off-JVM heap.
- Fault Tolerance: if any partition of an RDD is lost, the lost block will automatically be recomputed using only the transformations that originally created it.

The operations on RDDs take user-defined functions, which are considered closures in functional programming style features provided by the Scala programming language, used to implement the Spark runtime. A closure can refer to variables in the scope when created, which will be copied to the workers when Spark runs a closure. We recall that, exploiting the JVM and serialization features, it is possible to serialize closures and send them to Workers.

Operations on RDDs take user-defined functions as arguments, which act as closures that can refer to non-local variables from the scope where they were created. These captured variables will be copied to the workers when Spark runs a closure. We recall that, exploiting the JVM and serialization features, it is possible to serialize closures and send them to workers.

Spark also offers two kinds of shared variables:

- Broadcast variables, which are copied to the workers once, and appropriate when large read-only data is used in multiple operations
- Accumulators, which are states local to Workers that are only "updated" through an associative operation and can therefore be efficiently supported in parallel. Accumulators can thus be used to implement counters or sums. Only the driver program can read the accumulator's value. Spark natively supports accumulators of numeric types.

By reusing cached data in RDDs, Spark offers great performance improvement over Hadoop MapReduce [131], thus making it suitable for iterative machine learning algorithms. Similar to MapReduce, Spark is independent of the underlying storage system. It is the application developer's duty to organize data on distributed nodes — i.e., by partitioning and collocating related datasets, etc. — if a distributed file system is not available. These are critical for interactive analytics, since just caching is insufficient and not effective for extremely large data.

Spark Streaming

For stream processing, Spark implements an extension through the Spark Streaming module, providing a high-level abstraction called *discretized stream* or *DStream* [133]. Such streams represent results in continuous sequences of RDDs of the same type, called *micro-batch*. 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 simple translation $\texttt{op}(a) = [\texttt{op}(a_1), \texttt{op}(a_2), \ldots]$, where $[\cdot]$ refers to a possibly unbounded ordered sequence, $a = [a_1, a_2, \ldots]$ is a DStream, and each item a_i is a micro-batch of type RDD. All RDDs in a DStream are processed in order, whereas data items inside an RDD are processed in parallel without any ordering guarantees.

3.3.5 Apache Flink

Formerly known as Stratosphere [21], *Apache Flink* [42] focuses on stream programming. The abstraction used is the *DataStream*, which is 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 multiset—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 more advanced stream processors as many of its core features were already considered in the initial design [42]. A strong attention is put on providing exactly-once processing guarantee, reached by implementing an algorithm based on Chandy-Lamport algorithm for distributed snapshots [44]. In this algorithm, watermark items are periodically injected into the data stream and trigger any receiving component to create a checkpoint of its local state. On success, all local checkpoints for a given watermark comprise a distributed global system checkpoint. In a failure scenario, all components are reset to the last valid global checkpoint and data is replayed from the corresponding watermark. Since data items may never overtake watermark items, acknowledgment does not happen on a per-item basis.

Other aspects of Flink will be further investigated in Chapter 4.

Flink Programming and Execution Model

Flink adopts a programming model similar to Spark, in which operations are applied to datasets and streams in a object oriented fashion. The basic building blocks are streams and transformations (note that a DataSet is internally also a stream). A stream is an intermediate result, and a transformation is an operation that takes one or more streams as input, and computes one or more result streams.

Listing 3.4 shows Flink's source code for the simple Word Count application.

```
1 public class WordCountExample {
2   public static void main(String[] args) throws Exception {
3     final ExecutionEnvironment env =
4       ExecutionEnvironment.getExecutionEnvironment();
5     DataSet<String> text = env.fromElements("Text...");
6     DataSet<Tuple2<String, Integer>> wordCounts =
7       text.flatMap(new LineSplitter())
8           .groupBy(0)
9           .sum(1);
10
11    wordCounts.print();
12  }
13
14 public static class LineSplitter
15   implements FlatMapFunction<String, Tuple2<String, Integer>> {
16     @Override
17     public void flatMap(String line,
18     Collector<Tuple2<String, Integer>> out) {
19       for (String word : line.split(" ")) {
20         out.collect(new Tuple2<String, Integer>(word, 1));
21       }
22     }
23   }
24 }
```

Listing 3.4: A Java Word Count example in Flink.

Flink programs are mapped to streaming dataflows, consisting of streams and transformation operators that build arbitrary directed acyclic graphs (DAGs). Each

dataflow starts with one or more sources and ends in one or more sinks. Special forms of cycles are permitted via iteration constructs that are discussed in Section 4.2.3.

Flink's execution model relies on the Master-Worker model: a deployment has at least one job manager process that coordinates checkpointing, recovery, and that receives Flink jobs. The job manager also schedules work across the task manager processes (i.e. workers) which usually reside on separate machines and in turn execute the code. For distributed execution, Flink optimizes the DAG by chaining operator subtasks together into tasks: it reduces the overhead of thread-to-thread handover and buffering, and increases overall throughput while decreasing latency. Stateful stream operators are discussed in Section 4.2.3.

3.3.6 Apache Storm

Apache Storm [97, 117, 126] is a framework targeting only stream processing. 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. Most of the logic of a computation goes into Bolts, such as functions, filters, streaming joins or streaming aggregations. A Topology is the composition of Spouts and Bolts resulting in a network. Storm uses *tuples* as its data model, that is, named lists of values of arbitrary type. Hence, Bolts are parametrized with per-tuple kernel code. Each time a tuple is available from some input stream, the kernel code gets activated to work on that input tuple. Bolts and Spouts are locally stateful, as we discuss in Section 4.2.3, while no global consistent state is supported. Globally stateful computations can be implemented since the kernel code of Spouts and Bolts is arbitrary. However, eventual global state management would be the sole responsibility of the user, who has to be aware of the underlying execution model in order ensure program coordination among Spouts and Bolts. It is also possible to define cyclic graphs by way of feedback channels connecting Bolts.

While Storm targets single-tuple granularity in its base interface, the Trident API is an abstraction that provides declarative stream processing on top of Storm. Namely, Trident processes streams as a series of micro-batches belonging to a stream considered as a single object. Listing 3.5 show Storm's source code for the simple Word Count application.

```
1 public class WordCountTopology {
2   public static class SplitSentence extends ShellBolt  implements IRichBolt {
3   public SplitSentence() {
4     super("python", "splitsentence.py");
5   }
6
7   @Override
8   public void declareOutputFields(OutputFieldsDeclarer declarer) {
9     declarer.declare(new Fields("word"));
10  }
11
12  @Override
13  public Map<String, Object> getComponentConfiguration() {
14    return null;
15  }
16 }
17
18 public static class WordCount extends BaseBasicBolt {
19   Map<String, Integer> counts = new HashMap<String, Integer>();
20
21     @Override
22   public void execute(Tuple tuple, BasicOutputCollector collector) {
23     String word = tuple.getString(0);
24     Integer count = counts.get(word);
25      if (count == null) count = 0;
26     count++;
27     counts.put(word, count);
28     collector.emit(new Values(word, count));
29 }
30
31   @Override
32   public void declareOutputFields(OutputFieldsDeclarer declarer) {
33     declarer.declare(new Fields("word", "count"));
34   }
35 }
36
37 public static void main(String[] args) throws Exception {
38   TopologyBuilder builder = new TopologyBuilder();
39   builder.setSpout("spout", new RandomSentenceSpout(), 5);
40   builder.setBolt("split", new SplitSentence(), 8).shuffleGrouping("spout");
41   builder.setBolt("count", new WordCount(), 12).fieldsGrouping("split", new
       Fields("word"));
42   Config conf = new Config();
43   conf.setDebug(true);
44   conf.setNumWorkers(3);
45   StormSubmitter.submitTopology(args[0], conf, builder.createTopology());
46 }
47
```

Listing 3.5: A Java Word Count example in Storm.

Tasks and Grouping

The execution of each and every Spout and Bolt by Storm is called a Task, which is the smallest logical unit of a Topology. In simple words, a Task is either the execution of a Spout or a Bolt. At a given time, each Spout and Bolt can have multiple instances running in multiple separate threads. The execution model in Storm is based on a Master-Worker paradigm, in which Workers receive Tasks from the runtime. Apache Storm has two type of nodes, Nimbus (master node) and Supervisor (worker node). Nimbus is the central component of Apache Storm, running the Storm Topology. Nimbus analyzes the Topology and gathers the tasks to be executed. Then, it distributes each task to an available supervisor. A Supervisor will have one or more worker processes to which it delegates Tasks. Worker processes will spawn as many executors as needed and run the task. Apache Storm uses an internal distributed messaging system for the communication between Nimbus and Supervisors.

Stream grouping controls how the tuples are routed in the Topology. There are four main built-in groupings:

- Shuffle Grouping: tuples are randomly distributed across the Bolt's tasks so that each Bolt is guaranteed to get an equal number of tuples
- Field Grouping: this performs a tuple partitioning, in which tuples with the same field values are sent forward to the same worker executing the bolts
- Global Grouping: all the streams can be grouped and forwarded to one Bolt. This grouping sends tuples generated by all instances of the source to a single target instance.
- All Grouping: sends a single copy of each tuple to all instances of the receiving bolt. This kind of grouping is used to send signals to Bolt (i.e., broadcast).

3.3.7 FlumeJava

FlumeJava [43] is a Java library developed by Google for data-parallel pipelines. It can be considered the ancestor of Google Dataflow. Its core is centered on parallel operators on parallel collections: internally, it implements parallel operations using deferred evaluation, thus the invocation of a parallel operation is stored with its arguments in an internal execution plan graph structure. Once the execution plan for the whole computation has been constructed, FlumeJava optimizes the execution plan by, for example, fusing chains of parallel operations together into a small number of MapReduce operations. FlumeJava supports only batch execution and with a Bulk-synchronous model: the runtime executor traverses the operations in the execution plan in forward topological order, executing each one in turn. The same program can execute completely locally when run on small inputs and using many parallel machines when run on large inputs.

Data Model and Transformations

In FlumeJava, there are several classes representing different data. The core abstraction for representing data is the `PCollection`, containing ordered as well as unordered data. The second core abstraction is the `PTable`, representing data in the form of key-value pairs, where the value is always a PCollection. PCollections are transformed via parallel operations, such as `parallelDo` (i.e., map) or `join`, for instance. Those parallel operations are executed lazily. Each PCollection object is represented internally with a deferred (not yet computed) or materialized (computed) state. A deferred PCollection holds a pointer to the deferred operation that computes it, holding references to the PCollections that are its arguments (which may themselves be deferred or materialized) and the deferred PCollections that are its results. When a FlumeJava operation like parallelDo() is called, it creates a ParallelDo deferred operation object and returns a new deferred PCollection that points to it. The result of executing a series of FlumeJava operations is thus a directed acyclic graph of deferred PCollections and operations, called *the execution plan*. The execution plan is visited by the internal optimizer with the goal of producing the most efficient execution plan.

One of the most important operations is the *MapShuffleCombineReduce* (MSCR), in which FlumeJava optimizer transforms combinations of ParallelDo, GroupByKey, CombineValues, and Flatten operations into single MapReduce step. It extends MapReduce by allowing multiple reducers and combiners. Furthermore it allows each reducer to produce multiple outputs, by removing the requirement that a reducer must produce outputs with the same key as the reducer input, and by allowing pass-through outputs, thereby making it a better target for the optimizer, which improves communication patterns.

The code shown in Listing 3.6 presents a word count example in Apache Crunch [52], the Java library for creating FlumeJava pipelines.

```
public class WordCount {
    public static void main(String[] args) throws Exception {
    // Create an object to coordinate pipeline creation and execution.
    Pipeline pipeline = new MRPipeline(WordCount.class);

    // Reference a given text file as a collection of Strings.
    PCollection<String> lines = pipeline.readTextFile(args[0]);

    // Define a function that splits each line in a PCollection of Strings
    // into a PCollection made up of the individual words in the file.
    PCollection<String> words = lines.parallelDo(
      new DoFn<String, String>() {
        public void process(String line, Emitter<String> emitter) {
          for (String word : line.split("\\s+")) {
            emitter.emit(word);
          }
        }
      },
      Writables.strings() // Indicates the serialization format
    );

    // The count method applies a series of Crunch primitives and returns a
    // map of the top 20 unique words in the input PCollection to their counts.
    // We then read the results of the MapReduce jobs that performed the
    // computations into the client and write them to stdout.
    for (Pair<String, Long> wordCount: words.count().top(20).materialize()) {
      System.out.println(wordCount);
    }
  }
}
```

Listing 3.6: A Complete Java Word Count example in Crunch API for FlumeJava.

3.3.8 Google Dataflow

Google Dataflow SDK [6] is part of the Google Cloud Platform [72]. Here, "Dataflow" is used refering to the "Dataflow model" to describe the processing and programming model of the Cloud Platform. This framework aims to provide a unified model for stream, batch and micro-batch processing. It is built based on four major concepts:

- Pipelines
- PCollections
- Transforms
- I/O Sources and Sinks

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 and transforms in a pipeline are unique to, and owned by, that Pipeline, while multiple Pipelines cannot share data or transforms. A Pipeline can be linear but it can also branch and merge, thus making a Pipeline a directed graph of steps and its construction can create this directed graph by using conditionals, loops, and other common programming structures. It is possible to implement iterative algorithms only if a fixed and known number of iterations is provided, while it may not be easy to implement algorithms where the Pipeline's execution graph depends on the data itself. This happens because of the computation graph which is built in an intermediate language that is then

optimized before being executed. With iterative algorithms, the complete graph structure cannot be known beforehand, so it is necessary to know the number of iteration in advance in order to replicate the subset of stages in the pipeline involved in the iterative step.

The DAG resulting from a Pipeline is optimized and compiled into a language-agnostic representation before execution.

The execution of a Dataflow program is typically launched on the Google Cloud Platform. When run, Dataflow runtime enters the Graph Construction Time phase in which it creates an execution graph on the basis of the Pipeline, including all the Transforms and processing functions. The execution graph is translated into JSON format and it transmitted to the Dataflow service endpoint. Once validated, this graph is translated into a job on the Dataflow service, which automatically parallelizes and distributes the processing logic to the workers allocated by the user to perform the job.

Data Model and Transformations

A **PCollection** represents a potentially large, immutable bag of elements, that can be either bounded or unbounded. Elements in a PCollection can be of any type, but the type must be consistent. However, the user must provide the runtime with the encoding of the data type as a byte string in order to serialize and send data among distributed processing units. Dataflow's Transforms use PCollections, that is, it is the only abstract data type accepted by operations. Each PCollection is local to a specific Pipeline object and has the following properties: 1) It is immutable; each transformation instantiates a new PCollection. 2) It does not support random access to individual elements and, 3) A PCollection is private to a Pipeline.

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 can never be available for processing at any one time and they can be grouped by using windowing to create logical windows of finite size.

A **Transform** represents an operation on PCollections, that is, it is a stage of the Pipeline. It accepts one (or multiple) PCollection(s) as input, performs an operation on the elements in the input PCollection(s), and produces one (or multiple) new PCollection(s) as output. Transforms are applied to an input PCollection by calling the `apply` method on that collection. The output PCollection is the value returned from `PCollection.apply`.

For example, Listing 3.7 shows how to create a Word Count Pipeline.

```
public static void main(String[] args) {
  // Create a pipeline parameterized by commandline flags.
  Pipeline p = Pipeline.create(PipelineOptionsFactory.fromArgs(arg));
  //The Count transform below returns a new PCollection of key/value
  //pairs, where each key represents a unique word in the text.

  p.apply(TextIO.Read.from("gs://..."))   // Read input.
    .apply(ParDo.named("ExtractWords").of(new DoFn<String, String>() {
      @Override
      public void processElement(ProcessContext c) {
        for (String word : c.element().split("[^a-zA-Z']+")) {
          if (!word.isEmpty()) {
            c.output(word);
          }
        }
      }
    }))
  .apply(Count.<String>perElement())
  .apply("FormatResults",
    MapElements.via(new SimpleFunction<KV<String, Long>, String>() {
    @Override
      public String apply(KV<String, Long> input) {
        return input.getKey() + ": " + input.getValue();
      }
    }))
  .apply(TextIO.Write.to("gs://..."));   // Write output.

  // Run the pipeline.
  p.run();
}
```

LISTING 3.7: Java code fragment for a Word Count example in Google Dataflow.

The `ParDo` is the core element-wise transform in Google Dataflow, invoking a user-specified function on each of the elements of the input PCollection to produce zero or more output elements (`flatmap` semantics) collected into an output PCollection. Elements are processed independently, possibly in parallel across distributed cloud resources.

When executing a `ParDo` transform, the elements of the input PCollection are first divided up into a certain number of "bundles" (which can be considered like micro-batches). These are farmed off to distributed worker machines (or run locally). For each bundle of input elements, processing proceeds as follows:

1. A fresh instance of the argument `DoFn` (namely the class argument of the `ParDo` transform) is created on a worker (through deserialization or other means).
2. The `DoFn`'s *startBundle* method is called to initialize the bundle, if overridden.
3. The `DoFn`'s *processElement* method is called on each of the input elements in the bundle.
4. The `DoFn`'s *finishBundle* method is called to complete its work, if overridden.

The `ParDo` can take also additional "side input", that is a state local to the `ParDo` instance or a broadcast value.

Google Dataflow will be further investigated in Chapter 4.

3.3.9 Thrill

Thrill [34] is a prototype of a general purpose big data batch processing framework with a data-flow style programming interface implemented in C++ and exploiting

template meta-programming. Thrill uses arrays rather than multisets as its primary data structure to enable operations like sorting, prefix sums, window scans, or combining corresponding fields of several arrays (zipping). Thrill programs run in a collective bulk-synchronous manner similar to most MPI programs, focusing on fast in-memory computation, but transparently uses external memory when needed. The functional programming style used by Thrill enables easy parallelization, which also works well for shared memory parallelism. A consequence of using C++ is that memory management has to be done explicitly. However, memory management in modern C++11 has been considerably simplified, and Thrill uses reference counting extensively. Thrill's authors wanted to emphasize that, with the advent of C++11 lambda-expressions, it has become easier to use C++ for big data processing using an API comparable to currently popular frameworks like Spark or Flink.

Thrill execution model is similar to MPI programs, where the same program is run on different machines, i.e., SPMD (Single Program, Multiple Data). The binary is executed simultaneously on all machines and communication supports TCP sockets and MPI. Each machine is called a *host* and each thread on a host is called a *worker*. There is no master or driver host, as all communication is done collectively.

Distributed Immutable Arrays

Thrill's data model is based on *distributed immutable array* (DIA). A DIA is an array of items distributed over the cluster, to which no direct access is permitted, that is, it is only possible to apply operations to the array as a whole. In a Thrill program, these operations are used to lazily construct a DIA data-flow graph in C++, as shown in Listing 3.8. The data-flow graph is only executed when an action operation is encountered. How DIA items are actually stored and in what way the operations are executed on the distributed system remains transparent to the user. In the current Thrill prototype, the array is usually distributed evenly among workers, in order. DIAs can contain any C++ data type, provided serialization methods are available. Thrill provides built-in serialization methods for all primitive types and most STL types; only custom non-trivial classes require additional methods. Each DIA operation is implemented as a C++ template class.

```
1  void WordCount(thrill::Context& ctx, std::string input, std::string output) {
2    using Pair = std::pair<std::string, size_t>;
3    auto word_pairs = ReadLines(ctx, input).template FlatMap<Pair>(
4    // flatmap lambda: split and emit each word
5      [](const std::string& line, auto emit) {
6        Split(line, ' ', [&](std::string_view sv) {
7          emit(Pair(sv.to_string(), 1));
8        });
9      });
10   word_pairs.ReduceByKey(
11     // key extractor: the word string
12     [](const Pair& p) { return p.first; },
13     // commutative reduction: add counters
14     [](const Pair& a, const Pair& b) { return Pair(a.first, a.second + b.second)
       ; }
15   )
16   .Map([](const Pair& p) { return p.first + ": " + std::to_string(p.second); })
17   .WriteLines(output);
18 }
```

Listing 3.8: A C++ complete Word Count example in Thrill.

The immutability of a DIA enables functional-style dataflow programming. As DIA operations can depend on other DIAs as inputs, these form a directed acyclic graph (DAG), which is called *the DIA dataflow graph*, where vertexes represent operations and directed edges represent data dependencies.

A DIA remains purely an abstract dataflow between two concrete DIA operations, allowing to apply optimizations such as pipelining or chaining, combining the logic of one or more functions into a single one (called pipeline). All independently parallelizable local operations (e.g., FlatMap, Map) and the first local computation steps of the next distributed DIA operation are packed into one block of optimized binary code. Via this chaining, it is possible to reduce the data movement overhead as well as the total number of operations that need to store intermediate arrays.

3.3.10 Kafka

Apache Kafka [87] is an open-source stream processing platform implemented in Scala and Java, aiming to provide a unified, high-throughput, low-latency platform for handling real-time data feeds. Kafka is based on a producer-consumer model exchanging streams of messages. Each stream of messages of a particular type is called a *topic*: a producer can publish messages to a topic, which are then stored at a set of servers called *brokers*, while a consumer can subscribe to one or more topics from the brokers, and consume the subscribed messages by pulling data from the brokers. A feature of Kafka is that each message stream provides an iterator interface over the stream produced. The consumer can iterate over every message in the stream and, unlike traditional iterators, the message stream iterator never terminates and exposes a blocking behavior until new messages are published. Since Kafka is distributed, a Kafka cluster typically consists of multiple brokers, hence a topic is divided into multiple partitions and each broker stores one or more of those. A consumer consumes messages from a particular partition sequentially.

Producer-Consumer Distributed Coordination

Each producer can publish a message to either a randomly selected partition or a partition semantically determined by a partitioning key and a partitioning function. Kafka implements the concept of *consumer groups* consisting of one or more consumers having subscribed to a set of topics. Different consumer groups consume independently the full set of subscribed messages without communication among groups, since they can be in different processes or machines. The consumer starts a thread to pull data from each owned partition, starting the iterator over the stream from the offset stored in the *offset registry*. As messages get pulled from a partition, the consumer periodically updates the latest consumed offset in the offset registry.

When there are multiple consumers within a group, each will be notified of a broker or consumer change. It is possible for a consumer to try to take ownership of a partition still owned by another consumer. In this case, the first consumer releases all the partitions that it currently owns and retries the re-balance process after a certain timeout.

In this architecture, there is no master node, since consumers coordinates among themselves in a decentralized manner.

3.3.11 Google TensorFlow

Google TensorFlow [2] is a framework specifically designed for machine learning applications, where the data model consists of multidimensional arrays called *tensors* and a program is a composition of operators processing tensors. A TensorFlow application is built as a functional-style expression, where each sub-expression can be given an explicit name. The TensorFlow programming model includes control flow operations and, notably, synchronization primitives (e.g., *MutexAcquire* and

MutexRelease for critical sections). This latter observation implies that TensorFlow exposes the underlying (parallel) execution model to the user, who has to program the eventual coordination of operators concurring over some global state. TensorFlow allows expressing conditionals and loops by means of specific control flow operators such as *For* operator.

Each node has firing rules that depend on the kind of incoming tokens. For example, *control dependencies* edges can carry synchronization tokens: the target node of such edges cannot execute until all appropriate synchronization signals have been received.

TensorFlow replicates actors implementing certain operators (e.g., tensor multiplication) on tensors (input tokens). Hence, each actor is a data-parallel actor operating on intra-task independent input elements—here, multi-dimensional arrays (tensors). Moreover, iterative actors/hierarchical actors (in case of cycles on a subgraph) are implemented with tags similar to the MIT Tagged-Token dataflow machine [28], where the iteration state is identified by a tag and independent iterations are executed in parallel. In iterative computations, it is worthwhile to underline that in TensorFlow an input can enter a loop iteration whenever it becomes available (as in Naiad), instead of imposing barriers after each iteration, as done, for instance, in Flink.

A TensorFlow application

Listing 3.9 presents a code snippet of a TensorFlow application generating the dataflow graph in figure 3.2. The example computes the rectified linear unit function for neural network activations.

```
1  import tensorflow as tf
2  b = tf.Variable(tf.zeros([100]))            # 100-d vector, init to zeroes
3  W = tf.Variable(tf.random_uniform([784,100],-1,1)) # 784x100 matrix w/rnd vals
4  x = tf.placeholder(name="x")                # Placeholder for input
5  relu = tf.nn.relu(tf.matmul(W, x) + b)      # Relu(Wx+b)
6  C = [...]                                   # Cost as a function of Relu
7  s = tf.Session()
8  for step in xrange(0, 10):
9    input = ...construct 100-D input array ...# Create 100-d vector for input
10   result = s.run(C, feed_dict={x: input})   # Fetch cost, feeding x=input
11   print step, result
```

Listing 3.9: Example of a Python TensorFlow code fragment in [2].

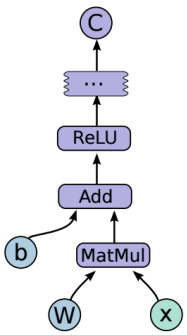


Figure 3.2: A TensorFlow application graph

Figure 3.2 shows a TensorFlow application graph example from [2]. A node of the graph represents a tensor operation, which can also be a data generation one (nodes W, b, x): operators are mapped to actors that take as input single tokens

representing Tensors (multi-dimensional arrays), and are activated once except for iterative computations.

3.3.12 Machine Learning and Deep Learning Frameworks

Besides we described only TensorFlow in our overview, different other Machine and Deep Learning frameworks are present in the community. In the following, we provide common characteristics from both data and programming model perspective of some well known frameworks such as Caffe, Torch and Theano. We do not provide a comprehensive list since focusing on Machine and Deep Learning it is out of the scope of this overview. A comparative study of such frameworks can be found in Bahrampour et al. work [30].

Caffe [82] is a framework for Deep Learning providing the user with state-of-the-art algorithms and a collection of reference models. It is implemented in C++ with Python and MATLAB interfaces. Furthermore, it provides support for offloading computations on GPUs. Caffe was first designed for vision, but it has been adopted and improved by users in speech recognition, robotics, neuroscience, and astronomy. Its data model relies on the *Blob*, a multi-dimensional array used as a wrapper over the actual data being processed, providing also synchronization capability between the CPU and the GPU. From an implementation perspective, a blob is an N-dimensional array stored in a C-contiguous fashion. Blobs provide a unified memory interface holding data; e.g., batches of images, model parameters, and derivatives for optimization.

At a higher level, Caffe considers applications as Dataflow graphs. Each program implements a Deep Network (or simply net): compositional models that are naturally represented as a collection of inter-connected layers that work on chunks of data. Caffe implements Networks in the concept of net, that is, a set of layers connected in a directed acyclic graph (DAG). Caffe does all the bookkeeping for any DAG of layers to ensure correctness of the forward and backward passes. A typical net begins with a data layer that loads from disk and ends with a loss layer that computes the objective for a task such as classification or reconstruction.

Theano [123] is a Python library that allows to define, optimize, and evaluate mathematical expressions involving multi-dimensional arrays, called *tensors*, which are the basic data model for neural network and deep learning programming. Like other frameworks, it provides support for kernel offloading on GPUs. Its programming model is similar to TensorFlow and, at higher level, applications are represented by directed acyclic graphs (DAG): these graphs are composed of interconnected *Apply*, *Variable* and *Op* nodes, the latter representing the application of an *Op* to some *Variable*.

Torch [51] is a machine learning framework implemented in Lua that runs on Lua (JIT) compiler. Torch implements also CUDA and CPU backends. The core data model is represented by Tensors, which extends basic set of types in the Lua programming language, to provide an efficient multi-dimensional array type. This Tensor library provides classic operations including linear algebra operations, implemented in C, leveraging SSE instructions on Intel platforms. Optionally, it can bind linear algebra operations to existing efficient BLAS/Lapack implementations (like Intel MKL). Torch provides a module for building neural networks using a Dataflow based API.

3.4 Fault Tolerance

Fault tolerance is an important aspect to consider when implementing frameworks for analytics. The main benefits of implementing fault tolerance is to guarantee

recovery from faults: consider, for instance, when multiple instances of an application are running and one server crashes for some reason. In this case, the system must be able to recover autonomously and consistently from the system failure. Of course, enabling fault tolerance decreases performances of applications. In Big Data analytics framework, fault tolerance includes both system failure and data recovery with checkpointing techniques, but also a software level fault tolerance. The latter is implemented within the framework itself and requires also application recovery from system failure, e.g., a remote worker node crashes and part of the computation must be restarted. This guarantees must be confirmed also when running streaming applications. Real-time stream processing systems must be always operational, which requires them to recover from all kinds of failures in the system, comprehending also the need to store and recover a certain amount of data already processed by the streaming application. Consider, for instance, Spark Streaming management of fault tolerance. In a Spark Streaming application, data is usually received from sources like Kafka and Flume are buffered in the executor's memory until their processing has completed. If the driver fails, all workers and executors fails as well, and data can not be recovered even if the driver is restarted. To avoid this data loss, Write Ahead Logs have been introduced. Write Ahead Logs (also known as a journal) are used in database and file systems to ensure the durability of any data operations. The intention of the operation is first stored into a durable log , and then the operation is applied to the data. If the system fails in the middle of applying the operation, this can recovered and re-executed by reading the log. It worths nothing that such operations at run-time increase the application execution time and decrease performance and resource utilization, but when running on distributed systems fault tolerance has to be guaranteed.

3.5 Summary

In this Chapter we provided a definition of Big Data and the description of most common tools for analytics and data management. Of course this list is not comprehensive of all tools, since it would be a very hard task to collect all of them in a single place and it goes beyond the scope of this work. Nevertheless, we believe that the descriptions we provided could be useful to help the reader in having a good overview about the topic and the results of the effort of computer scientists in the community.

Chapter 4

High-Level Model for Big Data Frameworks

In this chapter we analyze some well-known tools—Spark, Storm, Flink and Google Dataflow—and provide a common structure underlying all of them, based on the *Dataflow model* [91] that we identified as the common model that better describes all levels of abstraction, from the user-level API to the execution model. We instantiate the Dataflow model into a stack of layers where each layer represents a dataflow graph/model with a different meaning, describing a program from what is exposed to the programmer down to the underlying execution model layer.

4.1 The Dataflow Layered Model

With the increasing number of Big Data analytics tools, we witness a continuous fight among implementors/vendors in demonstrating how their tools are better than others in terms of performances and expressiveness. In this hype, for a user approaching Big Data analytics (even an educated computer scientist), it might be difficult to have a clear picture of the programming model underneath these tools and the expressiveness they provide to solve some user defined problem. With this in mind, we wanted to understand the features those tools provide to the user in terms of API and how they are related to parallel computing paradigms.

We use the Dataflow model to describe these tools since it is expressive enough to describe the batch, micro-batch and streaming models that are implemented in most tools for Big Data processing. Being all realized under the same common idea, we show how various Big Data analytics tools share almost the same base concepts, differing mostly in their implementation choices.

Furthermore, we put our attention to a problem arising from the high abstraction provided by the model that reflects into the examined tools. Especially when considering stream processing and state management, non-determinism may arise when processing one or more streams in one node of the graph, which is a well-known problem in parallel and distributed computing.

In Section 4.1.1, we outline an architecture that can describe all these models at different levels of abstraction (Fig. 4.1) from the (top) user-level API to the (bottom-level) actual network of processes. In particular, we show how the Dataflow model is general enough to subsume many different levels only by changing the semantics of actors and channels.

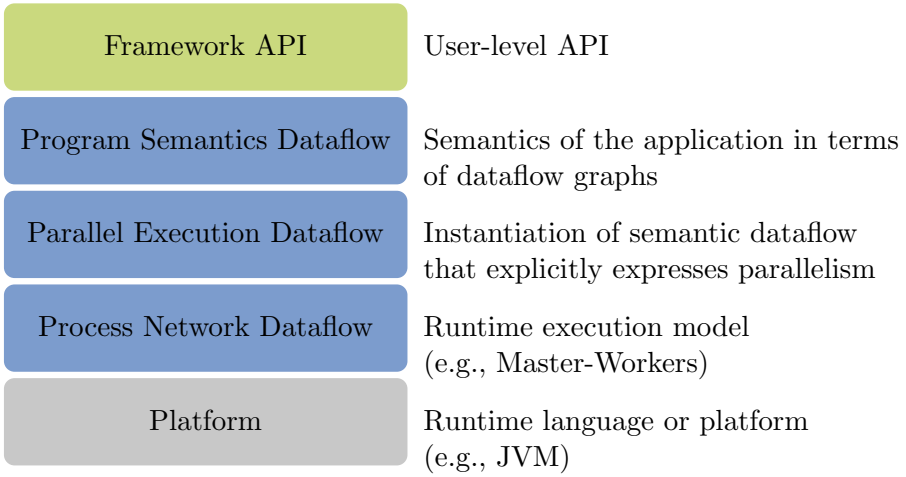


FIGURE 4.1: Layered model representing the levels of abstractions provided by the frameworks that were analyzed.

4.1.1 The Dataflow Stack

The layered model shown in Fig. 4.1 presents five layers, where the three intermediate layers are Dataflow models with different semantics, as described in the paragraphs below. Underneath these three layers is the *Platform* level, that is, the runtime or programming language used to implement a given framework (e.g., Java and Scala in Spark), a level which is beyond the scope of our paper. On top is the *Framework API* level, that describes the user API on top of the Dataflow graph, which will be detailed in Section 4.2. The three Dataflow models in between are as follows.

- *Program Semantics Dataflow*: We claim the API exposed by any of the considered frameworks can be translated into a Dataflow graph. The top level of our layered model captures this translation: programs at this level represent the *semantics* of data-processing applications in terms of Dataflow graphs. Programs at this level do not explicitly express any form of parallelism: they only express data dependencies (i.e., edges) among program components (i.e., actors). This aspect is covered in Section 4.3.
- *Parallel Execution Dataflow*: This level, covered in Section 4.4, represents an instantiation of the semantic dataflows in terms of processing elements (i.e., actors) connected by data channels (i.e., edges). Independent units—not connected by a channel—may execute in parallel. For example, a semantic actor can be replicated to express *data parallelism*, so that the given function can be applied to independent input data.
- *Process Network Dataflow*: This level, covered in Section 4.5, describes how the program is effectively deployed and executed onto the underlying platform. Actors are concrete computing entities (e.g., processes) and edges are communication channels. The most common approach is for the actual network to be a Master-Workers task executor.

4.2 Programming Models

Data-processing applications are generally divided into *batch* vs.*stream* processing. Batch programs process one or more *finite* datasets to produce a resulting finite output dataset, whereas stream programs process possibly unbounded sequences of data, called *streams*, doing so in an incremental manner. Operations over streams

may also have to respect a total data ordering—for instance, to represent time ordering.

Orthogonally, we divide the frameworks' user APIs into two categories: *declarative* and *topological*. Spark, Flink and Google Dataflow belong to the first category—they provide batch or stream processing in the form of operators over collections or streams—whereas Storm, Naiad, Dryad belong to the second one—they provides an API explicitly based on building graphs.

4.2.1 Declarative Data Processing

This model provides building blocks as data collections and operations on those collections. The data model follows domain-specific operators, for instance, relational algebra operators that operate on data structured with the key-value model.

Declarative batch processing applications are expressed as methods on objects representing collections (e.g. Spark, Google Dataflow and Flink) or as functions on values (e.g. *PCollections*, in Google Dataflow): these are algebras on finite datasets, whose data can be ordered or not. APIs with such operations are exposing a functional-like style. Here are three examples of operations with their (multiset-based) semantics:[1]

$$\texttt{groupByKey}(a) = \{(k, \{v : (k, v) \in a\})\} \tag{4.1}$$

$$\texttt{join}(a, b) = \{(k, (v_a, v_b)) : (k, v_a) \in a \wedge (k, v_b) \in b\} \tag{4.2}$$

$$\texttt{map}\langle f \rangle(a) = \{f(v) : v \in a\} \tag{4.3}$$

The `groupByKey` unary operation groups tuples sharing the same key (i.e., the first field of the tuple); thus it maps multisets of type $(K \times V)^*$ to multisets of type $(K \times V^*)^*$. The binary `join` operation merges two multisets by coupling values sharing the same key. Finally, the unary higher-order `map` operation applies the kernel function f to each element in the input multiset.

Declarative stream processing programs are expressed in terms of an algebra on eventually unbounded data (i.e., stream as a whole) where data ordering eventually matters. Data is usually organized in tuples having a key field used for example to express the position of each stream item with respect to a global order—a global timestamp—or to partition streams into substreams. For instance, this allows expressing relational algebra operators and data grouping. In a stream processing scenario, we also have to consider two important aspects: state and windowing; those are discussed in Section 4.2.3.

Apache Spark implements batch programming with a set of operators, called *transformations*, that are uniformly applied to whole datasets called *Resilient Distributed Datasets* (RDD) [131], which are immutable multisets. For stream processing, Spark implements an extension through the Spark Streaming module, providing a high-level abstraction called *discretized stream* or *DStream* [133]. Such streams represent results in continuous sequences of RDDs of the same type, called *micro-batch*. 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 simple translation $\texttt{op}(a) = [\texttt{op}(a_1), \texttt{op}(a_2), \ldots]$, where $[\cdot]$ refers to a possibly unbounded ordered sequence, $a = [a_1, a_2, \ldots]$ is a DStream, and each item a_i is a micro-batch of type RDD.

Apache Flink's main focus is on stream programming. The abstraction used is the DataStream, which is 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

[1] Here, $\{\cdot\}$ denotes *multisets* rather than sets.

multiset—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. Stateful stream operators and iterative batch processing are discussed in Section 4.2.3.

Google Dataflow [6] provides a unified programming and execution 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. Its programming model is based on three main entities: Pipeline, PCollection and Transformation. Transformations are basically Pipelines' stages, operating on PCollections, that are private to each Pipeline. A PCollection represents a potentially large, immutable bag of elements, that can be either bounded or unbounded. Elements in a PCollection can be of any type, but the type must be consistent. One of the main Transformations in Google Dataflow is the `ParDo` operation, used for generic parallel processing. The argument that provided to ParDo must be a subclass of a specific type provided by the Dataflow SDK, called `DoFn`. The ParDo takes each element in an input PCollection, performs some user defined processing function on that element, and then emits zero, one, or multiple elements to an output PCollection. The function provided is invoked independently, and in parallel, on multiple worker instances in a Dataflow job.

4.2.2 Topological Data Processing

Topological programs are expressed as graphs, built by explicitly connecting processing nodes and specifying the code executed by nodes.

Apache Storm is a framework that only targets stream processing. 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. Most of the logic of a computation goes into Bolts, such as functions, filters, streaming joins or streaming aggregations. A Topology is the composition of Spouts and Bolts resulting in a network. Storm uses *tuples* as its data model, that is, named lists of values of arbitrary type. Hence, Bolts are parametrized with per-tuple kernel code. Each time a tuple is available from some input stream, the kernel code gets activated to work on that input tuple. Bolts and Spouts are locally stateful, as we discuss in Section 4.2.3, while no global consistent state is supported. Yet, globally stateful computations can be implemented since the kernel code of Spouts and Bolts is arbitrary. However, eventual global state management would be the sole responsibility of the user, who has to be aware of the underlying execution model in order to ensure program coordination among Spouts and Bolts. It is also possible to define cyclic graphs by way of feedback channels connecting Bolts.

While Storm targets single-tuple granularity in its base interface, the Trident API is an abstraction that provides declarative stream processing on top of Storm. Namely, Trident processes streams as a series of micro-batches belonging to a stream considered as a single object.

4.2.3 State, Windowing and Iterative Computations

Frameworks providing *stateful* stream processing make it possible to express modifications (i.e., side-effects) to the system state that will be visible at some future point. If the state of the system is *global*, then it can be accessed by all system components. Another example of global state is the one managed in Naiad via tagged tokens in their Timely Dataflow model. On the other hand, *local* states can

be accessed only by a single system component. For example, the `mapWithState` functional in the Spark Streaming API realizes a form of local state, in which successive executions of the functional see the modifications to the state made by previous ones. Furthermore, state can be partitioned by shaping it as a tuple space, following, for instance, the aforementioned key-value paradigm.

Google Dataflow provides a form of state in the form of side-input, provided as additional inputs to a ParDo transform. A side input is a local state accessible by the DoFn object each time it processes an element in the input PCollection. Once specified, this side input can be read from within the ParDo transform's DoFn while processing each element.

Windowing is another concept provided by many stream processing frameworks. A *window* is informally defined as an ordered subset of items extracted from the stream. The most common form of windowing is referred as *sliding window* and it is characterized by the size (how many elements fall within the window) and the sliding policy (how items enter and exit from the window). Spark provides the simplest abstraction for defining windows, since they are just micro-batches over the DStream abstraction where it is possible to define only the window length and the sliding policy. Storm and Flink allow more arbitrary kinds of grouping, producing windows of Tuples and WindowedStreams, respectively. Notice this does not break the declarative or topological nature of the considered frameworks, since it just changes the type of the processed data. Notice also that windowing can be expressed in terms of stateful processing, by considering window-typed state.

Google Dataflow provides three kind of windowing patterns: *fixed windows* defined by a static window size (e.g. per hour, per second). *Sliding windows* defined by a window size and a sliding period. The period may be smaller then the size, to allow overlapping windows. *Session windows* capture some period of activity over a subset of data partitioned by a key. Typically, they are defined by a timeout gap. When used with partitioning (e.g., grouping by key), Grouping transforms consider each PCollection on a per-window basis. GroupByKey, for example, implicitly groups the elements of a PCollection by key first and then by window. Windowing can be used with bounded PCollections, and it considers only the implicit timestamps attached to each element of a PCollection, and data sources that create fixed data sets assign the same timestamp to every element. All the elements are by default part of a single, global window, causing the execution in classic MapReduce batch style.

Finally, we consider another common concept in batch processing, namely *iterative* processing. In Flink, iterations are expressed as the composition of arbitrary DataSet values by iterative operators, resulting in a so-called *IterativeDataSet*. Component DataSets represent for example *step functions*—executed in each iteration—or termination condition—evaluated to decide if iteration has to be terminated. Spark's iteration model is radically simpler, since no specific construct is provided to implement iterative processing. Instead, an RDD (endowed with transformations) can be embedded into a plain sequential loop. Google Dataflow iteration model is similar to Spark's but it's limited. It is possible to implement iterative algorithms only if a fixed and known number of iterations is provided, while it may not be easy to implement algorithms where the pipeline's execution graph depends on the data itself. This happens because of the computation graph build in the intermediate language that is then optimized before being executed. With iterative algorithms, the complete graph structure cannot be known in advance, so it is necessary to know the number of iteration in advance in order to replicate the subset of stages in the pipeline involved in the iterative step.

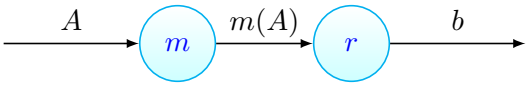


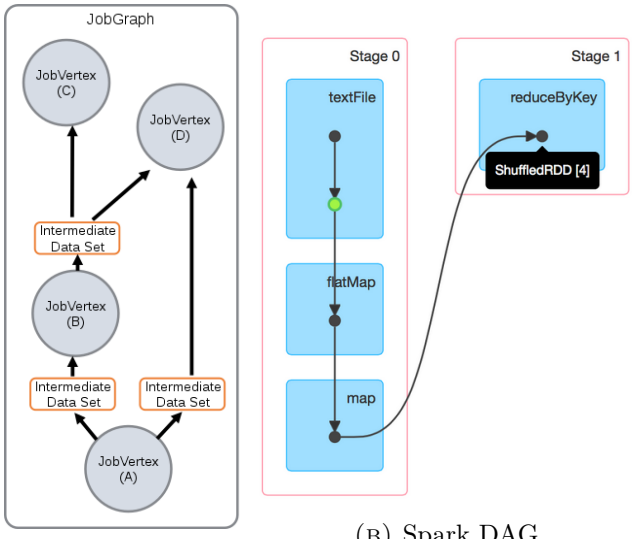
FIGURE 4.2: Functional Map and Reduce dataflow expressing data dependencies.

4.3 Program Semantics Dataflow

This level of our layered model provides a Dataflow representation of the program semantics. Such a model describes the application using operators and data dependencies among them, thus creating a topological view common to all frameworks. This level does not explicitly express parallelism: instead, parallelism is *implicit* through the data dependencies among actors (i.e., among operators), so that operators which have no direct or indirect dependencies can be executed concurrently.

4.3.1 Semantic Dataflow Graphs

A semantic Dataflow graph is a pair $G = \langle V, E \rangle$ where actors V represent operators, channels E represent data dependencies among operators and tokens represent data to be processed. For instance, consider a map function m followed by a reduce function r on a collection A and its result b, represented as the functional composition $b = r(m(A))$. This is represented by the graph in Fig. 4.2, which represents the semantic dataflow of a simple map-reduce program. Notice that the user program translation into the semantic dataflow can be subject to further optimization. For instance, two or more non-intensive kernels can be mapped onto the same actor to reduce resource usage.



(A) Flink JobGraph

(B) Spark DAG

FIGURE 4.3: A Flink JobGraph (4.3a). Spark DAG of the WordCount application (4.3b).

Notably, the Dataflow representation we propose is adopted by the considered frameworks as a pictorial representation of applications. Fig. 4.3b shows the semantic dataflow—called application DAG in Spark—related to the WordCount application, having as operations (in order): 1. read from text file; 2. a `flatMap` operator splitting the file into words; 3. a `map` operator that maps each word into a key-value

pair $(w, 1)$; 4. a `reduceByKey` operator that counts occurrences of each word in the input file. Note that the DAG is grouped into *stages* (namely, Stages 0 and 1), which divide *map* and *reduce* phases. This distinction is related to the underlying parallel execution model and will be covered in Section 4.4. Flink also provides a semantic representation—called JobGraph or *condensed view*— of the application, consisting of operators (JobVertex) and intermediate results (IntermediateDataSet, representing data dependencies among operators). Fig. 4.3a(b) presents a small example of a JobGraph.

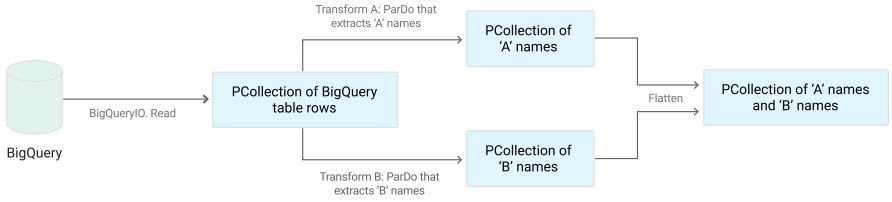


FIGURE 4.4: Example of a Google Dataflow Pipeline merging two PCollections.

In Fig. 4.4 it is reported the semantics dataflow representing an application where, after branching into two PCollections, namely `A` and `B`, the pipeline merges the two together into a single PCollection. This PCollection contains all names that begin with either `A` or `B`. The two PCollections are then merged into a final PCollection after the *Flatten* operator has merged PCollections of the same type. We can highlight two aspects in this example. The first one is the branching node: this node's semantics is to broadcast elements of the PCollection to *A-extractor* and *B-extractor* nodes, thus duplicating the input collection. The second aspect is the merging node, namely the *Flatten* node: it is a transform in the Google Dataflow SDK merging multiple PCollection, having a from-all input policy and producing a single output value.

4.3.2 Tokens and Actors Semantics

Although the frameworks provide a similar semantic expressiveness, some differences are visible regarding the meaning of tokens flowing across channels and how many times actors are activated.

When mapping a Spark program, tokens represent RDDs and DStreams for batch and stream processing respectively. Actors are operators—either transformations or actions in Spark nomenclature—that transform data or return values (in-memory collection or files). Actors are activated only once in both batch and stream processing, since each collection (either RDD or DStreams) is represented by a single token. In Flink the approach is similar: actors are activated only once in all scenarios except in iterative algorithms, as we discuss in Section 4.3.3. Tokens represent DataSets and DataStreams that identify whole datasets and streams respectively. Google Dataflow follows an approach similar to Spark, in which tokens represent PCollections either in batch and stream processing. Actors are represented by Transforms accepting PCollections in input and producing PCollections in output.

Storm is different since tokens represent a single item (called Tuple) of the stream. Consequently, actors, representing (macro) dataflow operators, are activated each time a new token is available.

From the discussion above, we can note that Storm's actors follow a *from-any* policy for consuming input tokens, while the other frameworks follow a *from-all* policy as

in the basic Dataflow model. In all the considered frameworks, output tokens are broadcast onto all channels going out of a node.

4.3.3 Semantics of State, Windowing and Iterations

In Section 4.2.3 we introduced stateful, windowing and iterative processing as convenient tools provided by the considered frameworks.

From a Dataflow perspective, stateful actors represent an extension to the basic model—as we sketched in Section 2.3.2—only in case of global state. In particular, globally-stateful processing breaks the functional nature of the basic Dataflow model, inhibiting for instance to reason in pure functional terms about program semantics (cf. Section 4.6). Conversely, locally-stateful processing can be emulated in terms of the pure Dataflow model, as discussed in [91]. As a direct consequence, windowing is not a proper extension since windows can be stored within each actor's local state [60]. However, the considered frameworks treat windowing as a primitive concept. This can be easily mapped to the Dataflow domain by just considering tokens of proper types.

Finally, iterations can be modeled by inserting loops in semantic dataflows. In this case, each actor involved in an iteration is activated each time a new token is available and the termination condition is not met. This implementation of iterative computations is similar to the hierarchical actors of Lee & Parks [91], used to encapsulate subgraphs modeling iterative algorithms.

4.4 Parallel Execution Dataflow

This level represents parallel implementations of semantic dataflows. As in the previous section, we start by introducing the approach and then we describe how the various frameworks instantiate it and what are the consequences this brings to the runtime.

The most straightforward source of parallelism comes directly from the Dataflow model, namely, independent actors can run in parallel. Furthermore, some actors can be replicated to increase parallelism by making replicas work over a *partition* of the input data—that is, by exploiting full *data parallelism*. This is the case, for instance, of the `map` operator defined in Section 4.2.1. Both the above schemas are referred as *embarrassingly parallel* processing, since there are no dependencies among actors. Note that introducing data parallelism requires partitioning input tokens into sub-tokens, distributing those to the various worker replicas, and then aggregating the resulting sub-tokens into an appropriate result token—much like `scatter`/`gather` operations in message passing programs. Finally, in case of dependent actors that are activated multiple times, parallelism can still be exploited by letting tokens "flow" as soon as each activation is completed. This well-known schema is referred as *stream/pipeline* parallelism.

Fig. 4.5 shows a parallel execution dataflow for the MapReduce semantic dataflow in Fig. 4.2. In this example, the dataset A is divided in 8 independent partitions and the map function m is executed by 8 actor replicas; the reduce phase is then executed in parallel by actors enabled by the incoming tokens (namely, the results) from their "producer" actors.

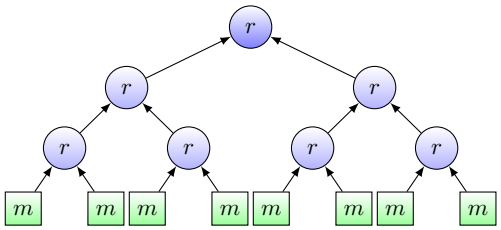
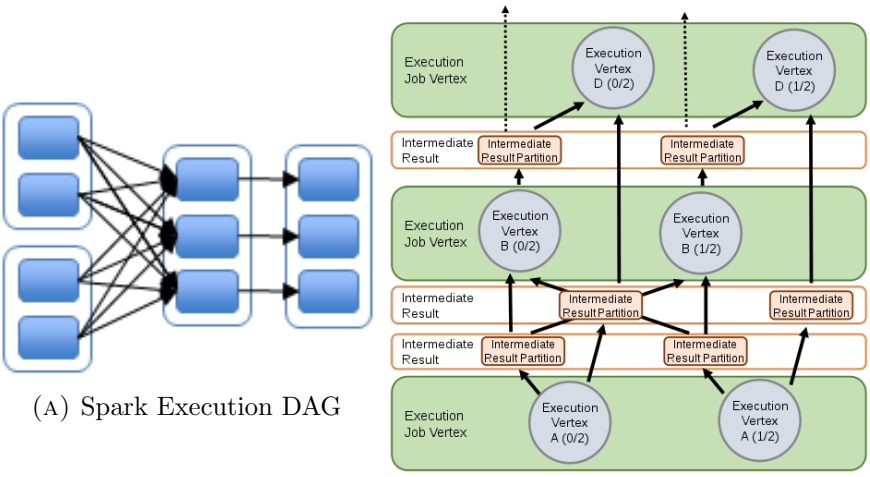


FIGURE 4.5: MapReduce execution dataflow with maximum level of parallelism reached by eight *map* instances.



(A) Spark Execution DAG

(B) Flink Execution Graph

FIGURE 4.6: Spark and Flink execution DAGs.

Spark identifies its parallel execution dataflow by a DAG such as the one shown in Fig. 4.6a, which is the input of the DAG Scheduler entity. This graph illustrates two main aspects: first, the fact that many parallel instances of actors are created for each function and, second, the actors are grouped into the so called *Stages* that are executed in parallel if and only if there is no dependency among them. Stages can be considered as the hierarchical actors in [91]. The actors grouping in stages brings another strong consequence, derived from the implementation of the Spark runtime: each stage that depends on one or more previous stages has to wait for their completion before execution. The depicted behavior is analogous to the one encountered in the Bulk Synchronous Parallelism paradigm (BSP) [128]. In a BSP algorithm, as well as in a Spark application, a computation proceeds in a series of global *supersteps* consisting in: 1) Concurrent computation, in which each actor executes its business code on its own partition of data; 2) Communication, where actors exchange data between themselves if necessary (the so called *shuffle* phase); 3) Barrier synchronization, where actors wait until all other actors have reached the same barrier.

Flink transforms the JobGraph (Fig. 4.3a) into the Execution Graph [42] (Fig. 4.6b), in which the JobVertex (a hierarchical actor) is an abstract vertex containing a certain number of ExecutionVertexes (actors), one per parallel sub-task. A key difference compared to the Spark execution graph is that a dependency does not represent a barrier among actors or hierarchical actors: instead, there is effective tokens pipelining and actors can be fired concurrently. This is a natural implementation for stream processing, but in this case, since the runtime is the same, it applies to batch processing applications as well. Conversely, iterative processing is implemented according to the BSP approach: one evaluation of the step function

	Spark	**Flink**	**Google Dataflow**
Graph specification	Implicit, OO-style chaining of transformations	Idem	Idem
DAG	Join operation	Idem	Join and Merge operation
Tokens	RDD	DataSet	PCollection
Nodes	Transformations from RDD to RDD	Transformations from DataSet to DataSet	Transformations from PCollection to PCollection
Parallelism	Data parallelism in transformations + Inter-actor, task parallelism, limited by per-stage BSP	Data parallelism in transformations + Inter-actor task parallelism	Idem
Iteration	Using *repetitive sequential executions* of the graph	Using `iterate & iterateDelta`	Using *repetitive sequential executions* of the graph

Table 4.1: Batch processing.

on all parallel instances forms a *superstep* (again a hierarchical actor), which is also the granularity of synchronization; all parallel tasks of an iteration need to complete the superstep before the next one is initiated, thus behaving like a *barrier* between iterations.

Storm creates an environment for the execution dataflow similar to the other frameworks. Each actor is replicated to increase the inter-actor parallelism and each group of replicas is identified by the name of the Bolt/Spout of the semantics dataflow they originally belong to, thus instantiating a hierarchical actor. Each of these actors (actors group) represents data parallel tasks without dependencies. Since Storm is a stream processing framework, pipeline parallelism is exploited. Hence, while an actor is processing a token (tuple), an upstream actor can process the next token concurrently, increasing both data parallelism within each actors group and task parallelism among groups.

In Google Dataflow the execution of a Dataflow program is typically launched on the Google Cloud Platform. When run, Dataflow runtime enters the Graph Construction Time phase in which it creates an execution graph on the basis of the Pipeline, including all the Transforms and processing functions. The parallel execution dataflow is similar to the one in Spark and Flink, and parallelism is expressed in terms of data parallelism in Transforms (e.g., ParDo function) and inter-actor parallelism on independent Transforms. In Google Dataflow nomenclature, this graph is called Execution Graph. Similarly to Flink, pipeline parallelism is exploited among successive actors.

	Google Dataflow	**Storm**	**Spark**	**Flink**
Graph specification	Implicit, OO-style chaining of transformations	Explicit, Connections between *bolts*	Implicit, OO-style chaining of transformations	Implicit, OO-style chaining of transformations
DAG	Join and Merge operation	Multiple incoming/outgoing connections	Join operation	Join operation
Tokens	PCollection	Tuple (fine-grain)	DStream	DataStream
Nodes	Transformations from PCollection to PCollection	Stateful with "arbitrary" emission of output tuples	Transformations from DStream to DStream	Transformations from DataStream to DataStream
Parallelism	Data parallelism in transformations + Inter-actor task parallelism	Data parallelism between different bolt instances + Stream parallelism between stream items by bolts	Analogous to Spark Batch parallelism	Analogous to Flink Batch parallelism+ Stream parallelism between stream items

TABLE 4.2: Stream processing comparison between Google Dataflow, Storm, Spark and Flink.

Summarizing, in sections 4.3 and 4.4 we showed how the considered frameworks can be compared under the very same model from both a semantic and a parallel implementation perspective. The comparison is summarized in Tables 4.1 for batch processing and 4.2 for stream processing.

4.5 Execution Models

This layer shows how the program is effectively executed, following the process and scheduling-based categorization described in Section 2.3.2.

4.5.1 Scheduling-based Execution

In Spark, Flink and Storm, the resulting process network dataflow follows the Master-Workers pattern, where actors from previous layers are transformed into tasks. Fig. 4.7a shows a representation of the Spark Master-Workers runtime. We will use this structure also to examine Storm and Flink, since the pattern is similar for them: they differ only in how tasks are distributed among workers and how the inter/intra-communication between actors is managed.

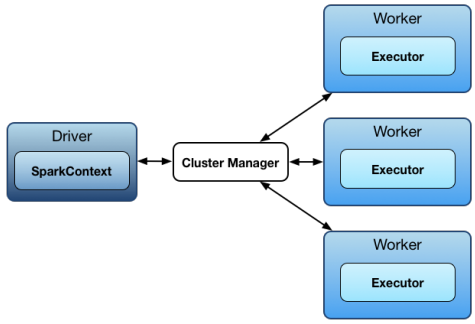
The Master has total control over program execution, job scheduling, communications, failure management, resource allocations, etc. The master also relies on a cluster manager, an external service for acquiring resources on the cluster (like Mesos, YARN or Zookeper).

The master is the one that knows the semantic dataflow representing the current application, while workers are completely agnostic about the whole dataflow: they only obtain tasks to execute, that represent actors of the execution dataflow the master is running. It is only when the execution is effectively launched that the semantic dataflow is built and eventually optimized to obtain the best execution plan (Flink, Google Dataflow). With this postponed evaluation, the master creates the parallel execution dataflow to be executed.

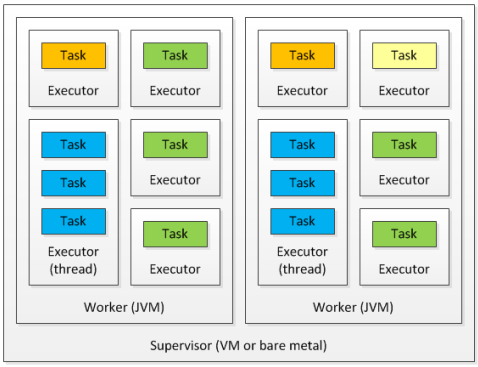
Each framework has its own instance of the master entity: in Spark it is called SparkContext, in Flink it is the JobManager and in Storm it is called Nimbus, in Google Dataflow it is called Cloud Dataflow Service. In Storm and Flink, the data distribution is managed in a decentralized manner, that is, it is delegated to each executor, since they use pipelined data transfers and forward tokens as soon as they are produced. For efficiency, in Flink tuples are collected in a buffer which is sent over the network once it is full or reach a certain time threshold. In Spark batch, data can be possibly dispatched by the master but typically each worker get data from a DFS. In Spark streaming, the master is the one responsible for data distribution: it discretizes the stream into micro-batches that are buffered into workers' memory.

The master generally keeps track of distributed tasks, decides when to schedule the next tasks, reacts to finished vs. failed tasks, keeps track of the semantic dataflow progress, and orchestrates collective communications and data exchange among workers. This last aspect is crucial when executing the so-called *shuffle operation*, which implies a data exchange among executors. Whereas workers do not have any information about others, to exchange data they have to request information to the master and, moreover, specify they are ready to send/receive data.

In Google Dataflow the Master is represented by the *Cloud Dataflow* managed service that deploys and execute the DAG representing the application, built during the Graph Construction Time (see Sect. 4.4). Once on the Dataflow service, the DAG becomes a Dataflow Job. The Cloud Dataflow managed service automatically



(A) Master-Workers



(B) Worker hierarchy

FIGURE 4.7: Master-Workers structure of the Spark runtime (a) and Worker hierarchy example in Storm (b).

partitions data and distributes the Transforms code to Compute Engine instances (Workers) for parallel processing.

Workers are nodes executing the actor logic, namely, a worker node is a process in the cluster. Within a worker, a certain number of parallel executors is instantiated, that execute tasks related to the given application. Workers have no information about the dataflow at any level since they are scheduled by the master. Despite this, the different frameworks use different nomenclatures: in Spark, Storm and Flink cluster nodes are decomposed into *Workers*, *Executors* and *Tasks*.

A Worker is a node of the cluster, i.e., a Spark worker instance. A node may host multiple Worker instances. An Executor is a (parallel) process that is spawned in a Worker process and it executes Tasks, which are the actual kernel of an actor of the dataflow. Fig. 4.7b illustrates this structure in Storm, an example that would also be valid for Spark and Flink.

In Google Dataflow, workers are called *Google Compute Engine*, and occasionally are referred to as Workers or VMs. The Dataflow managed service deploys Compute Engine virtual machines associated with Dataflow jobs using Managed Instance Groups. A Managed Instance Group creates multiple Compute Engine instances from a common template and allows the user to control and manage them as a group. The Compute Engines execute both serial and parallel code (e.g., `ParDo` parallel code) related to a job (parallel execution DAG).

4.5.2 Process-based Execution

In TensorFlow, actors are effectively mapped to threads and possibly distributed on different nodes. The cardinality of the semantic dataflow is preserved, as each actor node is instantiated into one node, and the allocation is decided using a placement algorithm based on cost model optimization. This model is statically estimated based on heuristics or on previous dataflow execution of the same application. The dataflow is distributed on cluster nodes and each node/Worker may host one or more dataflow actors/Tasks, that internally implement data parallelism with a pool of threads/Executors working on Tensors. Communication among actors is done using the send/receive paradigm, allowing workers to manage their own data movement or to receive data without involving the master node, thus decentralizing the logic and the execution of the application.

As we have seen in Sec. 3.3, a sort of mixed model is proposed by Naiad. Nodes represent data-parallel computations. Each computer or thread, called *shard*, executes the entire dataflow graph. It keeps its fraction of the state of all nodes resident in local memory throughout, as for a scheduling based execution. Execution occurs in a coordinated fashion, with all shards processing the same node at any time, and graph edges are implemented by channels that route records between shards as required. There is no Master entity directing the execution: each shard is autonomous also in fault tolerance and recovering.

4.6 Limitations of the Dataflow Model

Reasoning about programs using the Dataflow model is attractive since it makes the program semantics independent from the underlying execution model. In particular, it abstracts away any form of parallelism due to its pure functional nature. The most relevant consequence, as discussed in many theoretical works about Kahn Process Network and similar models—such as Dataflow—is the fact that all computations are *deterministic*.

Conversely, many parallel runtime systems exploit nondeterministic behaviors to provide efficient implementations. For example, consider the Master-Workers pattern discussed in Section 4.5. A naive implementation of the Master node distributes tasks to N Workers according to a round-robin policy—task i goes to worker $i \pmod N$—which leads to a deterministic process. An alternative policy, generally referred as *on-demand*, distributes tasks by considering the load level of each worker, for example, to implement a form of load balancing. The resulting processes are clearly nondeterministic, since the mapping from tasks to workers depends on the relative service times.

Non-determinism can be encountered at all levels of our layered model in Fig. 4.1. For example, actors in Storm's topologies consume tokens from incoming streams according to a from-any policy—process a token from any non-empty input channel—thus no assumption can be made about the order in which stream tokens are processed. More generally, the semantics of stateful streaming programs depends on the order in which stream items are processed, which is not specified by the semantics of the semantic dataflow actors in Section 4.3. As a consequence, this prevents from reasoning in purely Dataflow—i.e., functional—terms about programs in which actor nodes include arbitrary code in some imperative language (e.g., shared variables).

4.7 Summary

In this chapter we analyzed Spark, Storm, Flink and Google Dataflow by showing the common structure underlying all of them, based on the *Dataflow model*. We provided a stack of layers that can be useful to the reader to understand how the same ideas, either if implemented differently, are common to these frameworks. The proposed stack is composed by the following layers: *Program Semantics Dataflow* representing the *semantics* of data-processing applications in terms of Dataflow graphs, where no form of parallelism is expressed; *Parallel Execution Dataflow*: represents an instantiation of the semantic dataflows in terms of processing elements. For example, a semantic actor can be replicated to express *data parallelism*, so that the given function can be applied to independent input data; finally, the *Process Network Dataflow* describes how the program is effectively deployed and executed onto the underlying platform.

Chapter 5

PiCo Programming Model

In this Chapter, we propose a new programming model based on Pipelines and operators, which are the building blocks of PiCo programs [65]. In the model we propose, we use the term Pipeline to denote a workflow that processes data collections—rather than a computational process—as is common in the data processing community [72].

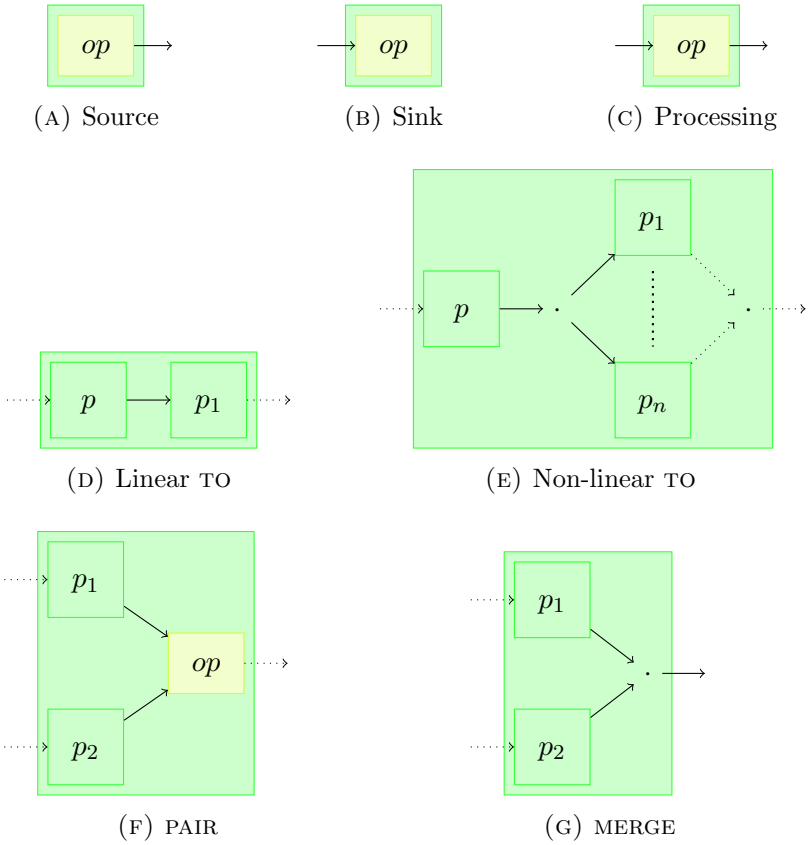
The novelty with respect to other frameworks is that all PiCo operators are polymorphic with respect to data types. This makes it possible to 1) re-use the same algorithms and pipelines on different data models (e.g., streams, lists, sets, etc); 2) reuse the same operators in different contexts, and 3) update operators without affecting the calling context, i.e., the previous and following stages in the pipeline. Notice that in other mainstream frameworks, such as Spark, the update of a pipeline by changing a transformation with another is not necessarily trivial, since it may require the development of an input and output proxy to adapt the new transformation for the calling context.

In the same line, we provide a formal framework (i.e., typing and semantics) that characterizes programs from the perspective of how they transform the data structures they process—rather than the computational processes they represent. This approach allows to reason about programs at an abstract level, without taking into account any aspect from the underlying execution model or implementation.

This chapter proceeds as follows. We formally define the syntax of a program, which is based on Pipelines and operators whereas it hides the data structures produced and generated by the program. We define the Program Semantics layer of the Dataflow stack as it has been defined in [120]. Then we provide the formalization of a minimal type system defining legal compositions of operators into Pipelines. Finally, we provide a semantic interpretation that maps any PiCo program to a functional Dataflow graph, representing the transformation flow followed by the processed collections.

5.1 Syntax

We propose a programming model for processing data collections, based on the Dataflow model. The building blocks of a PiCo program are *Pipelines* and *Operators*, which we investigate in this section. Conversely, *Collections* are not included in the syntax and they are introduced in Section 5.2.1 since they contribute at defining the type system and the semantic interpretation of PiCo programs.



(A) Source (B) Sink (C) Processing

(D) Linear TO (E) Non-linear TO

(F) PAIR (G) MERGE

FIGURE 5.1: Graphical representation of PiCo Pipelines

Pipeline	Structural properties	Behavior
NEW *op*	-	data is processed by operator *op* (i.e., *unary* Pipeline)
TO $p\ p_1\ \ldots\ p_n$	associativity for linear Pipelines: TO (TO $p_A\ p_B$) $p_C \equiv$ TO p_A (TO $p_B\ p_C$) $\equiv$ $p_A \mid p_B \mid p_C$ destination commutativity: TO $p\ p_1 \ldots p_n \equiv$ TO $p\ p_{\pi(1)} \ldots p_{\pi(n)}$ for any π permutation of $1..n$	data from Pipeline p is sent to all Pipelines p_i (i.e., broadcast)
PAIR $p_1\ p_2\ op$	-	data from Pipelines p_1 and p_2 are pair-wise processed by operator *op*
MERGE $p_1\ p_2$	associativity: MERGE (MERGE $p_1\ p_2$) $p_3 \equiv$ MERGE p_1 (MERGE $p_2\ p_3$) $\equiv$ $p_1 + p_2 + p_3$ commutativity: MERGE $p_1\ p_2 \equiv$ MERGE $p_2\ p_1$	data from Pipelines p_1 and p_2 are merged, respecting the ordering in case of ordered collections

TABLE 5.1: Pipelines

5.1.1 Pipelines

The cornerstone concept in the Programming Model is the *Pipeline*, basically a DAG-composition of processing *operators*. Pipelines are built according to the following grammar[1]:

⟨*Pipeline*⟩ ::= NEW ⟨***unary**-operator*⟩
| TO ⟨*Pipeline*⟩ ⟨*Pipeline*⟩ ... ⟨*Pipeline*⟩
| PAIR ⟨*Pipeline*⟩ ⟨*Pipeline*⟩ ⟨***binary**-operator*⟩
| MERGE ⟨*Pipeline*⟩ ⟨*Pipeline*⟩

We categorize Pipelines according to the number of collections they take as input and output:

- A source Pipeline takes no input and produces one output collection
- A sink Pipeline consumes one input collection and produces no output
- A processing Pipeline consumes one input collection and produces one output collection

A pictorial representation of Pipelines is reported in Figure 5.1. We refer to Figs. 5.1a, 5.1b and 5.1c as *unary* Pipelines, since they are composed by a single operator. Figs. 5.1e and 5.1d represent, respectively, linear (i.e., one-to-one) and branching (i.e., one-to-n) TO composition. Figs. 5.1f and 5.1g represent composition of Pipelines by, respectively, pairing and merging. A dotted line means the respective path may be void (e.g., a source Pipeline has void input path). Moreover, as we show in Section 5.2, Pipelines are not allowed to consume more than one input collection, thus both PAIR and MERGE Pipelines must have at least one void input path.

The meaning of each Pipeline is summarized in Table 5.1.

5.1.2 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`⟩ | ⟨`b-combine`⟩

The intuitive meanings of the core operators are summarized in Table 5.2.

[1] For simplicity, here we introduce the non-terminal `unary-`*operator* (resp. `binary-`*operator*) that includes core and partitioning unary (resp. binary) operators.

Operator family	Categorization	Decomposition	Behavior
`map`	unary, element-wise	no	applies a user function to each element in the input collection
`combine`	unary, collective	yes	synthesizes all the elements in the input collection into an atomic value, according to a user-defined policy
`b-map`	binary, pair-wise	yes	the binary counterpart of `map`: applies a (binary) user function to each pair generated by pairing (i.e. zipping/joining) two input collections
`b-combine`	binary, collective	yes	the binary counterpart of `combine`: synthesizes all pairs generated by pairing (i.e. zipping/joining) two input collections
`emit`	produce-only	no	reads data from a source, e.g., regular collection, text file, tweet feed, etc.
`collect`	consume-only	no	writes data to some destination, e.g., regular collection, text file, screen, etc.

TABLE 5.2: Core operator families.

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:

$\langle operator \rangle ::= \langle \texttt{core-}operator \rangle$
$\quad | \quad \langle \texttt{w-}operator \rangle \mid \langle \texttt{p-}operator \rangle \mid \langle \texttt{w-p-}operator \rangle$

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

For those operators *op* not supporting decomposition (cf. Table 5.2), the following structural equivalence holds: $op \equiv \texttt{w-}op \equiv \texttt{p-}op \equiv \texttt{w-p-}op$.

Data-Parallel Operators

Operators in the `map` family are defined according to the following grammar:

$\langle map \rangle ::= \texttt{map}\ f \mid \texttt{flatmap}\ f$

where f is a user-defined function (i.e., the *kernel* function) from a host language.[2] The former produces exactly one output element from each input element (one-to-one user function), whereas the latter produces a (possibly empty) bounded sequence of output elements for each input element (one-to-many user function) and the output collection is the merging of the output sequences.

Operators in the `combine` family synthesize all the elements from an input collection into a single value, according to a user-defined kernel. They are defined according to the following grammar:

$\langle \textit{combine} \rangle ::= \texttt{reduce} \oplus \mid \texttt{fold+reduce} \ \oplus_1 \ z \ \oplus_2$

The former corresponds to the classical reduction, whereas the latter is a two-phase aggregation that consists in the reduction of partial accumulative states (i.e., partitioned folding with explicit initial value). The parameters for the `fold+reduce` operator specify the initial value for each partial accumulator ($z \in S$, the initial value for the folding), how each input item affects the aggregative state ($\oplus_1 : S \times T \to S$, the folding function) and how aggregative states are combined into a final accumulator ($\oplus_2 : S \times S \to S$, the reduce function).

Pairing

Operators in the `b-map` family are intended to be the binary counterparts of `map` operators:

$\langle \textit{b-map} \rangle ::= \texttt{zip-map} \ f \mid \texttt{join-map} \ f$
$\quad \mid \ \texttt{zip-flatmap} \ f \mid \texttt{join-flatmap} \ f$

The binary user function f takes as input pairs of elements, one from each of the input collections. Variants `zip-` and `join-` corresponds to the following pairing policies, respectively:

- zipping of ordered collections produces the pairs of elements with the same position within the order of respective collections
- joining of bounded collections produces the Cartesian product of the input collections

Analogously, operators in the `b-combine` family are the binary counterparts of `combine` operators.

Sources and Sinks

Operators in the `emit` and `collect` families model data collection sources and sinks, respectively:

$\langle \textit{emit} \rangle ::= \texttt{from-file}$ file $\mid \texttt{from-socket}$ socket $\mid \ldots$

$\langle \textit{collect} \rangle ::= \texttt{to-file}$ file $\mid \texttt{to-socket}$ socket $\mid \ldots$

[2] Note that we treat kernels as terminal symbols, thus we do not define the language in which kernel functions are defined; we rather denote this aspect to a specific implementation of the model.

Windowing

Windowing is a well-known approach for overcoming the difficulties stemming from the unbounded nature of stream processing. The basic idea is to process parts of some recent stream history upon the arrival of new stream items, rather than store and process the whole stream each time.

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

Among the various definitions from the literature, for the sake of simplicity we only consider policies producing *sliding windows*, characterized by two parameters, namely, a window size $|W|$—specifying which elements fall into a window—and a sliding factor δ—specifying how the window slides over the stream items. Both parameters can be expressed either in time units (i.e., time-based windowing) or in number of items (i.e., count-based windowing). In this setting, a windowing policy ω is a term $(|W|, \delta, b)$ where b is either `time` or `count`. A typical case is when $|W| = \delta$, referred as a *tumbling* policy.

The meaning of the supported windowing policies will be detailed in semantic terms (Section 5.3.1). Although the PiCo syntax only supports a limited class of windowing policies, the semantics we provide is general enough to express other policies such as session windows [72].

As we will show in Section 5.2, we rely on tumbling windowing to extend bounded operators[3] and have them deal with unbounded collections; for instance, `combine` operators are bounded and require windowing to extend them to unbounded collections.

Partitioning

Logically, partitioning operators take a collection, produces a set (one per group) of sub-collections (with the same type as the input one) and applies the subsequent operation to each sub-collection. Partitioning operators are defined according to the following grammar, where π is a user-defined partitioning policy that maps each item to the respective sub-collection:

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

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

The decomposition by both partitioning and windowing considers the former as the external decomposition, thus it logically produces a set (one per group) of collections of windows:

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

Algorithm 1 A WORD-COUNT Pipeline

$f = \lambda l.\text{list-map } (\lambda w.\,(w, 1))\ (\text{split } l)$
`tokenize` = `flatmap` f

$\oplus = \lambda xy.\,(\pi_1(x), \pi_2(x) + \pi_2(y))$
`keyed-sum` = `p-`(`reduce` $\oplus$) π_1

`file-read` = `from-file` input-file
`file-write` = `to-file` output-file

WORD-COUNT = NEW `tokenize` | NEW `keyed-sum`
FILE-WORD-COUNT = NEW `file-read` | WORD-COUNT | NEW `file-write`

5.1.3 Running Example: The word-count Pipeline

We illustrate a simple WORD-COUNT Pipeline in Algorithm 1. We assume an hypothetical PiCo implementation where the host language provides some common functions over basic types—such as strings and lists—and a syntax for defining and naming functional transformations. In this setting, the functions f and $\oplus$ in the example are user-defined kernels (i.e., functional transformations) and:

- split is a host function mapping a text line (i.e., a string) into the list of words occurring in the line
- list-map is a classical host map over lists
- π_1 is the left-projection partitioning policy (cf. example below, Section 5.3.1, Definition 3)

The operators have the following meaning:

- `tokenize` is a `flatmap` operator that receives lines l of text and produces, for each word w in each line, a pair $(w, 1)$;
- `keyed-sum` is a `p-reduce` operator that partitions the pairs based on w (obtained with π_1, using group-by-word) and then sums up each group to (w, n_w), where w occurs n_w times in the input text;
- `file-read` is an `emit` operator that reads from a text file and generates a list of lines;
- `file-write` is a `collect` operator that writes a bag of pairs (w, n_w) to a text file.

5.2 Type System

Legal Pipelines are defined according to typing rules, described below. We denote the typing relation as $a : \tau$, if and only if there exists a legal inference assigning type τ to the term a.

5.2.1 Collection Types

We mentioned earlier (Section 5.1) that collections are *implicit* entities that flow across Pipelines through the DAG edges. A collection is either *bounded* or *unbounded*; moreover, it is also either *ordered* or *unordered*. A combination of the

[3] We say an operator is *bounded* if it can only deal with bounded collections.

Operator	Type
Unary	
`map`	$T_\sigma \to U_\sigma, \forall \sigma \in \Sigma$
`combine`, `p-combine`	$T_\sigma \to U_\sigma, \forall \sigma \in \Sigma_b$
`w-combine`, `w-p-combine`	$T_\sigma \to U_\sigma, \forall \sigma \in \Sigma_o$
`emit`	$\emptyset \to U_\sigma$
`collect`	$T_\sigma \to \emptyset$
Binary	
`b-map`, `p-b-map`	$T_\sigma \times T'_\sigma \to U_\sigma, \forall \sigma \in \Sigma_b$
`w-b-map`, `w-p-b-map`	$T_\sigma \times T'_\sigma \to U_\sigma, \forall \sigma \in \Sigma_o$

TABLE 5.3: Operator types.

$$\frac{op : T_\sigma \to U_\sigma, \sigma \in \Sigma_o}{\texttt{w-}op\ \omega : T_{\sigma'} \to U_{\sigma'}, \sigma' \in \Sigma_o}\ \texttt{w-}$$

FIGURE 5.2: Unbounded extension provided by windowing

mentioned characteristics defines the *structure type* of a collection. We refer to each structure type with a mnemonic name:

- 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 its structure type and its *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, respectively. 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 $\emptyset$.

5.2.2 Operator Types

Operator types are defined in terms of input/output signatures. The typing of operators is reported in Table 5.3. We do not show the type inference rules since they are straightforward.

From the type specification, we say each operator is characterized by its input and output degrees (i.e., the cardinality of left and right-hand side of the $\to$ symbol, respectively). All operators but `collect` have output degree 1, while `collect` has output degree 0. All binary operators have input degree 2, `emit` has input degree 0 and all the other operators have input degree 1.

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.[4]

As we mentioned in Section 5.2.1, a windowing operator may behave as the unbounded extension of the respective bounded operator. This is formalized by the

[4]For example, an emitter for a finite text file would generate a bounded collection of strings, whereas an emitter for stream of tweets would generate an unbounded collection of tweet objects.

$$\frac{op : \tau}{\text{NEW } op : \tau}\ \text{NEW}$$

$$\frac{p : T_\sigma^\circ \to U_\sigma \quad p_i : U_\sigma \to (V_\sigma^\circ)_i \quad \exists i : (V_\sigma^\circ)_i = V_\sigma}{\text{TO } p\ p_1\ \ldots\ p_n : T_\sigma^\circ \to V_\sigma}\ \text{TO}$$

$$\frac{p : T_\sigma^\circ \to U_\sigma \quad p_i : U_\sigma \to \emptyset}{\text{TO } p\ p_1\ \ldots\ p_n : T_\sigma^\circ \to \emptyset}\ \text{TO}_\emptyset$$

$$\frac{p : T_\sigma^\circ \to U_\sigma \quad p' : \emptyset \to U'_\sigma \quad a : U_\sigma \times U'_\sigma \to V_\sigma^\circ}{\text{PAIR } p\ p'\ a : T_\sigma^\circ \to V_\sigma^\circ}\ \text{PAIR}$$

$$\frac{p : \emptyset \to U_\sigma \quad p' : T_\sigma^\circ \to U'_\sigma \quad a : U_\sigma \times U'_\sigma \to V_\sigma^\circ}{\text{PAIR } p\ p'\ a : T_\sigma^\circ \to V_\sigma^\circ}\ \text{PAIR}'$$

$$\frac{p : T_\sigma^\circ \to U_\sigma \quad p' : \emptyset \to U_\sigma}{\text{MERGE } p\ p' : T_\sigma^\circ \to U_\sigma}\ \text{MERGE}$$

FIGURE 5.3: Pipeline typing

inference rule `w-` that is reported in Figure 5.2: given an operator *op* dealing with ordered structure types (bounded or unbounded), its windowing counterpart `w-`*op* can operate on *any* ordered structure type, including stream. The analogous principle underlies the inference rules for all the `w-` operators.

5.2.3 Pipeline Types

Pipeline types are defined according to the inference rules in Figure 5.3. For simplicity, we use the meta-variable T_σ°, which can be rewritten as either T_σ or $\emptyset$, to represent the optional collection type[5]. The awkward rule TO covers the case in which, in a TO Pipeline, at least one destination Pipeline p_i has non-void output type V_σ; in such case, all the destination Pipelines with non-void output type must have the same output type V_σ, which is also the output type of the resulting Pipeline.

Finally, we define the notion of top-level Pipelines, representing Pipelines that may be executed.

Definition 1. *A* top-level *Pipeline is a* non-empty *Pipeline of type* $\emptyset \to \emptyset$.

Running Example: Typing of word-count

We present the types of the WORD-COUNT components, defined in Section 5.1. We omit full type derivations since they are straightforward applications of the typing rules.

The operators are all unary and have the following types:

$$\begin{array}{ll}
\texttt{tokenize} & : \text{String}_\sigma \to (\text{String} \times \mathbb{N})_\sigma, \forall \sigma \in \Sigma \\
\texttt{keyed-sum} & : (\text{String} \times \mathbb{N})_\sigma \to (\text{String} \times \mathbb{N})_\sigma, \forall \sigma \in \Sigma \\
\texttt{file-read} & : \emptyset_{\text{bag}} \to \text{String}_{\text{bag}} \\
\texttt{file-write} & : (\text{String} \times \mathbb{N})_{\text{bag}} \to \emptyset_{\text{bag}}
\end{array}$$

[5] We remark the optional collection type is a mere syntactic rewriting, thus it does not represent any additional feature of the typing system.

Pipelines have the following types:

$$\begin{array}{ll} \textsc{word-count} & : \text{String}_\sigma \rightarrow (\text{String} \times \mathbb{N})_\sigma, \forall \sigma \in \Sigma \\ \textsc{file-word-count} & : \emptyset \rightarrow \emptyset \end{array}$$

We remark that WORD-COUNT is polymorphic whereas FILE-WORD-COUNT is a top-level Pipeline.

5.3 Semantics

We propose an interpretation of Pipelines in terms of semantic Dataflow graphs, as defined in [120]. Namely, we propose the following mapping:

- Collections ⇒ Dataflow tokens
- Operators ⇒ Dataflow vertexes
- Pipelines ⇒ Dataflow graphs

Note that collections in semantic Dataflow graphs are treated as a whole, thus they are mapped to single Dataflow tokens that flow through the graph of transformations. In this setting, semantic operators (i.e., Dataflow vertexes) map an input collection to the respective output collection upon a single firing.

5.3.1 Semantic Collections

Dataflow tokens are data collections of T-typed elements, where T is the data type of the collection. Unordered collections are semantically mapped to multi-sets, whereas ordered collections are mapped to sequences.

We denote an unordered data collection of data type T with the following, "$\{ \ldots \}$" being interpreted as a multi-set (i.e., unordered collection with possible multiple occurrences of elements):

$$m = \{m_0, m_1, \ldots, m_{|m|-1}\} \tag{5.1}$$

A sequence (i.e., semantic ordered collection) associates a numeric *timestamp* to each item, representing its temporal coordinate, in time units, with respect to time zero. Therefore, we denote the generic item of a sequence having data type T as (t_i, s_i) where $i \in \mathbb{N}$ is the position of the item in the sequence, $t_i \in \mathbb{N}$ is the timestamp and $s_i \in T$ is the item value. We denote an ordered data collection of data type T with the following, where $\overset{(b)}{=}$ holds only for bounded sequences (i.e., lists):

$$\begin{aligned} s &= [(t_0, s_0), (t_1, s_1), (t_2, s_2), \ldots \bullet t_i \in \mathbb{N}, s_i \in T] \\ &= [(t_0, s_0)] +\!\!+ [(t_1, s_1), (t_2, s_2), \ldots] \\ &= (t_0, s_0) :: [(t_1, s_1), (t_2, s_2), \ldots] \\ &\overset{(b)}{=} \left[(t_0, s_0), (t_1, s_1), \ldots, (t_{|s|-1}, s_{|s|-1})\right] \end{aligned} \tag{5.2}$$

The symbol ++ represents the concatenation of sequence $[(t_0, s_0)]$ (head sequence) with the sequence $[(t_1, s_1), (t_2, s_2), \ldots]$ (tail sequence). The symbol :: represents the concatenation of element (t_0, s_0) (head element) with the sequence $[(t_1, s_1), (t_2, s_2), \ldots]$ (tail sequence).

We define the notion of *time-ordered sequences*.

Definition 2. *A sequence* $s = [(t_0, s_0), (t_1, s_1), (t_2, s_2), \ldots]$ *is time-ordered when the following condition is satisfied for any* $i, j \in \mathbb{N}$*:*

$$i \leq j \Rightarrow t_i \leq t_j$$

We denote as $\overrightarrow{s}$ any time-ordered permutation of s. The ability of dealing with non-time-ordered sequences, which is provided by PiCo, is sometimes referred as *out-of-order* data processing [72].

Before proceeding to semantic operators and Pipelines, we define some preliminary notions about the effect of partitioning and windowing over semantic collections.

Partitioned Collections

In Section 5.1.2, we introduced partitioning policies. In semantic terms, a partitioning policy π defines how to group collection elements.

Definition 3. *Given a multi-set* m *of data type* T*, a function* $\pi : T \to K$ *and a key* $k \in K$*, we define the k-selection* $\sigma_k^\pi(m)$ *as follows:*

$$\sigma_k^\pi(m) = \{m_i \bullet x \in m_i \wedge \pi(m_i) = k\} \tag{5.3}$$

Similarly, the k-selection $\sigma_k^\pi(s)$ *of a sequence* s *is the sub-sequence of* s *such that the following holds:*

$$\forall (t_i, s_i) \in s, (t_i, s_i) \in \sigma_k^\pi(s) \iff \pi(s_i) = k \tag{5.4}$$

We define the partitioned collection as the set of all groups generated according to a partitioning policy.

Definition 4. *Given a collection* c *and a partitioning policy* π*, the partitioned collection* c *according to* π*, noted* $c^{(\pi)}$*, is defined as follows:*

$$c^{(\pi)} = \{\sigma_k^\pi(c) \bullet k \in K \wedge |\sigma_k^\pi(c)| > 0\} \tag{5.5}$$

We remark that partitioning has no effect with respect to time-ordering.

Example: The group-by-key decomposition, with π_1 being the left projection,[6] uses a special case of selection where:

- the collection has data type $K \times V$
- $\pi = \pi_1$

Windowed Collections

Before proceeding further, we provide the preliminary notion of *sequence splitting*. A splitting function f defines how to split a sequence into two possibly overlapping sub-sequences, namely the *head* and the *tail*.

Definition 5. *Given a sequence* s *and a splitting function* f*, the splitting of* s *according to* f *is:*

$$f(s) = (h(s), t(s)) \tag{5.6}$$

where $h(s)$ *is a bounded prefix of* s*,* $t(s)$ *is a proper suffix of* s*, and there is a prefix* p *of* $h(s)$ *and a suffix* u *of* $t(s)$ *such that* $s = p + +u$*.*

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

In Section 5.1.2, we introduced windowing policies. In semantic terms, a windowing policy ω identifies a splitting function $f^{(\omega)}$. Considering a split sequence $f_\omega(s)$, the head $h_\omega(s)$ represents the elements falling into the window, whereas the tail $t_\omega(s)$ represents the remainder of the sequence.

We define the windowed sequence as the result of repeated applications of windowing with time-reordering of the heads.

Definition 6. *Given a sequence s and a windowing policy w, the windowed view of s according to w is:*

$$s^{(\omega)} = [\overrightarrow{s_0}, \overrightarrow{s_1}, \ldots, \overrightarrow{s_i}, \ldots] \tag{5.7}$$

where $s_i = h_\omega(\underbrace{t_\omega(t_\omega(\ldots t_\omega}_{i}(s)\ldots)))$

Example: The count-based policy $\omega = (5, 2, \texttt{count})$ extracts the first 5 items from the sequence at hand and discards the first 2 items of the sequence upon sliding, whereas the tumbling policy $\omega = (5, 5, \texttt{count})$ yields non-overlapping contiguous windows spanning 5 items.

5.3.2 Semantic Operators

We define the semantics of each operator in terms of its behavior with respect to token processing by following the structure of Table 5.3. We start from bounded operators and then we show how they can be extended to their unbounded counterparts by considering windowed streams.

Dataflow vertexes with one input edge and one output edge (i.e., unary operators with both input and output degrees equal to 1) take as input a token (i.e., a data collection), apply a transformation, and emit the resulting transformed token. Vertexes with no input edges (i.e., `emit`)/no output edges (i.e., `collect`) execute a routine to produce/consume an output/input token, respectively.

Semantic Core Operators

The bounded `map` operator has the following semantics:

$$\begin{aligned} \texttt{map}\ f\ m &= \{f(m_i) \bullet m_i \in m\} \\ \texttt{map}\ f\ s &= \left[(t_0, f(s_0)), \ldots, (t_{|s|-1}, f(s_{|s|-1}))\right] \end{aligned} \tag{5.8}$$

where m and s are input tokens (multi-set and list, respectively) whereas right-hand side terms are output tokens. In the ordered case, we refer to the above definition as *strict* semantic `map`, since it respects the global time-ordering of the input collection.

The bounded `flatmap` operator has the following semantics:

$$\begin{aligned} \texttt{flatmap}\ f\ m &= \bigcup \{f(m_i) \bullet m_i \in m\} \\ \texttt{flatmap}\ f\ s &= [(t_0, f(s_0)_0), (t_0, f(s_0)_1), \ldots, (t_0, f(s_0)_{n_0})] +\!+ \\ &\quad [(t_1, f(s_1)_0), \ldots, (t_1, f(s_1)_{n_1})] +\!+ \ldots +\!+ \\ &\quad \left[(t_{|s|-1}, f(s_{|s|-1})_0) \ldots, (t_{|s|-1}, f(s_{|s|-1})_{n_{|s|-1}})\right] \end{aligned} \tag{5.9}$$

where $f(s_i)_j$ is the j-th item of the list $f(s_i)$, that is, the output of the kernel function f over the input s_i. Notice that the timestamp of each output item is the same as the respective input item.

The bounded `reduce` operator has the following semantics, where $\oplus$ is both associative and commutative and, in the ordered variant, $t' = \max_{(t_i,s_i)\in s} t_i$:

$$\begin{aligned} \texttt{reduce}\ \oplus\ m &= \{\bigoplus \{m_i \in m\}\} \\ \texttt{reduce}\ \oplus\ s &= \left[(t', (\ldots(s_0 \oplus s_1) \oplus \ldots) \oplus s_{|s|-1})\right] \\ &\overset{(a)}{=} \left[(t', s_0 \oplus s_1 \oplus \ldots \oplus s_{|s|-1})\right] \\ &\overset{(c)}{=} \left[(t', \bigoplus \Pi_2(s))\right] \end{aligned} \tag{5.10}$$

meaning that, in the ordered variant, the timestamp of the resulting value is the same as the input item having the maximum timestamp. Equation $\overset{(a)}{=}$ holds since $\oplus$ is associative and equation $\overset{(c)}{=}$ holds since it is commutative.

The `fold+reduce` operator has a more complex semantics, defined with respect to an *arbitrary* partitioning of the input data. Informally, given a partition P of the input collection, each subset $P_i \in P$ is mapped to a local accumulator a_i, initialized with value z; then:

1. Each subset P_i is folded into its local accumulator a_i, using $\oplus_1$;
2. The local accumulators a_i are combined using $\oplus_2$, producing a reduced value r;

The formal definition—that we omit for the sake of simplicity—is similar to the semantic of `reduce`, with the same distinction between ordered and unordered processing and similar considerations about associativity and commutativity of user functions. We assume, without loss of generality, that the user parameters z and $\oplus_1$ are always defined such that the resulting `fold+reduce` operator is partition-independent, meaning that the result is independent from the choice of the partition P.

Semantic Decomposition

Given a bounded `combine` operator *op* and a selection function $\pi : T \to K$, the partitioning operator `p-`*op* has the following semantics over a generic collection c:

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

For instance, the group-by-key processing is obtained by using the by-key partitioning policy (cf. example below definition 3).

Similarly, given a bounded `combine` operator *op* and a windowing policy ω, the windowing operator `w-`*op* has the following semantics:

$$\texttt{w-}op\ \omega\ s = op\ s_0^{(\omega)} +\!+ \ldots +\!+\ op\ s_{|s^{(\omega)}|-1}^{(\omega)} \tag{5.11}$$

where $s_i^{(\omega)}$ is the i-th list in $s^{(\omega)}$ (cf. Definition 6).

As for the combination of the two partitioning mechanisms, `w-p-`*op*, it has the following semantics:

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

Thus, as mentioned in Section 5.1.2, partitioning first performs the decomposition, and then processes each group on a per-window basis.

Unbounded Operators

We remark that none of the semantic operators defined so far can deal with unbounded collections. As mentioned in Section 5.1.2, we rely on windowing for extending them to the unbounded case.

Given a (bounded) windowing `combine` operator *op*, the semantics of its unbounded variant is a trivial extension of the bounded case:

$$\texttt{w-}op\ \omega\ s = op\ s_0^{(\omega)} +\!+ \ldots +\!+\ c\ s_i^{(\omega)} +\!+ \ldots \tag{5.12}$$

The above incidentally also defines the semantics of unbounded windowing and partitioning `combine` operators.

We rely on the analogous approach to define the semantics of unbounded operators in the `map` family, but in this case the windowing policy is introduced at the semantic rather than syntactic level, since `map` operators do not support decomposition. Moreover, the windowing policy is forced to be batching (cf. Example below Definition 5). We illustrate this concept on `map` operators, but the same holds for `flatmap` ones. Given a bounded `map` operator, the semantics of its unbounded extension is as follows, where ω is a tumbling windowing policy:

$$[\![\texttt{map}\ f\ s]\!]_\omega = \texttt{map}\ f\ s_0^{(\omega)} +\!+ \ldots +\!+\ \texttt{map}\ f\ s_i^{(\omega)} +\!+ \ldots \tag{5.13}$$

We refer to the above definition as *weak* semantic `map` (cf. strict semantic `map` in Equation 5.8), since the time-ordering of the input collection is partially dropped. In the following chapters, we provide a PiCo implementation based on weak semantic operators for both bounded and unbounded processing.

Semantic Sources and Sinks

Finally, `emit`/`collect` operators do not have a functional semantics, since they produce/consume collections by interacting with the system state (e.g., read/write from/to a text file, read/write from/to a network socket). From the semantic perspective, we consider each `emit`/`collect` operator as a Dataflow node able to produce/consume as output/input a collection of a given type, as shown in Table 5.3. Moreover, `emit` operators of ordered type have the responsibility of tagging each emitted item with a timestamp.

5.3.3 Semantic Pipelines

The semantics of a Pipeline maps it to a semantic Dataflow graph. We define such mapping by induction on the Pipeline grammar defined in Section 5.1. The following definitions are basically a formalization of the pictorial representation in Figure 5.1.

We also define the notion of *input*, resp. *output*, vertex of a Dataflow graph G, denoted as $v_I(G)$ and $v_O(G)$, respectively. Conceptually, an input node represents a Pipeline source, whereas an output node represents a Pipeline sink.

The following formalization provides the semantics of any PiCo program.

- (NEW *op*) is mapped to the graph $G = (\{op\}, \emptyset)$; moreover, one of the following three cases hold:
  - *op* is an `emit` operator, then $v_O(G) = op$, while $v_I(G)$ is undefined
  - *op* is a `collect` operator, then $v_I(G) = op$, while $v_O(G)$ is undefined

  - *op* is an unary operator with both input and output degree equal to 1, then $v_I(G) = v_O(G) = op$

- (TO $p\ p_1\ \ldots\ p_n$) is mapped to the graph $G = (V, E)$ with:

$$\begin{aligned} V = &\ V(G_p) \cup V(G_{p_1}) \cup \ldots \cup V(G_{p_n}) \cup \{\mu\} \\ E = &\ E(G_p) \cup \bigcup_{i=1}^{n} E(G_{p_i}) \cup \bigcup_{i=1}^{n} \{(v_O(G_p), v_I(G_{p_i}))\} \cup \\ &\ \bigcup_{i=1}^{|G'|} \{(v_O(G'_i), \mu)\} \end{aligned}$$

  where μ is a non-determinate merging node as defined in [91] and $G' = \{G_{p_i} \bullet d_O(G_{p_i}) = 1\}$; moreover, $v_I(G) = v_I(G_p)$ if $d_I(G_p) = 1$ and undefined otherwise, while $v_O(G) = \mu$ if $|G'| > 0$ and undefined otherwise.

- (PAIR $p\ p'\ op$) is mapped to the graph $G = (V, E)$ with:

$$\begin{aligned} V &= V(G_p) \cup V(G_{p'}) \cup \{o\}\, p \\ E &= E(G_p) \cup E(G_{p'}) \cup \{(v_O(G_p), op)\,, (v_O(G_{p'}), op)\} \end{aligned}$$

  moreover, $v_O(G) = op$, while one of the following cases holds:
  - $v_I(G) = v_I(G_p)$ if the input degree of p is 1
  - $v_I(G) = v_I(G_{p'})$ if the input degree of p' is 1
  - $v_I(G)$ is undefined if both p and p' have output degree equal to 0

- (MERGE $p\ p'$) is mapped to the graph $G = (V, E)$ with:

$$\begin{aligned} V &= V(G_p) \cup V(G_{p'}) \cup \{\mu\} \\ E &= E(G_p) \cup E(G_{p'}) \cup \{(v_O(G_p), \mu)\,, (v_O(G_{p'}), \mu)\} \end{aligned}$$

  where μ is a non-determinate merging node; moreover, $v_O(G) = \mu$, while one of the following cases holds:
  - $v_I(G) = v_I(G_p)$ if the input degree of p is 1
  - $v_I(G) = v_I(G_{p'})$ if the input degree of p' is 1
  - $v_I(G)$ is undefined if both p and p' have output degree equal to 0

Running Example: Semantics of word-count

The tokens (i.e., data collections) flowing through the semantic Dataflow graph resulting from the WORD-COUNT Pipeline are bags of strings (e.g., lines produced by `file-read` and consumed by `tokenize`) or bags of string-$\mathbb{N}$ pairs (e.g., counts produced by `tokenize` and consumed by `keyed-sum`). In this example, as usual, string-$\mathbb{N}$ pairs are treated as key-value pairs, where keys are strings (i.e., words) and values are numbers (i.e., counts).

By applying the semantic of `flatmap`, `reduce` and `p-(reduce` $\oplus$) to Algorithm 1, the result obtained is that the token being emitted by the `combine` operator is a bag of pairs (w, n_w) for each word w in the input token of the `flatmap` operator.

The Dataflow graph resulting from the semantic interpretation of the WORD-COUNT Pipeline defined in Section 5.1 is $G = (V, E)$, where:

$$\begin{aligned} V &= \{\texttt{tokenize}, \texttt{keyed-sum}\} \\ E &= \{(\texttt{tokenize}, \texttt{keyed-sum})\} \end{aligned}$$

Finally, the FILE-WORD-COUNT Pipeline results in the graph $G = (V, E)$ where:

$$\begin{aligned} V &= \{\texttt{file-read}, \texttt{tokenize}, \texttt{keyed-sum}, \texttt{file-write}\} \\ E &= \{(\texttt{file-read}, \texttt{tokenize})\,, \\ &\quad (\texttt{tokenize}, \texttt{keyed-sum})\,, \\ &\quad (\texttt{keyed-sum}, \texttt{file-write})\} \end{aligned}$$

5.4 Programming Model Expressiveness

In this section, we provide a set of use cases adapted from examples in Flink's user guide [68]. Besides they are very simple examples, they exploit grouping, partitioning, windowing and Pipelines merging. We aim to show the expressiveness of our model without using any concrete API, to demonstrate that the model is independent from its implementation.

5.4.1 Use Cases: Stock Market

The first use case is about analyzing stock market data streams. In this use case, we:

1. read and merge two stock market data streams from two sockets (algorithm 2)
2. compute statistics on this market data stream, like rolling aggregations per stock (algorithm 3)
3. emit price warning alerts when the prices change (algorithm 4)
4. compute correlations between the market data streams and a Twitter stream with stock mentions (algorithm 5)

Algorithm 2 The READ-PRICE Pipeline

READ-PRICES = NEW `from-socket` s_1 + NEW `from-socket` s_2

Read from multiple sources Algorithm 2 shows the STOCK-READ Pipeline, which reads and merges two stock market data streams from sockets s_1 and s_2. Assuming StockName and Price are types representing stock names and prices, respectively, then the type of each `emit` operator is the following (since `emit` operators are polymorphic with respect to data type):

$$\emptyset \to (\text{StockName} \times \text{Price})_{\{\text{stream}\}}$$

Therefore it is also the type of READ-PRICES since it is a MERGE of two `emit` operators of such type.

Algorithm 3 The STOCK-STATS Pipeline

`min` = `reduce` $(\lambda xy.\min(x, y))$
`max` = `reduce` $(\lambda xy.\max(x, y))$
`sum-count` = `fold+reduce` $(\lambda ax.((\pi_1(a)) + 1, (\pi_2(a)) + x))\ (0, 0)$
$(\lambda a_1 a_2.(\pi_1(s_1) + \pi_1(a_2), \pi_2(a_1) + \pi_2(a_2)))$
`normalize` = `map` $(\lambda x.\pi_2(x)/\pi_1(x))$
$\omega = (10, 5,$ `count`$)$

STOCK-STATS = TO READ-PRICES
NEW `w-p-(min)` $\pi_1\ \omega$
NEW `w-p-(max)` $\pi_1\ \omega$
(NEW `w-p-(sum-count)` $\pi_1\ \omega$ | NEW `normalize`)

Statistics on market data stream Algorithm 3 shows the STOCK-STATS Pipeline, that computes three different statistics—minimum, maximum and mean—for each stock name, over the prices coming from the READ-PRICES Pipeline. These

statistics are windowing based, since the data processed belongs to a stream possibly unbound. The specified window policy $\omega = (10, 5, \texttt{count})$ creates windows of 10 elements with sliding factor 5.

The type of STOCK-STATS is $\emptyset \rightarrow (\text{StockName} \times \text{Price})_{\{\text{stream}\}}$, the same as READ-PRICES.

Algorithm 4 The PRICE-WARNINGS Pipeline

$\texttt{collect} = \texttt{fold+reduce}\ \ (\lambda s x.s \cup \{x\})\ \emptyset$
$\qquad\qquad\qquad\qquad\qquad\quad (\lambda s_1 s_2.s_1 \cup s_2)$
$\texttt{fluctuation} = \texttt{map}\ (\lambda s.\text{set-fluctuation}(s))$
$\texttt{high-pass} = \texttt{flatmap}\ (\lambda \delta.\text{if } \delta \geq 0.05 \text{ then yield } \delta)$
$\omega = (10, 5, \texttt{count})$

PRICE-WARNINGS = READ-PRICES |
NEW `w-p-(collect)` $\pi_1\ \omega$ | NEW `fluctuation`
NEW `high-pass`

Generate price fluctuation warnings Algorithm 4 shows the Pipeline PRICE-WARNINGS, that generates a warning each time the stock market data within a window exhibits high price fluctuation for a certain stock name—`yield` is a host-language method that produces an element.

In the example, the `fold+reduce` operator `fluctuation` just builds the sets, one per window, of all items falling within the window, whereas the downstream `map` computes the fluctuation over each set. This is a generic pattern that allows to combine collection items by re-using available user functions defined over collective data structures.

The type of PRICE-WARNINGS is again $\emptyset \rightarrow (\text{StockName} \times \text{Price})_{\{\text{stream}\}}$.

Algorithm 5 The CORRELATE-STOCKS-TWEETS Pipeline

READ-TWEETS = NEW `from-twitter` | NEW `tokenize-tweets`
$\omega = (10, 10, \texttt{count})$

CORRELATE-STOCKS-TWEETS = PAIR PRICE-WARNINGS READ-TWEETS
`w-p-(correlate)` $\pi_1\ \omega$

Correlate warnings with tweets Algorithm 5 shows CORRELATE-STOCKS-TWEETS, a Pipeline that generates a correlation between warning generated by PRICE-WARNINGS and tweets coming from a Twitter feed. The READ-TWEETS Pipeline generates a stream of (StockName × String) items, representing tweets each mentioning a stock name. Stocks and tweets are paired according to a join-by-key policy (cf. definition 3), where the key is the stock name.

In the example, `correlate` is a `join-fold+reduce` operator that computes the correlation between two joined collections. As we mentioned in Section 5.1.2, we rely on windowing to apply the (bounded) `join-fold+reduce` operator to unbounded streams. In the example, we use a simple tumbling policy $\omega = (10, 10, \texttt{count})$ in order to correlate items from the two collections in a 10-by-10 fashion.

5.5 Summary

In this chapter we proposed a new programming model based on Pipelines and operators, which are the building blocks of PiCo programs, first defining the syntax of programs, then providing a formalization of the type system and semantics.

The contribution of PiCo with respect to the state-of-the-art is also in the definition and formalization of a programming model that is independent from the effective API and runtime implementation. In the state-of-the-art tools for analytics, this aspect is typically not considered and the user is left in some cases to its own interpretation of the documentation. This happens particularly when the implementation of operators in state-of-the-art tools is conditioned in part or totally by the runtime implementation itself.

Chapter 6

PiCo Parallel Execution Graph

In this chapter, we show how a PiCo program is compiled into a graph of parallel processing nodes. The compilation step takes as input the direct acyclic dataflow (DAG) resulting from a PiCo program (the Semantic DAG) and transforms it, with a set of rules, into a graph that we call the Parallel Execution (PE) Graph, representing a possible parallelization of the Semantic DAG.

The resulting graph is a classical macro-Dataflow network [91], in which tokens represent portions of data collections and nodes are persistent processing nodes mapping input to output tokens, according to a pure functional behavior.

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 or 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 the compiled graphs, similarly to what is done by an optimizing compiler over intermediate representations.

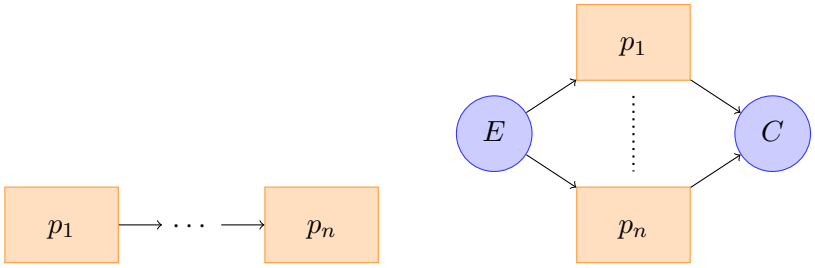
In this chapter, we define the Parallel Execution layer of the Dataflow stack as it has been defined in [120]. We remark, as stressed in the aforementioned work, that the parallel execution model discussed in this chapter is abstract with respect to any actual implementation. For instance, it may be implemented in shared memory or through a distributed runtime. Moreover, a compiled (and optimized) Dataflow graph may be directly mapped to an actual network of computing units (e.g., communicating threads or processes) or executed by a macro-Dataflow interpreter.

This chapter proceeds as follows. We first define the target language and show the compilation of each single operator with respect to different compilation environments. Then we show compilations of Pipelines constructors NEW, MERGE and TO. Finally, we show the optimization phase in which operators, while creating Pipelines, are simplified into more compact and efficient target objects.

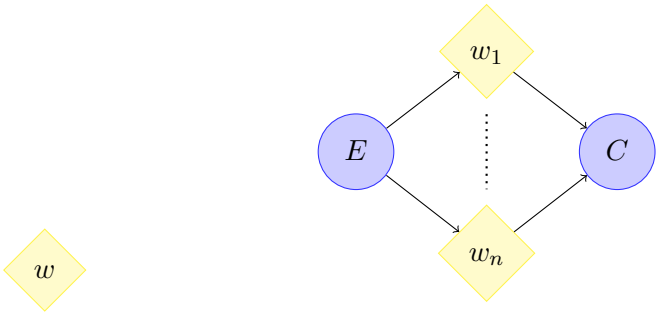
6.1 Compilation

In this section we describe how PiCo operators and Pipelines are compiled into parallel execution (PE) graphs [120]. The target language is composed of Dataflow graphs and it is inspired by the FastFlow library architecture.

The schema is the following: given a PiCo program, it is mapped to an *intermediate representation* (IR) graph that expresses the available parallelism; then an IR graph



(A) PE pipe (B) PE farm



(C) PE processing node (D) PE operator-farm

FIGURE 6.1: Grammar of PE graphs

optimization is performed, resulting into the actual parallel execution (PE) graph. Both the IR and the PE graphs are expressed in terms of the target language. Top-level terms (cf. definition 1) are compiled into *executable* PE graphs.

We only consider PiCo terms in the respective normal form, that is, the form induced by applying all the structural equivalences reported in table 5.1. For the sake of simplicity, we limit the discussion by considering the PiCo subset not including binary operators, since they can be treated in an analogous way as unary operators.

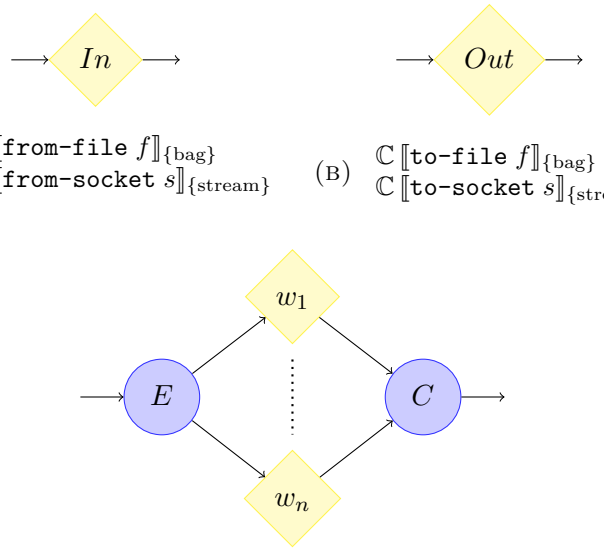
We represent the compilation of a PiCo term p as the following, where ρ is a compilation environment:

$$\mathbb{C}\,[\![p]\!]_\rho$$

We include such environments at this level to reflect the fact that operators and Pipelines with the same syntax can be mapped to different PE graphs. We consider simple compilation environments composed only by a set of structure types that allow compilations depending from the collection structure types processed—PiCo terms are polymorphic (cf. Section 5.2). Thus, a Pipeline p can be compiled into two different PE graphs $\mathbb{C}\,[\![p]\!]_{\delta_1}$ and $\mathbb{C}\,[\![p]\!]_{\delta_2}$, where $\delta_1 \subseteq \Sigma$ and $\delta_2 \subseteq \Sigma$. The selection of the actual compilation is unique when it comes to (sub-terms of) top-level Pipelines, since the environment is propagated top-down by the compilation process (i.e., through inherited attributes). Moreover, we omit the compilation environment if it can be easily inferred from the context.

Figure 6.1 graphically represents the target language's grammar. A PE graph is one of the following symbols:

- PE operators (Figures 6.1c and 6.1d), representing PiCo operators and singleton Pipelines (i.e., NEW)
- PE farms (Figure 6.1b), representing branching TO and MERGE Pipelines
- PE pipes (Figure 6.1a), representing linear TO Pipelines



(A) $\mathbb{C}\,[\![\texttt{from-file}\ f]\!]_{\{\text{bag}\}}$
$\mathbb{C}\,[\![\texttt{from-socket}\ s]\!]_{\{\text{stream}\}}$

(B) $\mathbb{C}\,[\![\texttt{to-file}\ f]\!]_{\{\text{bag}\}}$
$\mathbb{C}\,[\![\texttt{to-socket}\ s]\!]_{\{\text{stream}\}}$

(C) $\mathbb{C}\,[\![\texttt{map}\ f]\!]_{\Sigma}$
$\mathbb{C}\,[\![\texttt{flatmap}\ f]\!]_{\Sigma}$
$\mathbb{C}\,[\![\texttt{reduce}\ \oplus]\!]_{\Sigma_b}$ and decomposing variants
$\mathbb{C}\,[\![\texttt{fold+reduce}\ \oplus_1\ z\ \oplus_2]\!]_{\Sigma_b}$ and decomposing variants

FIGURE 6.2: Operators compilation in the target language.

A PE operator can be either a processing node (Figure 6.1c) or a farm-composition of n processing nodes and additional mediator nodes (Figure 6.1d). Processing nodes are *atomic* processing units, whereas mediator nodes are generic entities responsible for inter-node communication. For instance, a mediator can be a tree composition of processing units.

6.1.1 Operators

Each operator defined in Section 5.1.2 is compiled into a single PE operator. The compilation of any PE node has at least one input and one output port, since it exchanges synchronization tokens in addition to data.

Figure 6.2 schematizes the compilation of PiCo operators into the target language. All the PE operators have a single entry point and a single exit point in order to be connected to other operators.

The compilation results are composed by the following entities:

- *Processing nodes* are executing user code. In figure 6.2c, they are the workers in the PE graphs for `map`, `flatmap` and `reduce`; moreover, *In* and *Out* nodes in Figures 6.2a and 6.2b are processing nodes that execute some host-language routine (possibly taking some user code as parameter) to process input and output collections.
- *Emitters* are mediator nodes, identified by E, dispatching collection elements to processing nodes, possibly coming from upstream nodes. The dispatching may have different policies with respect to windowing and/or partitioning.
- *Collectors* are mediator nodes, identified by C, collecting items from Workers and possibly forwarding results to the downstream nodes. Collector nodes may have to reorder items in case of ordered collections before forwarding.

PE nodes have varied flavors since they have varied roles. For instance, a PE node can be stateless or stateful depending on the role it plays. We provide some exemplified specialization of various PE nodes, with respect to the respective compilation subjects.

We consider two different compilation approaches: fine-grained and batching. The former approach is the most naive, in which data collections are decomposed down to their atomic components with respect to a specific operator; in the latter approach, collections are decomposed into non-atomic portions (sometimes referred as batches or micro-batches) to enable more efficient implementations.

Fine-grained PE graphs

In the naive fine-grained compilation approach, the tokens flowing through the PE graph represent the smallest portions of the data collection at hand. For instance, a multi-set is decomposed down to its single items—a token for each item—when processed by a PE operator resulting from the compilation of a `map` operator; in the case of windowing operators, a token represents an entire window $s_i^{(\omega)}$ (cf. Equation 5.11).

- $\mathbb{C}[\![\texttt{map}\ f]\!]_\Sigma$: the Emitter distributes tokens to the Workers that execute the function f over each input token (i.e., a single item from the input collection). Workers and Emitter are stateless. In the simplest case of unordered processing, also the Collector is stateless and it simply gathers and forwards the tokens coming from the workers. In the case of ordered processing, the Collector is deputed to item reordering and it may rely on buffering—in which case it is stateful. As we discuss later, this approach poses relevant challenges in the case of stream processing.
- $\mathbb{C}[\![\texttt{reduce}\ \oplus]\!]_{\Sigma_b}$: the Emitter is analogous to the previous case. The Workers calculate partial results and send them to the Collector that computes the final result. All computations apply the $\oplus$ function on the collected data. Therefore, Workers are stateful since they need to store locally the collected data. Notice that, from the semantic reduction in Equation 5.10, time-ordering is irrelevant in this case.
- $\mathbb{C}[\![\texttt{p-(reduce}\ \oplus)\ \pi]\!]_{\Sigma_b}$: the partitioning policy π is encoded into the Emitter, that distributes items to Workers by mapping a given key k to a specific Worker. This is not required in principle, but it simplifies the implementation since only the Emitter needs to be aware of the partitioning policy; moreover, since all the processing for a given key k is localized in the same Worker, the Collector is a simple forwarding node and kernels relying on partitioned state are supported without any need for coordination control.[1] Workers apply the $\oplus$ function to compute the reduce of each group and emit the results to the Collector, that simply gathers and forwards the results. Also in this case, time-ordering can be safely ignored.
- $\mathbb{C}[\![\texttt{w-(reduce}\ \oplus)\ \omega]\!]_{\Sigma_o}$: as discussed in [60], windowing stream processing can be regarded as a form of stateful processing, since each computation depends on some recent stream history. In the aforementioned work, several patterns for sliding-windows stream processing are proposed in terms of Dataflow farms. In the Window Farming (WF) pattern, each window can be processed independently by any worker. We consider the PE operator at hand as an instance of the WF pattern, since each window can be reduced independently. The main issue with this scenario is that, since windows may overlap, each stream item may be needed by more than one worker. In the proposed approach, windows are statically mapped to Workers, therefore the

[1] Although the current PiCo formulation does not support this feature.

(stateless) Emitter simply computes the set of windows that will contain a certain item and sends each item to all the workers that need it. Each Worker maintains an internal state to produce the windows and applies the (sequential) reduction over each window once it is complete. The Collector relies on buffering to reconstruct the time-ordering between windows. We remark only parallelism among different windows is exploited, whereas each Worker processes each input window sequentially.

- $\mathbb{C}[\![\texttt{w-p-(reduce}\ \oplus)\ \pi\ \omega]\!]_{\Sigma_o}$: in the Key Partitioning (KP) pattern [60], the stream is logically partitioned into disjoint sub-streams and the time-ordering must be respected only within each sub-stream. Thus it is natural to consider the PE operator at hand as an instance of the KP pattern, where sub-streams are constructed according to the partitioning policy π. The resulting PE operator-farm is analogous to the previous case, except from the Collector that in this case is simpler since no time-reordering is required.

As we show in Figure 6.2, all the PE graphs resulting from the compilation of data-parallel operators (cf. Section 5.1.2) are structurally identical.

Batching PE graphs

The first problem with the fine-grained compilation approach is related to the computational granularity, which is a well-known issue in the domain of parallel processing performance. Setting the Workers to process data at the finest granularity induces a high communication traffic between the computational units of any underlying implementation. Therefore, the fine-grained approach increases the ratio of communication over computation, which is one of the main factors of inefficiency when it comes to parallel processing. In particular, the discussed issue would have a relevant impact in any distributed implementation, in which the communication between processing nodes is expensive.

The second problem with the fine-grained approach is more subtle and is related to the semantics of PiCo operators as defined in Section 5.3.2. Let us consider the compilation of a `map` operator over lists (i.e., bounded sequences) with strict semantics (cf. Equation 5.8). In the PE operator resulting from its compilation, the Collector has to restore the time-ordering among all the collection items, thus in the worst case it has to buffer all the results from the Workers before starting to emit any token. Moreover, it is not possible to implement an unbounded strict semantic `map`. For instance, in the fine-grained compilation setting, this would require infinite buffering by the Collector. Conversely, if we consider the weak semantics (cf. Equation 5.13), the relative service times of the Workers for each item determine the order in which the Collector receives the items to be reordered. Therefore, it is impossible to implement the weak semantic map without passing all the information about the original time-ordering from the Emitter to the Collector.

The aspects discussed above make an eventual implementation of the fine-grained approach cumbersome and inherently inefficient. To overcome such difficulties, we propose to use instead a batching compilation approach. The idea is simple: the stream is sliced according to a tumbling windowing policy and the processing is carried on a per-slice setting. Therefore all the data is processed on a per-window basis, such that the tokens flowing through PE graphs represent portions of data collections rather than single items.

We already introduced per-window processing in the fine-grained compilation of windowing operators `w-(reduce` $\oplus$) and `w-p-(reduce` $\oplus$), which we retain in the batching approach.

The batching compilation of a `reduce` operator is a simplified version of the compilation of a `w-reduce` operator (i.e., an instance WF pattern). The simplification

comes from the windowing policy is a tumbling one (cf. Example above Definition 5), thus each stream item falls into exactly one window. Moreover there is no need for time-reordering by the Collector due to the semantics of the `reduce` operator. The Workers perform the reduction at two levels: over the elements of an input window (i.e., an input token) and over such reduced values—one per window. Partial results are sent to the Collector that performs the final reduction.

In the same line, the batching compilation of a `p-reduce` operator is analogous to the compilation of a `w-p-reduce` operator (i.e., an instance of the KP pattern).

The batching compilation of a `map` operator is based on the weak semantic `map`. It follows the same schema as the batching `reduce`, but the Emitter keeps an internal buffer to build the time-ordered permutations $s_i^{(\omega)}$ (cf. Equation 5.11) prior to sending the items to the Workers. This enforces the time-ordering within each window, while the time-ordering between different windows is guaranteed by the reordering Collector.

Finally, the batching compilation of a `flatmap` operator is analogous to the `map` case, where tokens emitted by each Worker includes all the items produced while processing the respective input window, in order to respect the weak semantics in Equation 5.13.

Compilation environments

We introduced compilation environments ρ in order to allow parametric compilation of executable PiCo terms into executable PE graphs, depending for instance on the structure type accepted by the subject term. As a use case for compilation environments, we combine parametric batching compilation with the distinction between so-called *batch* and *stream* processing.

We define the compilation of a `map` operator over unbounded streams (i.e., $\rho = \{\text{stream}\}$) to result into a PE farm, in which the Emitter iteratively waits for an incoming data token—namely, a window, since we are considering batching compilation—and dispatches it to the proper farm Worker. This schema is commonly referred as *stream processing*. The idea underlying stream processing is that processing nodes are somehow reactive, in order to process incoming data and synchronization tokens and minimize latency.

Conversely, the compilation of a `map` operator over bounded collections (e.g., $\rho = \{\text{bag}\}$) could result into a farm in which the Emitter waits for a collective data token (i.e., a whole data bag), distributes bag portions to the farm Workers (i.e., scattering) and then suspends itself to avoid consuming computational resources; it may be eventually waken up again to distribute another bag. Both the non-free waking mechanism and the data buffering induced by such batch processing schema introduce some delay, but in the meanwhile the emitter avoids consuming computational resources.

We provide more details about stream processing in Section 6.3.

6.1.2 Pipelines

In this section we show the compilation of PiCo Pipelines into PE graphs. The NEW constructor creates a new Pipeline starting from a single unary operator, thus its compilation coincides with operators compilations previously defined in Section 6.1.1, so we will only consider MERGE and TO constructors.

Merging Pipelines

The MERGE operator unifies n Pipelines producing a single output collection that is the union of the inputs. This operator is both associative and commutative as reported in Table 5.1.

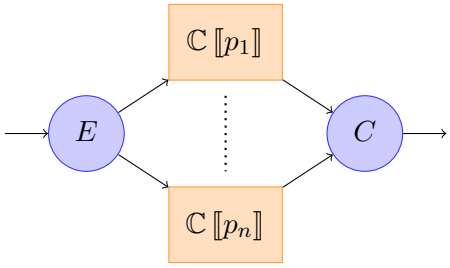


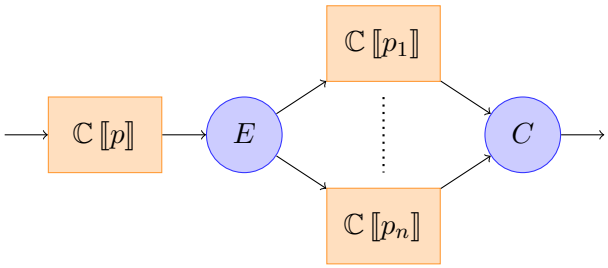
FIGURE 6.3: Compilation of a MERGE Pipeline

Figure 6.3 shows the compilation of $\mathbb{C}[\![p_1 + \ldots + p_n]\!]$, a Pipeline that merges n Pipelines. In this case, the Emitter is simply managing synchronization messages and it is used as a connection point if the resulting pipe is to be composed with another one. The Collector is performing the actual merging, reading input items according to a *from-any* policy and producing a single output. Namely, the Collector node is a classical non-determinate merge, as defined in [91].

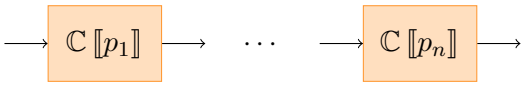
Notice that it never happens for a MERGE-farm to be nested into another MERGE-farm since this is precluded by the associativity of MERGE.

Connecting Pipelines

The TO constructor connects a Pipeline to one or more different Pipelines by broadcasting its output values. This operator is both associative and commutative as reported in Table 5.1.



(A) $\mathbb{C}[\![\text{TO } p\ p_1\ \ldots\ p_n]\!]$



(B) $\mathbb{C}[\![p_1 \mid \ldots \mid p_n]\!] = \mathbb{C}[\![p_1]\!] \circ \ldots \circ \mathbb{C}[\![p_n]\!]$

FIGURE 6.4: Compilation of TO Pipelines

Figure 6.4 shows the compilation of a branching one-to-n Pipeline (Fig. 6.4a) and a sequence of n linear one-to-one Pipelines (Fig. 6.4b). In the case shown in Fig. 6.4a,

the Emitter node is broadcasting input to all Pipelines. Results produced by P_1 and P_2 in Fig. 6.4a are merged by the Collector.

Notice that it never happens for a linear pipe to be nested into another linear pipe since this is precluded by the associativity of TO.

6.2 Compilation optimizations

We now provide a set of optimizations that demonstrates how compositions of operators can be reduced in a more compact and efficient form by removing redundancies and centralization points. We refer to an optimization from a PE graph g_1 to another PE graph g_2 with the following notation, meaning that g_1 can be rewritten into the (optimized) PE graph g_2:

$$g_1 \Rightarrow g_2$$

In the following, we use the standard notation $\Rightarrow^*$ to indicate the application of a number of optimizations.

We remark all the proposed optimizations do not break the correctness of the compilation with respect to the semantics of the subject program. Although we do not provide formal proofs, it can be shown that all the proposed rewriting corresponds to a semantic equivalence.

6.2.1 Composition and Shuffle

We identify two kinds of PE operator compositions, that is, those that generate a *shuffle* and those that do not. A shuffle is a phase of the running application in which data need to be repartitioned among processing nodes following some criteria (e.g. key-based shuffle induced by key-based partitioning). This schema is generally implemented by letting nodes establish a point-to-point communication among each other to exchange data needed to proceed with the computation. In general, it is possible to say that PE graphs that generate a shuffle are those that include a data partitioning or re-partitioning, such as `p-combine` and `w-p-combine`.

6.2.2 Common patterns

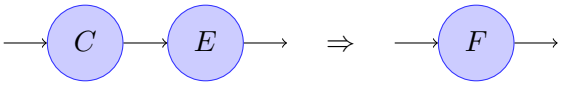


FIGURE 6.5: Forwarding Simplification

The first pattern of reduction we propose is the *Forwarding Simplification*, in which two consecutive nodes that simply forward tokens are fused to a single node, as shown in Figure 6.5. For instance, Figure 6.6a shows the starting configuration for a forwarding simplification, in which C_1 and E_2 are collapsed into F_1, acting as a forwarder node (Fig. 6.6b).

The *Worker-to-Worker* reduction removes the intermediate Emitter or Collector between two set of Workers of two consecutive farms. Given two pools of n workers u and v, the *Worker-to-Worker* optimization directly connects Workers with the same index (u_i and v_i) into n independent 2-stage Pipelines, thus creating a farm of pipes. This optimization can be applied if and only if the node in between

the two pools is only forwarding data or synchronization tokens. For instance, a *Worker-to-Worker* optimization is applied in the optimization of Figure 6.6b into Figure 6.6c.

It is also possible to apply a further optimization to the *Worker-to-Worker* scenario, that we call *Workers Fusion*. It consists in fusing two connected workers (as the result of a *Worker-to-Worker* optimization) into a single node, thus eliminating any intermediate communication. However for simplicity we do not show the *Workers Fusion* in the following sections even where it is applicable.

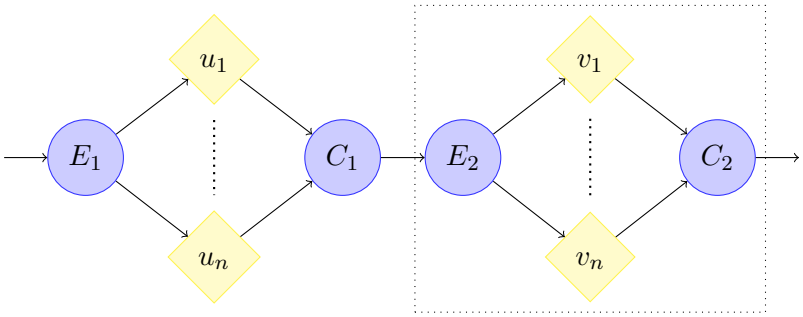
The *All-to-All* optimization is valid when a shuffle is required. The starting configuration of nodes is the same as for the *Worker-to-Worker* optimization. The centralization point is defined by an emitting node partitioning data following a π policy. This centralization can be dropped by connecting all worker nodes with an all-to-all pattern and moving the partitioning logic to each upstream worker u_i. For instance, this optimization is applied in Figure 6.8. When optimizing Figure 6.8b into Figure 6.8c, the partitioning policy is moved to each `map` node, that are deputed to send data to the correct destination Worker.

If the Emitter node is also preparing data for windowing (as in Figure 6.10), this role has to be moved to the downstream workers. Since each downstream worker receives data from any upstream peer, some protocol is needed in order to guarantee the time-ordering is preserved within each partition.

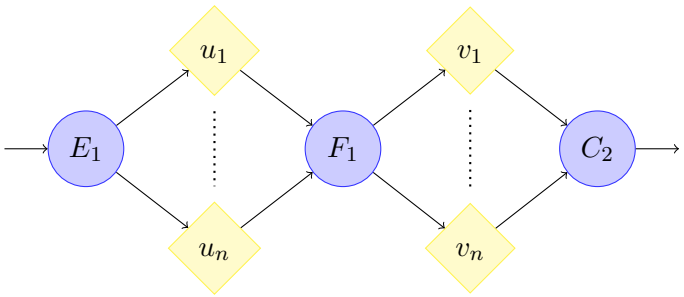
6.2.3 Operators compositions

Following, we show some applications of the optimization patterns defined above.

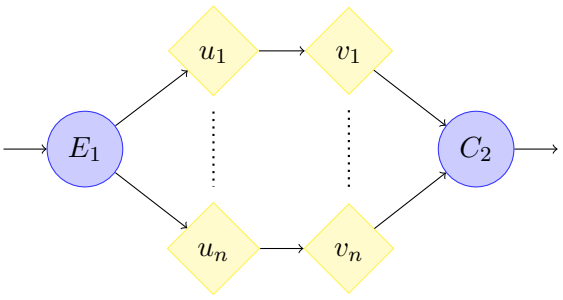
Composition of `map` and `flatmap`



(A) $\mathbb{C}[\![\texttt{map}\ f]\!] \circ \mathbb{C}[\![\texttt{map}\ g]\!]$
$\mathbb{C}[\![\texttt{flatmap}\ f]\!] \circ \mathbb{C}[\![\texttt{flatmap}\ g]\!]$



(B) Result of *Forwarding Simplification* optimization

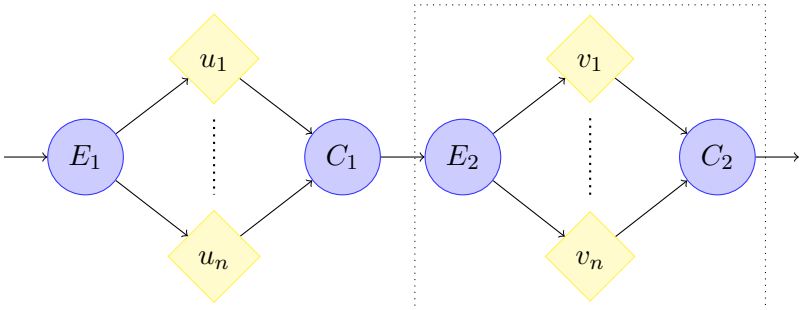


(C) Final network resulting from the *Worker-to-Worker* optimization.

FIGURE 6.6: Compilation of `map`-to-`map` and `flatmap`-to-`flatmap` composition.

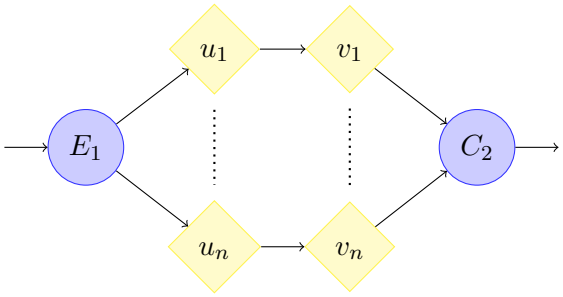
In Figure 6.6 we describe the compilation of a composition of two consecutive `map` or `flatmap` operators—compiling the TO composition of two Pipelines having as `map` and `flatmap` operators that produces the same outcome. In this example, both *Worker-to-Worker* and *Forwarding Simplification* optimizations are applied.

Composition of `map` and `reduce`



(A) $\mathbb{C}[\![\texttt{map}\ f]\!] \circ \mathbb{C}[\![\texttt{reduce}\ \oplus]\!]$

$\Downarrow_*$

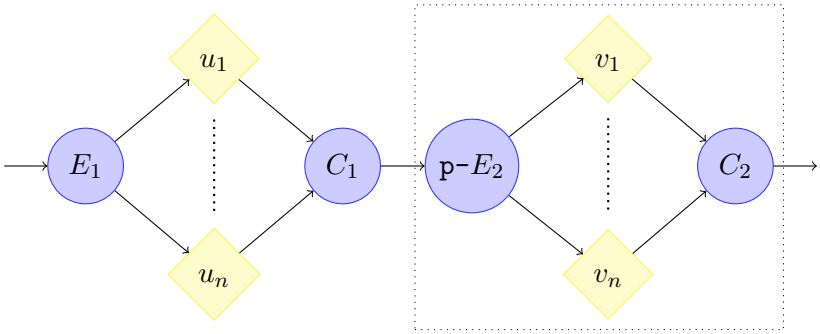


(B) Final network resulting from *Worker-to-Worker* and *Forwarding Simplification*.

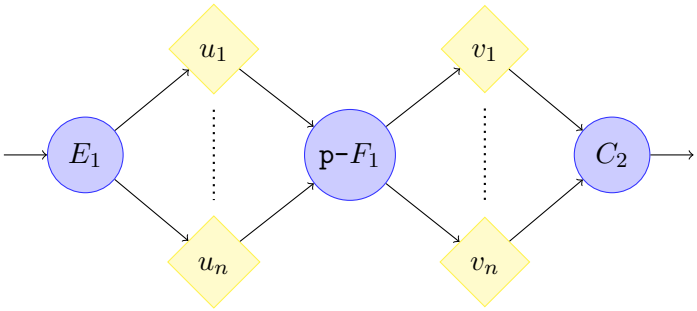
FIGURE 6.7: Compilation of `map-reduce` composition.

The optimization of a `map-reduce` composition, shown in Figure 6.7, is analogous to the `map-map` case. This is possible since the flat `reduce` (i.e. neither windowing nor partitioning) poses no requirement on ordering of data items.

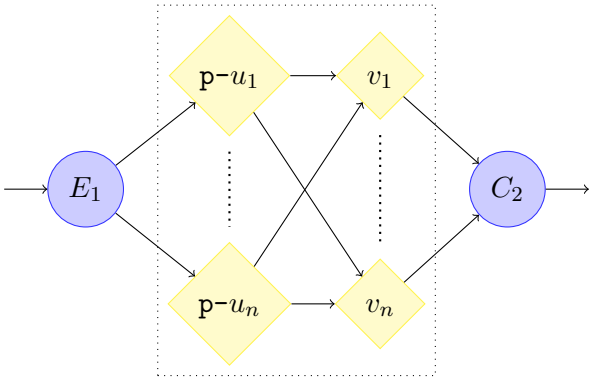
Composition of `map` and `p-reduce`



(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 6.8: Compilation of `map`-to-`p-reduce` composition.

Figure 6.8 shows the compilation of a `map`-to-`p-reduce` composition. This case introduces the concept of shuffle: between the `map` and `p-reduce` operators, the data is said to be shuffled (sometimes referred as parallel-sorted), meaning that data is moved from the `map` workers (i.e., the producers) to the `reduce` workers (i.e., the consumers) in which data will be reduced by following a partitioning criteria. By shuffling data, it is possible to assign, to each worker, data belonging to a given partition, and the `reduce` operator produces a single value for each partition. In general, data shuffling produces an *all-to-all* communication pattern

among `map` and `reduce` workers. The all-to-all shuffle is highlighted by the dotted box in Fig. 6.8c. As an optimization, it is possible to move part of the reducing phase into the `map` workers, so that each `reduce` worker computes the final result for each key by combining partial results coming from `map` workers.

Composition of `map` and `w-reduce`

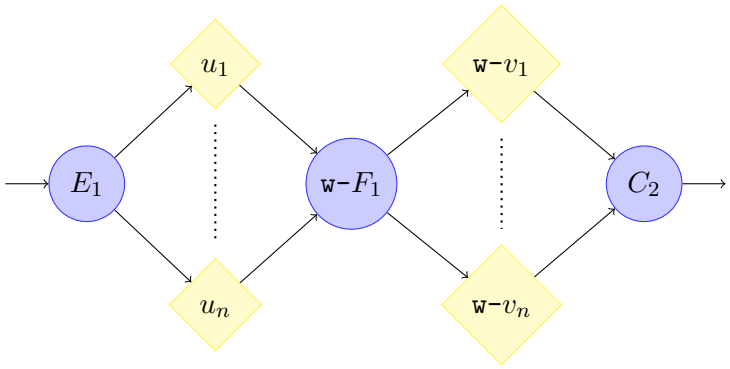
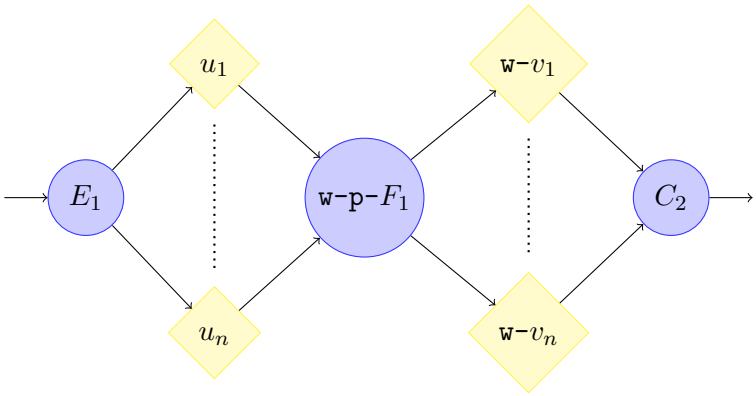


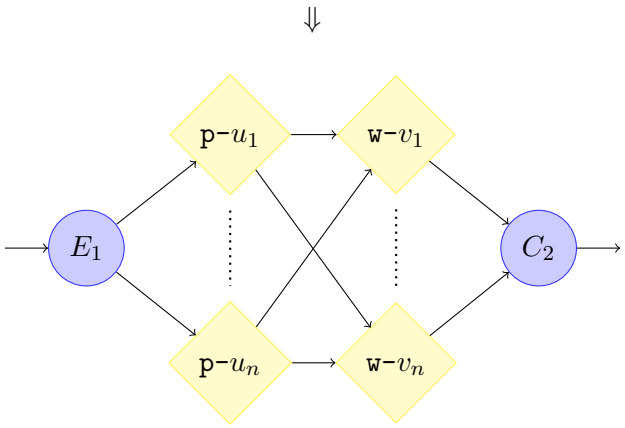
FIGURE 6.9: Compilation of `map`-to-`w-reduce` composition. Centralization in w-F_1 for data reordering before `w-reduce` workers.

Figure 6.9 shows the compilation of a `map`-to-`w-reduce` composition. The first optimization step is the same as the one for `map`-to-`w-reduce` optimization (Fig. 6.8a), thus this step is not described for simplicity since it shows the forwarding nodes reduction. The centralization point w-F_1 is required to reorder items of the stream before windowing. The final reordering or the reduce value for each window is guaranteed by the C_2 Collector.

Composition of `map` and `w-p-reduce`



(A) Centralization in w-p-F_1 for data reordering before `w-p-reduce` workers



(B) Final network resulting from *All-to-All* simplification, where partitioning is done by `map` workers and windowing (for each partition) is done by `reduce` workers.

FIGURE 6.10: Compilation of `map`-to-`w-p-reduce` composition.

Figure 6.10 shows the compilation of a `map`-to-`w-p-reduce` composition. Again we omit the *Forwarding Simplification* optimization for simplicity. As resulting from the first *Forwarding Simplification* optimization, a centralization point w-p-F_1 is needed to reorder stream items. By applying the *All-to-All* optimization, it is possible to make the `map` operator partition data among downstream nodes. Hence the reducers reorder incoming data according to time-ordering, apply windowing on each partition and produce a reduced value for each window.

By comparing the optimized graphs in Figures 6.9 and 6.10, it can be noticed that adding a partitioning nature to windowed processing enables further optimization. This is not surprising since, as is discussed in [60], the KP pattern is inherently simpler than WF. The former can be implemented in such a way that each stream item is statically mapped to a single Worker, thus reducing both data traffic and coordination between processing units.

6.3 Stream Processing

In this section, we discuss some aspects of the proposed execution model from a stream processing perspective. In our setting, streams are defined as unbounded ordered sequences (cf. Section 5.3.1), thus we consider Dataflow graphs in which nodes process infinite sequences of tokens.

Historically, all the data-parallel operators we included in the PiCo syntax (e.g., `map`, `reduce`) are defined only for bounded data structures. Although at the syntactic level we allow the application of data-parallel operators to streams (e.g., by composing an unbounded `emit` to a `map` operator), this ambiguity is resolved at the semantic level, since the semantics of each unbounded operator is defined in terms of its bounded counterpart. For instance, the operators in the `combine` family are defined for unbounded collections only if they are endowed with a windowing policy. The semantics of the resulting windowing operator processes each window (i.e., a bounded list) by means of the respective bounded semantics (cf. Equation 5.12).

Let us consider the semantic `map` over streams. As we showed in Section 6.1.1, it is not possible to implement a strict unbounded semantic `map` since this would require reordering all the (infinite) collection at hand. For this reason, we introduced the weak semantic `map` that can be easily extended to the unbounded case. The resulting semantics is defined in terms of its bounded counterpart and exploits a batching mechanism (cf. Equation 5.13).

This approach immediately exposes a trade-off: the width of the batching windows is directly proportional to the amount of time-ordering enforced, but it is also directly proportional to the latency for processing a single item. If a stream is time-ordered (cf. Definition 2), there is no need for reordering, thus the minimum batching (i.e., width = 1) can be exploited to achieve the lowest latency. Conversely, if the stream is highly time-disordered and we want to restore the ordering as much as possible, we have to pay a price in terms of latency.

We remark that all the frameworks for data processing we consider for comparison expose similar trade-offs in terms of operational parameters, such as window triggering policies [72, 67]. We propose a symmetric approach, in which this aspect is embedded into the abstract semantics of PiCo programs—rather than the operational semantics of the underlying runtime.

6.4 Summary

In this Chapter we discussed how a PiCo Pipelines and operators are compiled into a directed acyclic graph representing the parallelization of a given application, namely the parallel execution dataflow. We showed how each operator is compiled into its corresponding parallel version, providing a set of optimization applicable when composing parallel operators. DAG optimization is present in all analytics frameworks presented previously in this thesis, and it is done at runtime. Once the application is executed, the execution DAG can be optimized by applying some heuristics that create the best execution plan or by applying some pipelining (i.e., in Spark stages) among subsequent operators - thus a strategy similar to optimizations proposed in PiCo. The main strength in PiCo approach is that all proposed optimizations are statically predefined and they can be provided as pre-built DAG implemented directly in the host language. Furthermore, optimizations do not involve operators and pipelines only, but it is also specific with respect to the structure type of the Pipeline.

Chapter 7

PiCo API and Implementation

In this chapter, we provide a comprehensive description of the actual PiCo implementation, both at user API level and at runtime level. We also present a complete source code example (Word Count), which is used to describe how a PiCo program is compiled and executed.

7.1 C++ API

In this section, we present a C++ API for writing programs that can be statically mapped to PiCo Pipelines, defined in Sect. 5.1. By construction, this approach provides a C++ framework for data processing applications, each endowed with a well-defined functional semantics (cf. Sect. 5.3) and a clear parallel execution model (cf. Chap. 6).

We opted for a fluent interface, exploiting method cascading (aka. method chaining) to relay the instruction context to a subsequent call. PiCo design exposed in previous chapters makes it independent from the choice of the implementation language. We decided to implement PiCo runtime and API entirely in C++, so that we can exploit explicit memory management, a more efficient runtime in terms of resources utilization, and it is possible to take advantage compile time optimizations. Furthermore, with C++ it is possible to easily exploit hardware accelerators, making it more suitable for high-performance applications. Besides this choice can affect portability, we think that the advantages carried by a C++ runtime can overcome the advantages of having a compile-once-run-everywhere paradigm provided by the JVM. On the other hand, C++ poses some limitations in terms of compatibility with well established software for data management used in analytics stacks (Kafka, Flume, Hadoop HDFS), which in some cases provide a limited or no C++ frontend. Consider for instance access to HDFS: besides it exposes a C++ library for read and write operations, it is very limited and provides a strongly reduced set of operations with respect to the Java API.

In the remainder of this section, we provide a full description of all entities in a C++ PiCo program, providing also some source code extracts. The current implementation is only for shared memory applications. In the future, an implementation of the FastFlow runtime for distributed execution by mean of a Global Asynchronous Memory(GAM) system model. A GAM system consists in a network of executors (i.e., FastFlow workers as well as PiCo operators) accessing a global dynamic memory with weak sharing semantics. With this model, the programming model, the semantics DAG and its compilation in the parallel execution DAG in PiCo will not change.

7.1.1 Pipe

A C++ PiCo program is a set of operator objects composed into a Pipeline object, processing bounded or unbounded data. A Pipeline can be:

- created as the empty Pipeline, as in the first constructor
- created as a Pipeline consisting of a single operator, as in the second and third constructors
- modified by adding an operator, as in the `add` member function
- modified by appending other Pipelines, as in the `to` member functions
- merged with another Pipeline, as in the `merge` member function
- paired with another Pipeline by means of a binary operator, as in the `pair` member function

`Pipe` **API**	
`Pipe()`	Create an empty Pipe
`template<typename T>` `Pipe(const T& op)`	Create a Pipe from an initial operator
`template<typename T>` `Pipe(T&& op)`	Create a Pipe from an initial operator (move)
`template<typename T>` `Pipe& add(const T& op)`	Add a new stage to the Pipe
`template <typename T>` `Pipe& add(T&& op)`	Add a new stage to the Pipe (move)
`Pipe& to(const Pipe& pipe)`	Append a Pipe to the current one
`Pipe& to(std::vector<Pipe*>` `pipes)`	Append a set of independent Pipes taking input from the current one

`template<typename In1,`
`typename In2, typename Out>`
`Pipe& pair(`
`const BinaryOperator`
`<In1, In2> out& a,`
`const Pipe& pipe)`
	Pair the current Pipe with a second pipe by a BinaryOperator that combines the two input items (a pair) with the function specified by the user

`Pipe& merge(const Pipe& pipe)`	Merge data coming from the current Pipe and the one passed as argument. The resulting collection is the union of the collection of the two Pipes
`void print_DAG()`	Print the DAG as adjacency list and by a BFS visit
`void run()`	Execute the Pipe
`void to_dotfile(std::string` `filename)`	Encode the DAG into a dot file
`void pipe_time()`	Return the execution time in milliseconds

TABLE 7.1: the `Pipe` class API.

Table 7.1 summarizes the Pipeline class API.

In addition to the member functions for creating, modifying and combining Pipeline objects, the last four member functions may only be called on *executable* Pipelines, namely those representing top-level PiCo Pipelines (cf. Definition 1). For a Pipeline object, to be executable is a property that can be inferred from its type. We discuss the typing of Pipeline objects in Sect. 7.1.3.

7.1.2 Operators

The second part of the C++ PiCo API represents the PiCo operators. By following the grammar in Sect. 5.1.2, we organize the API in a hierarchical structure of unary and binary operator classes. The design of the operators API is based on inheritance in order to follow in an easy way the grammar describing all operators, nevertheless we recognize that the use of template programming without inheritance would improve runtime performance. The implementation makes use of dynamic polymorphism when building the semantics DAG, where virtual member functions are invoked to determine the kind of operator currently processed.

`UnaryOperator` is the base class representing PiCo unary operators, those with no more than one input and/or output collection. For instance, a `Map` object takes a C++ callable value (i.e., the kernel) as parameter and represents a PiCo operator `map`, which processes a collection by applying the kernel to each item. Also `ReadFromFile` is a sub-class of `UnaryOperator` and it 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 instance a `BinaryMap` object represents a PiCo operator `b-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 `pair` member function (cf. Table 7.1).

Operator Constructors	
`template<typename In, typename Out>` `Map(std::function<Out(In&)> mapf)`	`Map` constructor by defining its kernel function *mapf*: $In \rightarrow Out$

`template<typename In, typename Out>`
`FlatMap(std::function<void(In&,`
`FlatMapCollector<Out>& flatmapf)`
	`FlatMap` constructor by defining its kernel function *flatmapf*: $\langle In, FlatMapCollector\langle Out\rangle\rangle \rightarrow void$

`template<typename In>`
`Reduce(std::function<In(In&, In&)>`
`reducef)`
	`Reduce` constructor by defining its kernel function *reducef*: $\langle In, In\rangle \rightarrow In$

`template<typename In>`
`PReduce(std::function<In(In&, In&)>`
`preducef)`
	`PReduce` constructor by defining its kernel function *preducef*: $\langle In, In\rangle \rightarrow In$ on partitioned input (i.e., reduce by key)

`template<typename In1, typename In2>`
`BinaryMap(std::function<Out(In1&, In2&)>`
`bmapf)`
	`BinaryMap` constructor by defining its kernel function *bmapf*: $\langle In1, In2\rangle \rightarrow Out$

`ReadFromFile()`	`ReadFromFile` constructor to read input data from file, each line is returned to the user as `std::string`
`template<typename Out>` `ReadFromSocket(char delimiter)`	`ReadFromSocket` constructor to read input data from Socket, lines are separated by a user-defined delimiter and returned to the user as `std::string`

`template<typename In>`
`WriteToDisk(std::function<std::string(In&)>`
`func)`
	`WriteToDisk` constructor to write to the specified textfile by defining its kernel function: $In \rightarrow void$

`template<typename In>`
`WriteToStdOut(std::function<std::string(In&)>`
`func)`
	`WriteToStdOut` constructor to write to the standard output by defining its kernel function: $In \rightarrow void$

TABLE 7.2: Operator constructors.

Table 7.2 summarizes the constructors of C++ PiCo operators. In the following sections, we describe all the implemented classes, showing also their inheritance class diagrams.

The `map` family

`Map` is a `UnaryOperator`. As we anticipated above, it represents the PiCo operator `map`, which process an input collection by applying a user-defined kernel (a C++ callable value) to each item.

Similarly, `FlatMap` represents the `flatmap` operator, which produces zero, one or more elements upon processing each item from the input collection. A `FlatMap` object takes as input a `FlatMapCollector` object, representing the storage for the items produced by each execution of the kernel. An example of the resulting interface is reported in Listing 7.1.

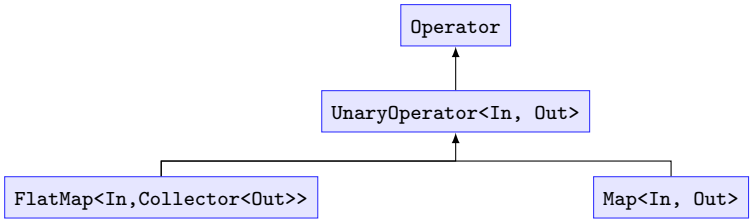


FIGURE 7.1: Inheritance class diagram for `map` and `flatmap`.

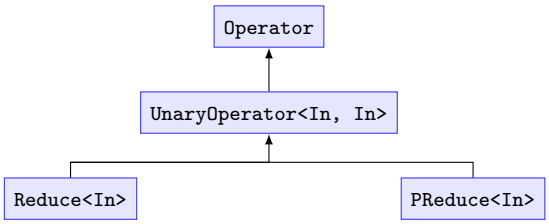


FIGURE 7.2: Inheritance class diagram for `reduce` and `p-reduce`.

In case of ordered collections, we opted for implementing the above classes in their weak semantic version (defined in Sect. 5.3.2, Eq. 5.13 for `map` operator) for both bounded and unbounded processing. This allows to design a simpler and more efficient runtime, as we will discuss later.

Fig. 7.1 reports the inheritance class diagram for `Map` and `FlatMap` classes.

The `combine` family

`Reduce` is the `UnaryOperator` representing the PiCo operator `reduce`. It synthesizes all the elements in the input collection into an atomic value, according to a user-defined binary kernel (a C++ callable value).

`PReduce` represents the PiCo operator `p-reduce`, thus it produces one synthesized value for each partition extracted from the input collection by the partitioning policy.

Fig. 7.2 reports the inheritance class diagram for `Reduce` and `PReduce` classes.

A windowing policy can be added to `Reduce` and `PReduce` objects by invoking the `window` member function. A windowing `Reduce` (or `PReduce`) logically splits the input collections into windows, according to the user-provided policy, and produces one synthesized value for each window.

Since we only provide a prototypical implementation, we only support the common by-key partitioning policy, thus only supporting reduction of key-value collections. Moreover, we only provide count-based tumbling windows.

Listing 7.1 illustrates a Pipeline using the partitioning and windowing variant of `Reduce`. In the example, the windowing policy is tumbling and count-based with width 8.

```
// define batch windowing
size_t size = 8;

// define a generic word-count pipeline
Pipe countWords;
countWords
  .add(FlatMap<std::string, std::string>(tokenizer))
  .add(Map<std::string, KV>([&](std::string in)
    {return KV(in,1);}))
  .add(PReduce<KV>([&](KV v1, KV v2)
    {return v1+v2;}).window(size));
```

LISTING 7.1: creating a simple Pipeline in the C++ PiCo API.

Sources and Sinks

As we will discuss in Sect. 7.1.3, source and sink objects play the crucial role of specifying the type of the collections processed by the Pipeline they belong to.

`ReadFromFile` and `ReadFromSocket` are sub-classes of `InputOperator`, a class representing PiCo sources, that produce the collections to be processed by the downstream Pipeline objects. The data populating the collections is read from either a text file, as for `ReadFromFile`, or a TCP/IP socket, as for `ReadFromSocket`.

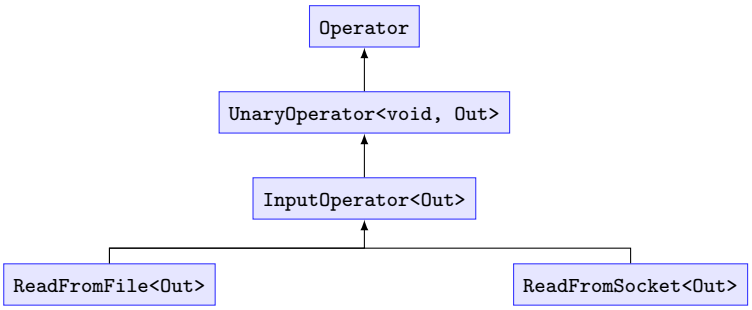


FIGURE 7.3: Inheritance class diagram for `ReadFromFile` and `ReadFromSocket`.

Figure 7.3 reports the inheritance class diagram for the `InputOperator` classes.

`WriteToDisk` and `WriteToStdOut` are sub-classes of `OutputOperator`, a class representing PiCo sinks, which consume the collections produced by the upstream Pipeline objects. The data is written to either a text file, as for `WriteToDisk`, or the standard output, as for `WriteToStdOut`. Moreover, a user-defined kernel (a C++ callable value) is used to process the data prior to writing them to the proper destination.

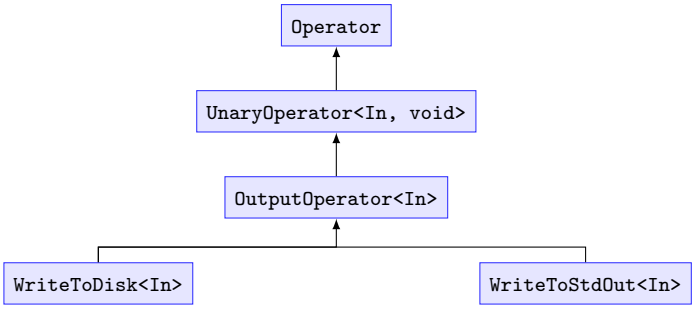


FIGURE 7.4: Inheritance class diagram for `WriteToDisk` and `WriteToStdOut`.

Figure 7.4 reports the inheritance class diagram for the `OutputOperator` classes.

Input file, output file, server name and port are passed to the application as arguments of the executable: `-i` and `-o` for the input and output file respectively, `-s` to define server address and `-p` to define the listening port.

We remark the presented API is a prototypical implementation, therefore a limited set of source and sink objects are provided only for exemplification.

7.1.3 Polymorphism

One distinguishing feature of the PiCo programming model, differently from state-of-the-art frameworks, is that the same syntactic object (e.g., a Pipeline or an operator) can be used to process data collections having different types. For example, the WORD-COUNT Pipeline in Algorithm 1 can be used to process bags, streams and lists once combined with proper sources and sinks. This feature is enabled by the type system we defined in Sect. 5.2 that allows polymorphic Pipelines and operators. More precisely, Pipelines and operators are polymorphic with respect to both the data type (the type of the collection items) and the structure type (the "shape" of the collection, 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 reported in Table 7.2, each operator takes a template parameter representing the data type of the collections processed by the operator. Moreover, each Pipeline object is decorated by the set of the supported structure type s.

We recall that all the polymorphism is dropped in top-level Pipelines. Therefore, executable Pipeline objects have an unique type. By construction, `InputOperator`s play the important role of specifying the unique structure type processed by the executable Pipeline they belong to. For instance, a Pipeline starting with an operator of `ReadFromFile` type will only process multi-sets, whereas a Pipeline starting with an operator of `ReadFromSocket` type will only process streams.

In the following paragraphs we provide some insight about the type checking and inference processes, which are an implementation of the type system discussed in Sect. 5.2.

Although Pipeline types are not fully exposed by the C++ API, they are exploited by the runtime to ensure only legal PiCo Pipelines are built, thus they are part of the API specification.

When a member function is called on a Pipeline object, the runtime performs the two following actions on the subject Pipeline:

1. it checks the legality of the call by inspecting the type of both the subject Pipeline and the call arguments (type checking)
2. it updates the type of the subject Pipeline (type inference)

We recall that each PiCo Pipeline has an input and output cardinality, either 0 or 1. For instance, top-level Pipelines have both input and output cardinality equal to zero. We say a Pipeline with non-zero input cardinality is a *consumer* since it consumes a collection, thus it needs to be prefixed by a source operator in order to be executed. Similarly, we say a Pipeline with non-zero output cardinality is a *producer*. For instance, the SIMPLE Pipe in Listing 7.1 is a producer.

When calling a member function on a Pipeline `p` object causing the addition of an operator `a` (i.e., `add` or `pair`), the invocation fails if any of the following conditions holds (i.e. type checking fails):

1. `p` is neither empty nor a producer, that is, it has already a sink operator (e.g., a `WriteToDisk`)
2. data type compatibility check fails, for instance because the output data type of `p` (i.e., the output data type of `p`'s last operator) differs from the the input data type of `a`
3. structure type s are incompatible, for instance `a` is a windowing operator and `p` only processes bags

Furthermore, after adding the operator `a`, the type of `p` is updated as follows:

1. `p` takes the `a`'s input or output degree if `a` is an input or output operator
2. the new `p`'s structure type is defined as the intersection of the structure type s of `p` and `a`

When modifying `p` by appending another Pipeline `q`, the `to` member function fails if any of the following conditions holds:

1. `p` is neither empty nor a producer
2. `p` is not empty and `q` is not a consumer
3. data type or structure type compatibility check fails

When appending multiple Pipelines (resulting into a branching Pipeline), the previous constraints are still checked for each Pipeline to be added.

In case of merging or pairing, in addition to the above conditions, it is also checked that at least one of the two Pipelines to be combined is a non-producer one. This way it is guaranteed that any Pipeline has 0 or 1 entry and exit points and therefore it can be attached to other Pipelines and operators.

7.1.4 Running Example: Word Count in C++ PiCo

Listing 7.2 shows a complete example of the Word Count benchmark.

```
typedef KeyValue<std::string, int> KV;

static auto tokenizer = [](std::string& in,FlatMapCollector<KV>& collector) {
    std::istringstream f(in);
    std::string s;
    while (std::getline(f, s, ' ')) {
        collector.add(KV(s,1));
    }
};

int main(int argc, char** argv) {
    // parse command line
    parse_PiCo_args(argc, argv);

    /* define a generic word-count pipeline */
    Pipe countWords;
    countWords
    .add(FlatMap<std::string, std::string>(tokenizer)) //
    .add(Map<std::string, KV>([&](std::string in)
      {return KV(in,1);}))
    .add(PReduce<KV>([&](KV v1, KV v2)
      {return v1+v2;}));

    /* define i/o operators from/to file */
    ReadFromFile reader();
    WriteToDisk<KV> writer([&](KV in) {
        return in.to_string();
    });

    /* compose the pipeline */
    Pipe p2;
    p2 //the empty pipeline
    .add(reader) // add single operator
    .to(countWords) // append a pipeline
    .add(writer); // add single operator

    /* execute the pipeline */
    p2.run();

    return 0;
}
```

Listing 7.2: Word Count example in PiCo.

7.2 Runtime Implementation

The runtime of PiCo is implemented on top of the FastFlow library, so that we use `ff_node`, `ff_pipeline` and `ff_farm` as building blocks for the representation of the parallel execution graph (as described in Chap. 6). In this section, we describe how each component of PiCo is effectively mapped to a FastFlow node or pattern. We recall that by exploiting the FastFlow runtime, it is possible to move only pointers to data among `ff_node`s, using these pointers as capabilities for synchronizations (see Sect. 2.4.1).

Pipe The Pipe class in the user API represents the main entity in a PiCo application. When the `run()` member function is invoked on a Pipe, the empty `ff_pipeline` corresponding to the Pipe is created. After that, the semantics DAG (Sect. 5) representing the application is visited and, recursively, `ff_farm`s and `ff_node`s are added as stages to the `ff_pipeline`, depending on the current visited node. We recall that, in FastFlow, each `ff_node` is a thread.

Map, Flatmap, Reduce and PReduce operators are implemented as `ff_farm`s. A *parallelism* parameter defines how many workers are created in each farm. The total number of threads per `ff_farm` is one for the Emitter node, one for the Collector node and *parallelism* number of threads for workers. Worker nodes execute the kernel code specified at API level on each item received. The Collector node receives data from workers in a from-any policy and forwards them to the next stage of the Pipe.

A different implementation for Emitter and Collector is provided for the `p-reduce` farm. In this case, the Emitter node has to take care of partitioning items on their group basis (i.e., by key). More precisely, the Emitter partitions the key space among workers, forwarding elements belonging to the same group always to the same worker (of course, it is possible for one worker to receive more than one key to process). The Collector will start gathering results only when workers are done with partitioned reduce.

Input and output operators are implemented as single `ff_node`s. These nodes have specialized code performing input and output operations. Moreover, input nodes are instructed to send an End Of Stream (EOS) when the input generation completes. The EOS is implemented into two distinct tokens: the *FF_EOS* is internal to the FastFlow runtime and it instructs `ff_node`s to terminate after its reception. The *PICO_EOS* is internal to PiCo and processed by all `ff_node`s, and it does not cause the termination of `ff_node`s. It is used, for instance, by the `p-reduce` Emitter node to instruct workers that the end of stream/file is reached and to forward their reduce results to the Collector before terminating with the *FF_EOS*.

To and Merge are Pipe member functions for appending and merging pipes, respectively. In this paragraph, we consider the MERGE member function having signature `Pipe& merge(const Pipe& pipe)`. The merging invocation results in the instantiation of a `ff_farm` where the two Pipes to be merged are added as a workers as new `ff_pipeline`s—this is possible thanks to the FastFlow composability property (see Sect. 2.4.1).

Figure 7.5 shows an example of the DAG resulting from merging three pipelines. *EntryPointMERGE* is the `ff_farm` Emitter node, while the *Merge* node is its Collector receiving all results to be forwarded to subsequent stages (in this case, a write-to-disk `ff_node` writing to `void.txt` file). The *EntryPointMERGE* broadcasts a *PICO_SYNC* token to the Pipes to be merged in order to start generating input items (read from `nopunct.txt` file). Further information about the *PICO_SYNC* token are reported in Section 7.3.4. We recall that the order of items is respected only locally to each Pipe, while there exists an interleaving order in the output from the Collector.

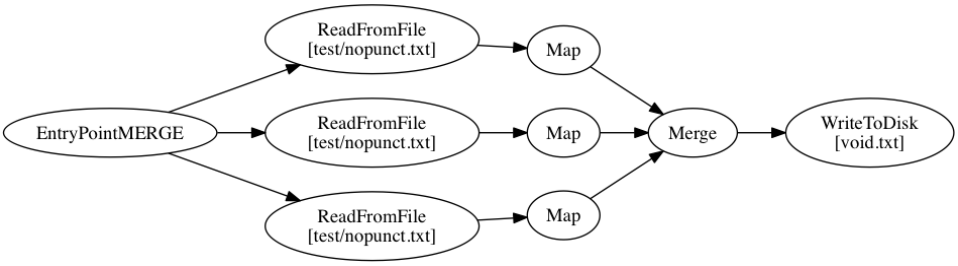


FIGURE 7.5: Semantics DAG resulting from merging three Pipes.

We recall that the order of items is respected only locally to each Pipe, while there exists a certain interleaving order in output from the Collector. When merging pipelines, the Emitter is always the first stage of the PiCo pipeline.

7.3 Anatomy of a PiCo Application

In this section, we provide a description of the PiCo execution model. For the sake of simplicity and completeness, we start from the source code of a PiCo application and we explain each step taken to reach the result.

7.3.1 User level

As a concrete example, shown in Listing 7.3, we use an application in which two Pipes are merged. These two Pipes take as input data from two files and map each line into a `<string, int>` pair. Then, each pair is written to a file.

```
typedef KeyValue<std::string, int> KV;
parse_PiCo_args(argc, argv);
/* define the operators */

auto map1=Map<std::string, KV>([](std::string in)
  {return KV(in, 1);});

auto map2=Map<std::string, KV>([](std::string in)
  {return KV(in, 2);});

auto reader=ReadFromFile<std::string>();

auto wtd=WriteToDisk<KV>([](KV in) {
      std::string value = "<";
      value.append(in.Key())
           .append(", ")
           .append(std::to_string(in.Value()))
           .append(">");
      return value;
  });

/* p1 read from file and process it by map1 */
Pipe p1(reader);
p1.add(map1);

/* p2 read from file and process it by map2 */
Pipe p2(reader);
p2.add(map2);

/* now merge p1 with p2 and write to file */
p1.merge(p2);
p1.add(wtd);

/* execute the pipeline */
p1.run();
```

Listing 7.3: Merging example in PiCo.

The input `reader` operator reads lines from a text file and simply returns each line read unmodified. Since data is read from disk, the structure type determined by the input operator is a bag (bounded, unordered). The output `wtd` operator takes all pairs of the type `KeyValue KV` and processes them with the user defined lambda expression.

Operators `map1` and `map2` take as input a line from their `reader` operator and produce a `KeyValue KV` pair: the input line is the key and the C++ lambda `auto`

`map1 = Map<std::string, KV>([](std::string in){return KV(in, 1)})` produces key-value pairs having input lines as key and value 1 as value, whereas `map2` gives value 2 to each pair. A copy of `reader` is used to compose the second pipe `Pipe p2(reader)`, which is then merged to the `p1` by `p1.merge(p2)`. We recall that compatibility check on structure type and the type check on last output type and new operator's input type is performed if the Pipe is not empty (Sect. 7.1.1).

7.3.2 Semantics dataflow

While composing the Pipe, the semantics dataflow graph is also created, that is, the dataflow representing the semantics of the (sequential) application in the form of a directed acyclic graph (see Chapter 4), where vertexes represent operators and edges represent data dependencies. This step is done while calling Pipe modifiers, thus no further Pipe preprocessing is needed to build the semantics DAG, represented as an adjacency list and implemented by a `std::map`. It is possible to visualize the DAG by invoking the `to_dotfile()` member function on the Pipe. The resulting DAG of the example application is shown in Figure 7.3.

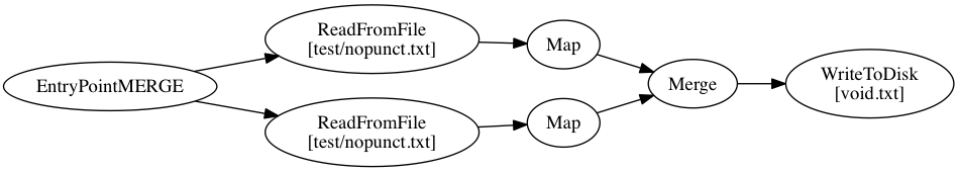


FIGURE 7.6: Semantics DAG resulting from the application in listing 7.3.

The class representing each node holds the information about its role in the semantics DAG. The role determines if the node is a processing node (i.e., an operator) or an auxiliary node that does not process data (i.e., the *EntryPointMERGE* node).

The role can have three different values: 1) *Processing*, representing an operator with user-defined code; 2) *EntryPoint/ExitPoint*, as the EntryPointMERGE and Merge nodes in Fig. 7.6 when merging Pipes (as well as Merge node in Fig. 7.7); 3) *BCast* node representing the starting point resulting from the invocation of the `to(std::vector<Pipe*> pipes)` member function as shown in Figure 7.7.

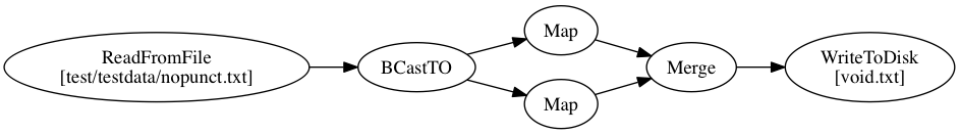


FIGURE 7.7: Semantics DAG resulting from connecting multiple Pipes.

7.3.3 Parallel execution dataflow

When the `run()` member function is called on `p1`, the semantics dataflow is processed to create the parallel execution dataflow. This graph represents the application in terms of processing elements (i.e., actors) connected by data channels (i.e., edges), where operators can be replicated to express data parallelism. This can be an intermediate representation of the possible parallelism exploitation, as shown in Sect. 6.1, where we provided the compilation of each operator in terms of patterns (e.g., farm). We implemented this intermediate representation directly in FastFlow by using nodes, farms and pipelines patterns.

The creation of the parallel execution dataflow is straightforward. Having an empty `ff_pipeline` `picoDAG` that will be executed, we then start visiting the first node of the semantics dataflow, which can be an input or an entry point node. On the basis of its role, a new `ff_node` or `ff_farm` is instantiated and added to `picoDAG`. The semantics DAG is recursively visited and the following operations are performed:

1. A single `ff_node` is added in case of input/output operators;
2. The corresponding `ff_farm` is added in case of operators different from I/O operators;
3. If an entry point is encountered, a new `ff_farm` is created and added to `picoDAG`;
   (a) a new `ff_pipeline` is created for each entry point's adjacent node;
   (b) these `ff_pipeline`s are built with new `ff_node`s created by recursively visiting the input Pipe's graph, until reaching the last node of each Pipe visited.

At the end, the resulting `picoDAG` is always a composition of `ff_pipeline`s and `ff_farm`s. Figure 7.8 shows the FastFlow network resulting from the example application with no optimization having been performed.

Each circle is a `ff_node`. Blue circles represent `ff_node`s executing user-defined code: read from file (`Rff`), the two map (`w1`, `wn`) and write to disk (`Wtd`). Green circles represent emitter and collector `ff_node`s: `Emap` and `Cmap` respectively for `map1` and `map2` `ff_farm`s, and `EPm` and `Cmerge` respectively for the `merge` `ff_farm`.

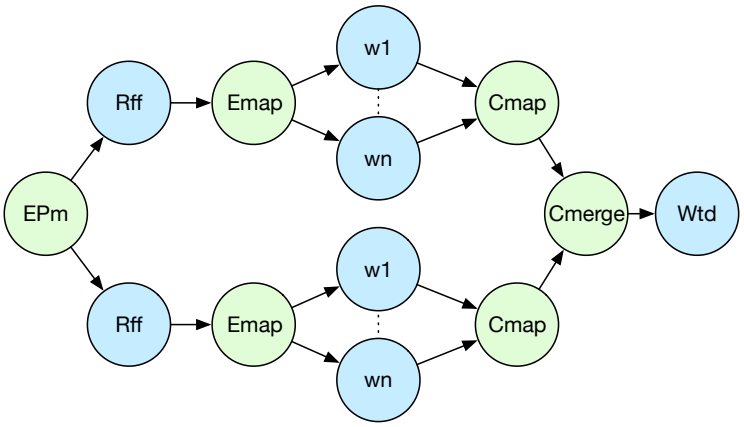


FIGURE 7.8: Parallel execution DAG resulting from the application in listing 7.3.

The figure represents the corresponding FastFlow network without any of the optimization proposed in Section 6.2. In this case, for instance, it would be possible to apply a *forwarding simplification* to unify `Cmerge` and `Cmap`s as well as unifying `Emap` and `Rff` `ff_node`s in each branch of the `EPm` `ff_node`.

7.3.4 FastFlow network execution

In this section, we provide the description of the execution of the `picoDAG` pipeline, starting from a brief summary of the FastFlow runtime.

From the orchestration viewpoint, the process model to be employed is a Communicating Sequential Processes CSP[1]/Actor hybrid model where processes (`ff_node`s) are named and the data paths between processes are explicitly identified (differently from the Actor model). The abstract units of communication and synchronization are known as *channels* and represent a stream of data exchanged between two processes. Each `ff_node` is a C++ class entering an infinite loop through its `svc()` member function where:

1. it gets a task from input channel (i.e., a pointer);
2. it executes business code on the task;
3. it puts a task into the output channel (i.e., a pointer)

Representing communication and synchronization by a channel ensures that synchronization is tied to communication and allows layers of abstraction at higher levels to compose parallel programs where synchronization is implicit. For a more complete review of FastFlow, please refer to Section 2.4.1.

Patterns to build a graph of `ff_node`s, such as farms, are defined in the *core patterns* level of FastFlow stack reported in Figure 2.8. Since the graph of `ff_node`s is a streaming network, any FastFlow graph is built using two streaming patterns (farm and pipeline) and one pattern-modifier (loopback, to build cyclic networks). These patterns can be nested to build arbitrarily large and complex graphs. Each `ff_node` in a `ff_pipeline` or `ff_farm` corresponds to a thread.

In a PiCo application, as the one in Fig. 7.8, a compiled DAG can start with an *input node* or with an *entry point* obtained by the compilation of a merge operator. In the case of an entry point (such as the first stage of the `ff_pipeline` in the example), this node is instructed to send to all its neighbor nodes a token (specific to the PiCo runtime, called *PICO_SYNC*), using a round-robin policy.

This token makes the computation start on input nodes `Rff` shown in Figure 7.8. When the input tokens generation ends, the `ff_node` corresponding to the `Rff` operator sends a *PICO_EOS* token and exits from the `svc()` member function[2]. On exit, it terminates its life-cycle by emitting the FastFlow *FF_EOS* token. Differently from all other operators `ff_node`s, which are executed as many times as the number of input data they process (i.e., on a stream of microbatches), an input node's `svc()` is executed only once and stops on input termination.

In the running example, both `Rff` nodes read lines from a file that are then forwarded to their following node of the pipeline. Tokens are sent out at microbatch granularity (in this case, a microbatch is a fixed size array of lines read from the input file).

Following our example, the next stages of both `Rff`s are the Emitters of the `map` farms. Since we implemented PiCo's parallel operators following the weak semantic map (defined in Sect. 5.3.2, eq. 5.13 for `map` operator) on both bounded or unbounded processing, the `map` `ff_farm`s process microbatches instead of single tokens.

Each worker of the `map` `ff_farm` processes the received microbatch by applying the user-defined function. Then each worker allocates a new microbatch to store the result of the user-defined function, and then deleting the received microbatch. The new microbatch is forwarded to the next node. The general behavior of a worker during its `svc()` call is that it deletes each input microbatch (allocated by the Emitter) after it has been processed and the results of the kernel function (applied to all elements of the microbatch) are stored into a new microbatch. When

[1]The CSP model describes a systems in terms of component processes operating independently, which interact with each other through message-passing communication.

[2]We recall that the `svc()` member function of a `ff_node` executes the kernel code of the operator.

received, *PICO_EOS* or *PICO_SYNC* tokens are forwarded as well. When the Collector receives *PICO_EOS* tokens from all workers, it then forwards the PiCo end of stream token to the next stage, namely the `Cmerge` node.

The `Cmerge ff_node` implements a from-any policy to collect results. Then it forwards results to the next stage, in this case the output operator `Wtd`. This last node is a single sequential `ff_node` (we recall that input and output processing nodes are always sequential), writing the received data to a specified file. When `Wtd` receives *PICO_EOS*, the file is closed and the computation terminates.

At this point, the FastFlow runtime manages `ff_node`s destruction and runtime cleanup.

7.4 Summary

In this Chapter we provided a description of the actual PiCo implementation, both at user API level and at runtime level. PiCo API implementation follows a fluent interface, exploiting method cascading (aka. method chaining) to relay the instruction context to a subsequent call. It is implemented entirely in C++, so that we can exploit explicit memory management, a more efficient runtime in terms of resources utilization, and it is possible to take advantage compile time optimizations. The design of the operators API is based on inheritance in order to follow in an easy way the grammar describing all operators, nevertheless we recognize that the use of template programming without inheritance would improve runtime performance. The implementation makes use of dynamic polymorphism when building the semantics DAG, where virtual member functions are invoked to determine the kind of operator currently processed.

We described the anatomy of PiCo runtime starting from a complete source code example (Word Count). The Pipeline described in the source code is first provided as the semantics DAG, then we went through the compilation step, in which the Pipeline is compiled into the parallel execution graph (namely, the parallel execution DAG). The parallel execution graph, which is agnostic with respect to the implementation, is converted to a FastFlow network of patterns that results from the composition of `ff_pipeline` and `ff_farm`s. Finally, the effective execution is described, which is available in shared memory only. In the future, an implementation of the FastFlow runtime for distributed execution by mean of a Global Asynchronous Memory(GAM) system model. A GAM system consists in a network of executors (i.e., FastFlow workers as well as PiCo operators) accessing a global dynamic memory with weak sharing semantics.

Chapter 8

Case Studies and Experiments

This chapter provides a set of experiments based on examples defined in Sect. 5.4. We compare PiCo to Flink and Spark, focusing on expressiveness of the programming model and on performances in shared memory.

8.1 Use Cases

We tested PiCo with both batch and stream applications. A first set of experiments comprehends the word count and stock market analysis. We now describe each use case, also providing source code snapshots reporting classes related to the core of each application. All listings are reported in Appendix A.

8.1.1 Word Count

Considered as the "Hello, World!" of Big Data analytics, a word count application typically belongs to batch processing.

The input is a text file, which is split into lines. Then, each line is tokenized into words: this operation is implemented as a `flatmap`, which outputs a number of items depending on the input line. Each of these items, namely the words of the input file, are processed by a `map` operator which produces a key-value pair $< w, 1 >$ for each word w. After each word has been processed, all pairs are grouped by the values of each pair are reduced by a sum. The final result is a single pair for each word, where the value represents the number of occurrences of a word in the text.

A PiCo word count application can be found in listing 7.2.

The source code for Flink and Spark Word Count are shown in listings A.3 and A.6 respectively.

8.1.2 Stock Market

The following examples implements the use cases reported in Sect. 5.4. The first use case implements the "Stock Pricing" program that computes a price for each option read from a text file. Each line is parsed to extract stock names followed by stock option data. A `map` operator then computes prices by means of the Black & Scholes algorithm for each option and, finally, a reducer extracts the maximum price for each stock name. In the following sections, we provide source codes extracts for implementation in PiCo (listing A.1), Flink (listing A.4) and Spark (listing A.7).

8.2 Experiments

The architecture used for experiments is the Occam Supercomputer (Open Computing Cluster for Advanced data Manipulation) [127, 8], designed and managed by the University of Torino and the National Institute for Nuclear Physics.

Occam has the following technical characteristics:

- **2 Management Nodes**: *CPU* - 2x Intel® Xeon® Processor E5-2640 v3 8 core 2.6GHz, *RAM* - 64GB/2133MHz, *Disk* - 2x HDD 1Tb Raid0, *Net* - IB 56Gb + 2x10Gb + 4x1GB, *FormFactor* - 1U
- **4 Fat Nodes**: *CPU* - 4x Intel® Xeon® Processor E7-4830 v3 12 core/2.1Ghz, *RAM* - 768GB/1666MHz (48 x 16Gb) DDR4, *Disk* - 1 SSD 800GB + 1 HDD 2TB 7200rpm, *Net* - IB 56Gb + 2x10Gb
- **32 Light Nodes**: *CPU* - 2x Intel® Xeon® Processor E5-2680 v3, 12 core 2.5Ghz, *RAM* - 128GB/2133 (8x16 Gb), *Disk* - SSD 400GB SATA 1.8 inch, *Net* - IB 56Gb + 2x10Gb, *FormFactor* - high density (4 nodes x RU)
- **4 GPU Nodes**: *CPU* - 2x Intel® Xeon® Processor E5-2680 v3, 12 core 2.1Ghz, *RAM* - 128GB/2133 (8x16Gb) DDR4, *Disk* - 1 x SSD 800GB sas 6 Gbps 2.5 inch, *Net* - IB 56Gb + 2x10Gb, *GPU* - 2 x NVIDIA K40 on PCI-E Gen3 x16
- **Scratch Disk**: *Disk* - HDD 4 TB SAS 7200 rpm, *Capacity* - 320 TB RAW + 256 TB usable, *Net* - 2 x IB 56Gb FDR + 2 x 10Gb, *File System* - Lustre Parallel Filesystem
- **Storage**: *Disk* - 180 x 6 TB 7200 rpm SAS 6Gbps, *Capacity* - 1080 TB raw 768 TB usable, *Net* - 2 x IB 56Gb + 4 x 10GbE, *File System* - NFS export, *Fault Tolerance* - RAID 6 Equivalent with Dynamic Disk Pools
- **Network**: *IB layer* - 56 Gbps, *ETH10G layer* - 10 Gbps, *IB topology* - FAT-TREE, *ETH10G topology* - FLAT

We performed tests on scalability and best execution time comparing PiCo to Spark and Flink on batch and stream applications. The current experimentation is only on shared memory applications. In the future, an implementation of the FastFlow runtime for distributed execution by mean of a Global Asynchronous Memory(GAM) system model[1], so that we will be able to have a performance evaluation also in a distributed memory model.

8.2.1 Batch Applications

In the first set of experiments, we run the Word Count and Stock Pricing examples reported in the previous section. First, we show scalability obtained by PiCo with respect to the number of parallel threads used for parallel operators.

We compute the scalability factor as the relation between the execution time with parallelism 1 (i.e., sequential operators) and the execution time obtained by exploiting more parallelism. It is defined as $T_1/T_i \, \forall \, i \in \{1, \ldots, nc\}$ with nc representing the number physical cores of the machine. The maximum number of physical cores is 48 on Fat Nodes (4 x Intel® Xeon® Processor E7-4830 v3 12 core/2.1Ghz).

[1] A GAM system consists in a network of executors (i.e., FastFlow workers as well as PiCo operators) accessing a global dynamic memory with weak sharing semantics.

PiCo

By default, PiCo allocates a single worker thread in each farm corresponding to an operator, and allocates microbatches with size 8 to collect computed data (see Sect. 6.1.1 for batch semantics). We tested PiCo with different microbatch sizes to measure its scalability also with respect to number of microbatches allocations. The best size for microbatch is 512 elements, which naturally leads to a reduction of dynamic memory allocations.

To increase performance, we also used two different threads pinning strategy. The default pinning strategy in Occam is interleaved on physical cores first, so that each consecutive operator of the pipeline is mapped to a different socket of the node. In this scenario, a linear mapping using physical cores first helped in reducing memory access time to data, since allocations are done mainly in the memory near the thread that is accessing that data.

Figure 8.1 shows scalability and execution times for the Word Count application: each value represents the average of 20 runs for each number of workers, the microbatch size is 512, and the thread pinning strategy is physical cores first. The size of the input file is 600MB. It a text file of random words taken from a dictionary of 1K words. In the Word Count pipeline, PiCo instantiates a total of 5 fixed threads (corresponding to sequential operators), plus the main thread, plus a user-defined number of workers for the `flatmap` operator. To exploit at most 48 physical cores, we can run at most 42 worker threads.

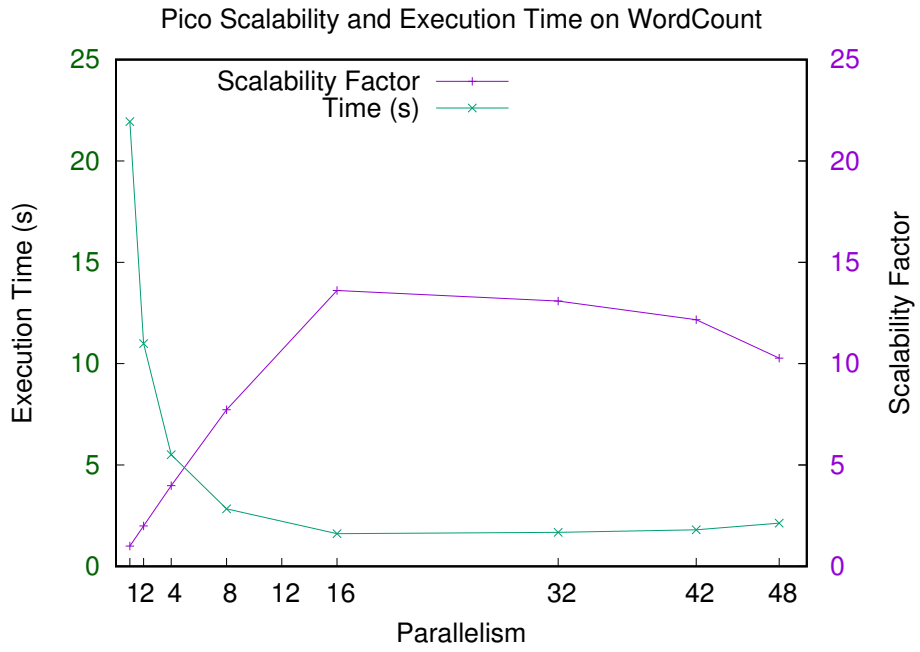


FIGURE 8.1: Scalability and execution times for Word Count application in PiCo.

With 16 and 32 workers for the `map` operator, mapped on physical cores, PiCo obtains similar average execution times: the best average execution time obtained is 1.61 seconds with 16 workers and a scalability factor of 13.60.

Figure 8.2 shows scalability and execution times for the Stock Pricing application: each value represents the average of 20 runs for each number of workers, the microbatch size is 512, and the thread pinning strategy is linear on physical cores first. In

the Stock Pricing pipeline, PiCo instantiates a total of 4 fixed threads (corresponding to sequential operators), plus the main thread, plus a user-defined number of workers for the `map` +`p-reduce` operator.

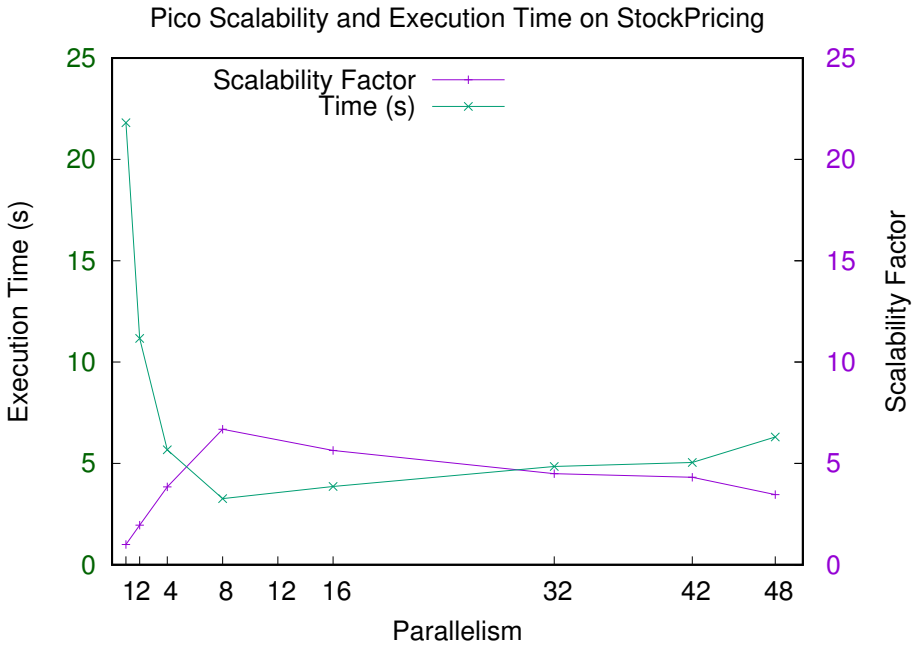


FIGURE 8.2: Scalability and execution times for Stock Pricing application in PiCo.

With 8 workers for the `map` + `p-reduce` operator, mapped on physical cores, PiCo obtains the best average execution time of 3.26 seconds and a scalability factor of 6.68.

Limitations on scalability. The applications tested in PiCo showed a common behavior on the first node of each Pipeline—the Read From File (RFF) operator. As the number of workers increases, it increases also the execution time of the read from file, making it the application's bottleneck. The RFF's kernel code is the following: each line of the input file is read in a `std::string` and stored into a fixed-size microbatch. Once the microbatch is full, it is forwarded to the next operator and a new microbatch is allocated. This is repeated until the end of file is reached. As the microbatch size increases, the number of their allocation decreases but RFF still represents the bottleneck. In tables 8.2 and 8.1, we report execution times of a single run of Word Count and Stock Pricing with two different microbatch sizes (8 and 512), showing how the scalability on workers is still increasing despite the bottleneck on RFF, giving room for improvement on the total execution time.

Microbatch size 8 (execution time in ms)				
workers	**exec. time**	**read from file**	**worker**	**worker scalability**
1	21874.50	901.98	21685.10	1.00
2	10807.80	915.61	10786.20	2.01
8	2916.80	1016.87	2753.39	7.87
16	1718.20	1711.02	1535.38	14.12
32	1755.85	1743.04	924.14	23.46
Microbatch size 512 (execution time in ms)				
workers	**exec. time**	**read from file**	**worker**	**worker scalability**
1	22091.60	869.07	22087.20	1.00
2	11030.70	861.86	11026.50	2.00
8	2841.45	932.27	2799.31	7.89
16	1635.55	1595.28	1607.74	13.74
32	1603.78	1589.13	917.44	24.07

TABLE 8.1: Decomposition of execution times and scalability highlighting the bottleneck on ReadFromFile operator in the Word Count benchmark.

Microbatch size 8 (execution time in ms)				
workers	**exec. time**	**read from file**	**worker**	**worker scalability**
1	22286.70	2391.62	22048.00	1.00
2	11549.10	2524.72	11430.20	1.93
8	3588.49	3586.57	3131.49	7.04
16	5135.71	5133.08	1730.37	12.74
32	6096.26	6092.02	934.54	23.59
Microbatch size 512 (execution time in ms)				
workers	**exec. time**	**read from file**	**worker**	**worker scalability**
1	21775.10	2123.27	21766.70	1
2	11289.20	2201.70	10946.40	1.99
8	3328.80	3326.84	2967.67	7.33
16	3370.84	3768.64	2510.13	8.67
32	4707.89	4704.69	868.11	25.07

TABLE 8.2: Decomposition of execution times and scalability highlighting the bottleneck on ReadFromFile operator in the Stock Pricing benchmark.

Tables 8.1 and 8.2 show that, despite the high scalability reached by workers, the total execution time is limited by the execution time of the read from file. Times for read from file in table 8.2 are higher than in table 8.1 and this is due to the number of lines read in the Stock Pricing benchmark (10M lines with respect to 2M in Word Count): this reflects to a $5x$ higher number of allocations. Worker scalability is still not affected by the bottleneck in RFF operator.

We believe that this behavior comes from allocation contention, which could be relaxed by using an appropriate allocator. For instance, FastFlow provides a specialized allocator that allocates only large chunks of memory, slicing them up into little chunks all with the same size that can be reused once freed. Only one thread can perform allocation operations while any number of threads may perform deallocations using the allocator. An extension of the FastFlow allocator might be used by any number of threads to dynamically allocate/deallocate memory. Both are based on the idea of the Slab Allocator [38].

Comparison with other frameworks

We compared PiCo to Flink and Spark on both Word Count and Stock Pricing batch applications. In this section, we provide a comparison on minimum execution

time obtained by each tool as the average of 20 runs for each application, a study on execution times variability, and metrics on resources utilization. Table 8.3 reports all configuration used to run the various tools.

PiCo		
Parallelism	**Microbatch Size**	**Other Conf**
1-48	512	-
Flink		
Parallelism	**Task Slots**	**Other Conf**
1-32	48	Default
Spark		
Parallelism		**Other Conf**
1-48	-	Default

TABLE 8.3: Execution configurations for tested tools.

In Flink, each process has one or more *task slots*, each of which runs one pipeline of parallel tasks, namely, multiple successive tasks such as, for instance, the *n-th* parallel instance of a `map` operator. It is suggested to set this value to the number of physical cores of the machine. In Flink programs, the *parallelism* parameter determines how operations are split into individual tasks, which are assigned to task slots; that is, it defines parallelism degree for data parallel operators. By setting the parallelism to N, Flink tries to divide an operation into N parallel tasks computed concurrently using the available task slots. The number of task slots should be equal to the parallelism to ensure that all tasks can be computed in a task slot concurrently but, unfortunately, by setting parallelism to a value greater than 32 the program crashes because of insufficient internal resource availability.

It is also possible to run Spark on a single node in parallel, by defining the number of threads exploited by data parallel operators.

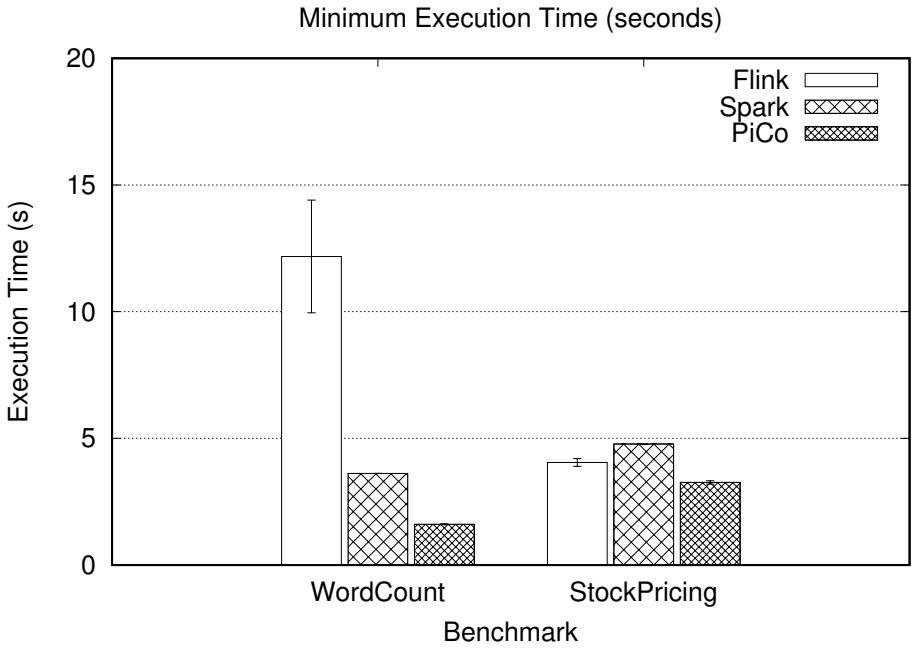


FIGURE 8.3: Comparison on best execution times for Word Count and Stock Pricing reached by Spark, Flink and Pico.

Figure 8.3 shows that PiCo reaches in both cases the best execution time. Table 8.4 also shows the average and the variability of execution times obtained by all the executions, showing that all the frameworks suffer from limited scalability after 32 cores. We remark that "parallelism" has different meanings for each tool that is described in above in this paragraph. PiCo shows a lower coefficient of variation with respect to Flink and Spark in almost all cases. To determine if any of the differences between the means of all collected execution times are statistically significant, we run the ANOVA test on the execution times obtained with the configuration used to obtain the best execution time. The test reported very low p-value even if coefficient of variation and standard deviations reported by Flink are very high. Therefore, according to the ANOVA test, the differences between the means are statistically significant. Table 8.4 shows that Flink is generally less stable than Spark, reaching a peak of 18.25% for the coefficient of variation. This behavior can be associated to different task scheduling to different executors at runtime, causing unbalanced workload (we recall that, in Flink, a task is a pipeline containing a subset of consecutive operators of the application graph). In Spark, the whole graph representing an application is replicated on all executors, so the probability of causing an unbalance is mitigated.

Spark							
Word Count							
Parallelism	1	2	4	8	16	32	48
avg (ms)	20983.24	12769.16	8846.12	5231.34	4159.26	**3617.80**	4211.80
sd (s)	0.35	0.16	0.22	0.42	0.35	0.13	0.22
cv (%)	1.66	1.29	2.49	8.04	8.34	3.59	5.33
avg Scal.	1	1.64	2.37	4.01	5.04	5.80	4.98
Stock Pricing							
Parallelism	1	2	4	8	16	32	48
avg (ms)	32970.27	20836.11	14719.28	6970.84	5158.77	**4773.17**	4983.87
sd (s)	0.21	0.30	0.25	0.14	0.11	0.14	0.19
cv (%)	0.63	1.43	1.70	2.00	2.23	2.87	3.79
avg Scal.	1	1.58	2.24	4.73	6.39	6.91	6.61
Flink							
Word Count							
Parallelism	1	2	4	8	16	32	48
avg (ms)	80095.40	44129.10	26043.35	16765.25	13622.20	**12179.00**	-
sd (s)	571.17	1368.31	1718.55	1939.74	1702.42	2222.3	-
cv (%)	0.71	3.10	6.60	11.57	12.50	18.25	-
avg Scal.	1	1.81	3.07	4.78	5.88	6.58	-
Stock Pricing							
Parallelism	1	2	4	8	16	32	48
avg (ms)	36513.6	19638.85	10807.9	6846.6	4787.2	**4048.35**	-
sd (s)	450.67	317.02	179.88	108.35	153.46	154.05	-
cv (%)	1.23	1.61	1.66	1.58	3.20	3.80	-
avg Scal.	1	1.86	3.38	5.33	7.63	9.02	-
PiCo							
Word Count							
Parallelism	1	2	4	8	16	32	48
avg (ms)	21938.15	10991.91	5510.19	2838.84	**1612.42**	1676.11	2135.24
sd (s)	57.51	45.73	24.09	23.95	20.74	31.82	60.51
cv (%)	0.26	0.42	0.44	0.84	1.29	1.90	2.83
avg Scal.	1	1.20	3.98	7.73	13.60	13.09	10.27
Stock Pricing							
Parallelism	1	2	4	8	16	32	48
avg (ms)	21807.41	11166.00	5673.48	**3261.60**	3865.59	4852.38	6305.70
sd (s)	176.56	335.43	126.50	66.29	44.62	63.53	55.83
cv (%)	0.81	3.00	2.23	2.03	1.15	1.30	0.88
avg Scal.	1	1.95	3.84	6.69	5.64	4.49	3.46

TABLE 8.4: Average, standard deviation and coefficient of variation on 20 runs for each benchmark. Best execution times are highlighted.

To examine resource consumptions, we measured CPU and memory utilization using the *sar* tool, which collects and displays all system activities statistics. The *sar* tool is part of the global system performance analysis *sysstat* package on Unix systems. We executed 10 runs for each configuration of the examined tools: results obtained show a low variability, so that we do not report average and variance of collected results. Table 8.5 shows CPU utilization percentages throughout the total execution time only for best execution times.

Word Count (execution time in ms)				
	Best exec. time	**Parallelism**	**CPU (%)**	**RAM (MB)**
Flink	12179.00	32	21.39%	3538.94
Spark	3617.80	32	12.94%	1494.22
PiCo	1612.42	16	19.38%	157.29
Stock Pricing (execution time in ms)				
	Best exec. time	**Parallelism**	**CPU (%)**	**RAM (MB)**
Flink	4048.35	32	22.06%	3460.30
Spark	4773.17	32	12.83%	1494.22
PiCo	3261.61	8	13.59%	78.64

TABLE 8.5: User's percentage usage of all CPUs and RAM used in MB, referred to best execution times.

Interesting results refer to the in RAM memory footprint, in which PiCo outperforms the other tools. Table 8.5 shows RAM MegaBytes used by each tool for each application. This confirms that both Spark and Flink maintain a constant amount of allocated resources in memory This is due to the fact that there is a resource preallocation managed by an internal allocator, which is in charge of reducing the overhead induced by the Garbage Collector. Hence, independently from the input size that, (about 600MB and 650MB in Word Count and Stock Pricing respectively) it is not possible to evaluate a correlation between input size and global memory footprint. For instance, Spark's *Project Tungsten* [59] introduces an explicit memory manager based on the `sun.misc.Unsafe` package, exposing C-style memory access (i.e., explicit allocation, deallocation, pointer arithmetic, etc.) in order to bypass the JVM garbage collection. As for Flink, it implements a Memory Manager pool, that is, a large collection of buffers (allocated at the beginning of the application) that are used by all runtime algorithms in which records are stored in serialized form. The Memory Manager allocates these buffers at startup and gives access to them to entities requesting memory. Once released, the memory is given back to the Memory Manager. PiCo does not yet rely on any allocator, so there is still room for improvement in its memory footprint.

8.2.2 Stream Applications

In this set of experiments, we compare PiCo to Flink and Spark when executing stream applications.

Spark implements its stream processing runtime over the batch processing one, thus exploiting the BSP runtime on stream microbatches, without providing a concrete form of pipelining and reducing the real-time processing feature. Flink and PiCo implements the same runtime for batch and streaming.

The application we test is the Stock Pricing (the same as for batch experiment), to which we added two more option pricing algorithms: Binomial Tree and Explicit Finite Difference. The Binomial Tree pricing model traces the evolution of the option's key by means of a binomial lattice (tree), for a number of time steps between the valuation and expiration dates. The Explicit Finite Difference pricing model is used to price options by approximating the continuous-time differential equation describing how an option price evolves over time by a set of (discrete-time) difference equations. The final result of the Stock Pricing use case is, for each stock, the maximum price variance obtained by the three algorithms (Black & Scholes, Binomial Tree, and Explicit Finite Difference). The input stream is of 10M stock options: each item is composed by a stock name and a fixed number of option values

In Appendix A, we present the partial source code for streaming Stock Pricing in Flink (listing A.5), Spark (listing A.8) and PiCo (listing A.2), so that it is possible also to show which are the difficulties in moving from a batch to a stream processing program in each case.

PiCo

Figure 8.4 shows scalability and execution times for the Stock Pricing streaming application: each value represents the average of 20 runs for each number of workers, and the thread pinning strategy is linear on physical cores first.

In the Stock Pricing pipeline, PiCo first instantiates 6 threads corresponding to sequential operators, such as read from socket and write to standard output, plus Emitter and Collector threads for `map` and `p-reduce` operators. Then, there is the main thread, plus k — a user-defined number — workers for the `map` and k for the `w-p-reduce` operators.

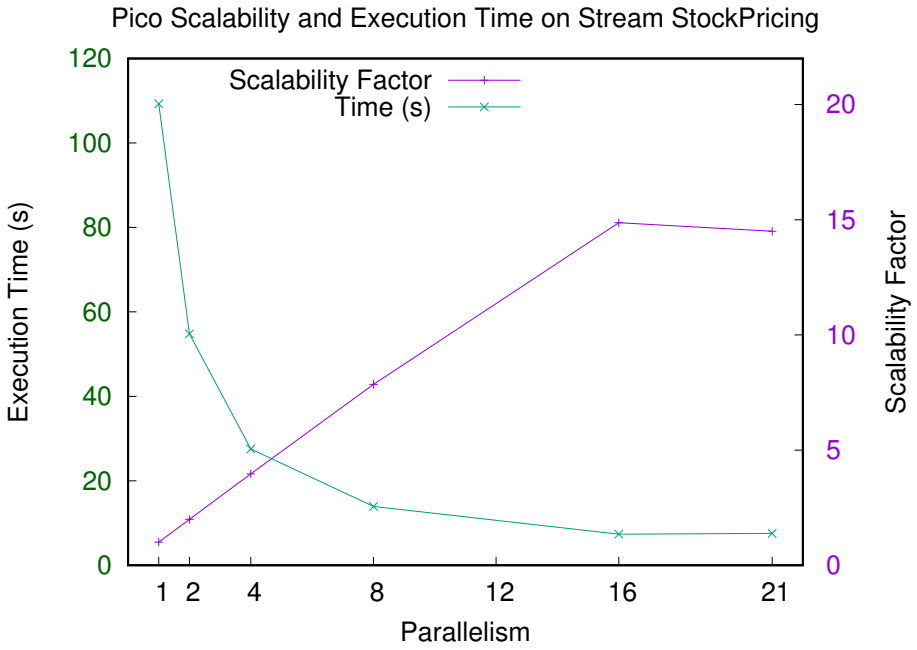


FIGURE 8.4: Scalability and execution times for Stream Stock Pricing application in PiCo.

With 16 workers for the `map` and 16 workers `w-p-reduce` operator, mapped on physical cores, PiCo obtains the best average execution time of 7.348 seconds and a scalability factor of 14.87.

Limitations on scalability. This use case also shows the slow down on the first node of the Pipeline—the ReadFromSocket (RFS) operator. As the number of workers increases, this increases also the execution time of the read from socket, making it the application's bottleneck, as was shown in the batch use cases reported in Table 8.6. The RFS's kernel code is the following: each line read from the socket is stored into a `std::string` when the delimiter is reached (in this example, "`\n`") and stored into a fixed-size microbatch. Once the microbatch is full, it is forwarded to the next operator and a new microbatch is allocated. This is repeated until the end of stream is reached. As the microbatch size increases, the number of their allocation decreases but RFS still represents the bottleneck. In the following table,

we report execution times of a single run of Stock Pricing with microbatch size 512, since previous tests showed that it represents the best granularity for good performance. We then compare execution times obtained with the two pinning strategy (interleaved and linear). The table shows how the scalability on workers is still increasing despite the bottleneck on RFS and how the two different pinning strategy help in reducing this effect, giving room for improvement on the total execution time. We show execution times for a number of workers up to a maximum of 21 for the `map` and `w-p-reduce` each, in order to exploit all physical cores of the node (48).

Interleaved Pinning Strategy (execution time in ms)				
workers	**exec. time**	**read from socket**	**map worker**	**map scalability**
1	100966.00	3882.59	100966.00	1.00
2	50870.60	4315.00	50870.00	1.98
4	28093.00	4576.83	28023.50	3.60
16	8855.81	8847.47	8848.53	11.41
21	8753.42	8745.43	8750.95	11.54
Linear Pinning Strategy (execution time in ms)				
workers	**exec. time**	**read from socket**	**map worker**	**map scalability**
1	109184.00	3962.74	109184.00	1.00
2	54747.90	3979.72	54747.50	1.99
4	27426.10	4064.94	27423.40	3.98
8	13927.80	4245.09	13926.20	7.84
16	7386.69	6341.60	7277.45	15.00
21	7548.15	7539.83	7545.49	14.47

TABLE 8.6: Decomposition of execution times and scalability highlighting the bottleneck on ReadFromSocket operator.

Comparison with other tools

We compared PiCo to Flink and Spark on the Stock Pricing streaming application. In this section, we provide a comparison on minimum execution time obtained by each tool as the average of 20 runs for each application, a study on execution times variability, and metrics on resources utilization. Configuration used for all tools are the reported in Table 8.3. We executed PiCo with a maximum of 21 worker threads for each data parallel operator (`map` and `w-p-reduce`), as previously defined, in order to exploit the maximum number of physical threads of the node. The window is count-based (or tumbling) and has size 8 in Flink and PiCo. In Spark, it is not possible to create count-based windows since only time-based ones are available, so we created tumbling windows with duration of 10 seconds, by which the execution time with one thread is similar to PiCo's one. We have to remark that comparison with Spark is not completely fair, since the windowing is not performed in a count-based fashion. To the best of our knowledge, it is not possible to implement such windowing policy in Spark.

We briefly recall some aspects of Flink's and Spark's runtime. In Flink, each process has one or more *task slots*, each of which runs one pipeline of parallel tasks (i.e., the nth instance of a `map` operator on the n-th partition of the input). The *parallelism* parameter determines how operations are split into individual tasks, which are assigned to task slots, that is, it defines parallelism degree for data parallel operators. By setting the parallelism to N, Flink tries to divide an operation into N parallel tasks computed concurrently using the available task slots. The number of task slots should be equal to the parallelism to ensure that all tasks can be computed in a task slot concurrently but, unfortunately, by setting parallelism to a value greater than 32, the program crashes because of insufficient internal resource availability. Due to this limitation, we run Flink with up to 21 worker threads, in order to be aligned with PiCo. We used the same maximum number of threads also in Spark,

even though it would be possible to exploit more parallelism: since Spark scalability in this use case is limited, it is not unfair to be aligned with PiCo's parallelism exploitation.

For stream processing, Spark implements an extension through the Spark Streaming module, providing a high-level abstraction called *discretized stream* or *DStream*. 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. All RDDs in a DStream are processed in order, whereas data items inside an RDD are processed in parallel without any ordering guarantees. Hence, Spark implements its stream processing runtime over the batch processing one, thus exploiting the BSP runtime on stream microbatches, without providing a concrete form of pipelining and reducing the real-time processing feature.

Stream Stock Pricing Best Execution Time (ms)			
	Best exec. time	**Parallelism**	**Scalability**
Flink	24784.60	16	9.21
Spark	42216.21	16	2.24
PiCo	7348.58	16	14.87

TABLE 8.7: Flink, Spark and PiCo best average execution times, showing also the scalability with respect to the average execution time with one thread.

Table 8.7 shows in a summarized manner the best execution times obtained by each tool, extracted from Table 8.8.

The table shows that PiCo reaches the best execution time with a higher scalability with respect to other tools, with a scalability of 14.87 in PiCo while 9.21 in Flink and 2.24 Spark.

Table 8.8 also shows the average and the variability of execution times obtained by all the executions. We remark that "parallelism" has different meanings for each tool, as described in the paragraph above. It shows that, also in this use case, Flink is slightly less predictable[2] than PiCo, reaching a peak of 2.06% for the coefficient of variation.

A strong variability is reported for Spark. We have seen, from the web user interface provided by Spark that the input rate from sockets has a great variability, introducing latencies that lead to a reported average of 5 seconds of task scheduling delay. Furthermore, it can be noticed that the best average execution time in Spark has a coefficient of variation of 38.90%, with a minimum execution time of 24 seconds and a maximum of 74 seconds (values not reported) and that, in all cases, Spark and Flink suffer from scalability issues. PiCo outperforms both Spark and Flink in terms of standard deviation, and outperforms Spark in terms of coefficient of variation. We again validated results with the ANOVA test, which validated our results.

[2]We consider predictability of execution times since the execution happened to fail, as we reported above in this section.

Flink Stream Stock Pricing						
Parallelism	1	2	4	8	16	21
avg (ms)	228383.40	119669.30	65031.35	36786.20	**24784.60**	27000.85
sd (s)	1721.24	975.84	525.27	273.85	270.51	556.52
cv (%)	0.75	0.81	0.81	0.74	1.09	2.06
avg Scal.	1.00	1.91	3.51	6.21	9.21	8.46
Spark Stream Stock Pricing						
Parallelism	1	2	4	8	16	21
avg (ms)	94400.00	74059.67	50886.67	52929.17	**42216.21**	85479.30
sd (s)	8961.03	20207.12	10037.38	5664.81	16424.54	77601.40
cv (%)	9.49	27.28	19.72	10.70	38.90	90.78
avg Scal.	1	1.27	1.85	1.78	2.24	1.10
PiCo Stream Stock Pricing						
Parallelism	1	2	4	8	16	21
avg (ms)	109273.35	54789.40	27532.85	13914.29	**7348.58**	7536.46
sd (s)	54.96	49.16	147.78	31.68	74.67	45.89
cv (%)	0.05	0.09	0.54	0.23	1.02	0.61
avg Scal.	1.00	1.99	3.97	7.85	14.87	14.50

TABLE 8.8: Average, standard deviation and coefficient of variation on 20 runs of the stream Stock Pricing benchmark. Best execution times are highlighted.

We again measured CPU and memory utilization using the *sar* tool. We executed 10 runs for each configuration: results obtained show a low variability, so that we do not report average and variance of collected results. Table 8.9 shows CPU utilization percentages only for average best execution times reported in table 8.8.

Stream Stock Pricing (execution time in ms)				
	Best exec. time	**Parallelism**	**CPU (%)**	**RAM (MB)**
Flink	24784.60	16	14.31%	4875.88
Spark	42216.21	16	10.23%	3169.32
PiCo	7348.58	16	38.85%	314.57

TABLE 8.9: User's percentage usage of all CPUs and RAM used in MB, referred to best execution times.

Flink and Spark show a memory utilization of 4 and 3 GB respectively, that is, one order of magnitude greater than PiCo, using only 314 MB and outperforming other tools in resource utilization.

Throughput Values for 10M Stock Options			
	Best exec. time (s)	**Parallelism**	**Stocks per Second**
Flink	24.78	16	403476.35
Spark	42.22	16	236875.81
PiCo	7.35	16	1360806.94

TABLE 8.10: Stream Stock Pricing: Throughput values computed as the number of input stock options with respect to the best execution time.

Finally, we provide throughput values computed with respect to the number of stock options in the input stream in the best execution time scenario reported in Table 8.8. We remark that the comparison with Spark is not completely fair since windowing is not performed in a count-based fashion. Table 8.10 shows that PiCo

processes more than 1.3M stock options per second, outperforming Flink and Spark, which processes about 400K and 200K stock options per second respectively.

8.3 Summary

In this Chapter we provided a set of experiments based on examples defined in Sect. 5.4, comprehending both batch and stream applications. We compared PiCo to Flink and Spark, focusing on expressiveness of the programming model and on performances in shared memory. The current experiments are run on shared memory only. By comparing execution times in both batch and stream applications, we reached the best execution time when comparing to state-of-the-art frameworks Spark and Flink. Nevertheless, results showed high dynamic allocation contention in input generation nodes, which limits PiCo scalability. An extension of PiCo using the FastFlow allocator might be used by any number of threads to dynamically allocate/deallocate memory. We also measured RAM and CPU utilization with the *sar* tool, which confirmed a lower memory consumption by PiCo with respect to the other frameworks when compared on batch application (Word Count and Stock Pricing) and stream application (Stock Pricing streaming): these results rely on the stability of a lightweight C++ runtime, in contrast to Java. What we reported in this Chapter is a preliminary experimental phase. We re working on providing more relevant benchmark, such as the Sort/Terasort, and support for HDFS is needed for such benchmarks.

Chapter 9

Conclusions

In this thesis, we presented PiCo, a new C++ DSL for data analytics pipelines. We started by studying and analyzing a large number of Big Data analytics tools, among which we identified the most representative ones: Spark [131], Storm [97], Flink [67] and Google Dataflow [5]. By analyzing in depth these frameworks, we identified the Dataflow model as the common model that better describes all levels of abstraction, from the user-level API to the execution model. Being all realized under the same common idea, we showed how various Big Data analytics tools share almost the same base concepts, differing mostly in their implementation choices. We then instantiated the Dataflow model into a stack of layers where each layer represents a dataflow graph/model with a different meaning, describing a program from what the programmer sees down to the underlying, lower-level, execution model layer (Chapter 4).

This study led to the specification and formalization of the minimum kernel of operations needed to create a pipeline for data analytics. As one of the strength of PiCo, we implemented a framework in which the data model is also hidden to the programmer, thus allowing the possibility to create a model that is *polymorphic* with respect to data model and processing model (i.e., stream or batch processing). This make it possible to 1) re-use the same algorithms and pipelines on different data models (e.g., stream, lists, sets, etc.); 2) reuse the same operators in different contexts, and 3) update operators without affecting the calling context. These aspects are fundamental for PiCo, since they differentiate it from all other frameworks exposing different data types to be used in the same application, forcing the user to re-think the whole application when moving from one operation to another. Furthermore, another goal reached, which is missing in other frameworks which usually provide the API description, is the one of formally defining the syntax of a program based on Pipelines and operators, hiding the data structures produced and generated by the program as well as providing a semantic interpretation that maps any PiCo program to a functional Dataflow graph — graph that represents the transformation flow followed by the processed collections (Chapter 5). This is in complete contrast with the unclear approach used by implementors of commercial-oriented Big Data analytics tools.

The formalization step concludes by showing how a PiCo program is compiled into a graph of parallel processing nodes. The compilation step takes as input the direct acyclic dataflow graph (DAG) resulting from a PiCo program (the Semantic DAG) and transforms it, using a set of rules, into a graph that we call the Parallel Execution (PE) Graph, representing a possible parallelization of the Semantic DAG. We provided this compilation step in a way that is abstract with respect to any actual implementation. For instance, it may be implemented in shared memory or through a distributed runtime. Moreover, a compiled (and optimized) Dataflow graph may be directly mapped to an actual network of computing units (e.g., communicating threads or processes) or executed by a macro-Dataflow interpreter (Chapter 6).

We succeed to implement our model into a C++14 compliant DSL whose aim is to focus on ease of programming with a clear and simple API, exposing to the user

a set of operator objects composed into a Pipeline object, processing bounded or unbounded data. This API was designed to exhibit a functional style over C++14 standard by defining a library of purely functional data transformation operators exhibiting 1) a well-defined functional and parallel semantics because of our formalization and 2) a fluent interface based on method chaining to improve code writing and readability (Chapter 7). Moreover, our API is data-model agnostic, that is, a PiCo program can address both batch and stream processing with a unique API and a unique runtime simply by having the user specify the data source as the first node of the Pipeline in a PiCo application. The type system described and implemented will check if the Pipeline built is compliant with respect to the kind of data being processed.

The current version of PiCo is built to run on shared memory only, but it will be possible to easily exploit distributed memory platforms thanks to its runtime level implemented on top of FastFlow which already supports execution on distributed systems. Furthermore, we remark that choosing FastFlow as the runtime for PiCo gives us almost for free the capability to realize a distributed memory implementation by still maintaining the very same implementation, only by changing communication among processes. To achieve this goal, we will provide an implementation of the FastFlow runtime for distributed execution by mean of a Global Asynchronous Memory(GAM) system model. The GAM system consists in a network of executors (i.e., FastFlow workers as well as PiCo operators) accessing a global dynamic memory with weak sharing semantics, allowing operators to communicate to each other in predefined communicators where they can exchange C++ smart pointers.

It is also possible to envision exploiting an underlying memory model such as PGAS or DSM: since FastFlow moves pointers and not data in communication channels among nodes, the same approach can be used to avoid data movement in a distributed scenario with an ad hoc relaxed consistency model. Another goal reachable by PiCo is the easily offloading of kernels to external devices such as GPU, FPGA, etc. This is possible for two reasons: the first is because of the C++ language, which naturally targets libraries and API for heterogeneous programming. The second is within the FastFlow runtime and API, which provides support for easily offloading tasks to GPUs. This is obviously a strength in the choice of implementing PiCo in C++ instead of Java/Scala, which also provide libraries for GPU offloading that are basically wrapper or extension for OpenCL or CUDA, based on auto-generated low-level bindings, but no such library is yet officially recognized as a Java extension.

From the actual performance viewpoint, we aim to solve dynamic allocation contention problems we are facing in input generation nodes, as showed in Chapter 8, which limits PiCo scalability. As future work, we could provide PiCo with the FastFlow allocator: this is a specialized allocator that allocates only large chunks of memory, slicing them up into little chunks of the same size which can be reused once freed. The FastFlow allocator relies on `malloc` and `free`, which makes it unfeasible to be used with C++ Standard Containers and with modern C++ in general. An extension of the FastFlow allocator in this direction might be used by any number of threads to dynamically allocate/deallocate memory.

In PiCo, we rely on the stability of a lightweight C++ runtime, in contrast to Java. We measured RAM and CPU utilization with the *sar* tool, which confirmed a lower memory consumption by PiCo with respect to the other frameworks when compared on batch application (Word Count and Stock Pricing) and stream application (Stock Pricing streaming). As another future work, we will provide PiCo with fault tolerance capabilities for automatic restore in case of failures. Another improvement for PiCo implementation on distributed systems would be to exploit the very same runtime on PGAS or DSMs, in order to still be able to use FastFlow's characteristic of moving pointers instead of data, thus allowing a high portability

at the cost of just managing communication among actors in a different memory model, which is left to the runtime.

In the experiments chapter, we showed that we achieved the goal of having a lightweight runtime able to better exploit resources (outperforming in memory utilization), obtaining a better execution time on benchmarks we tested with respect to Flink and Spark, which are two the most used tools nowadays. PiCo obtained good results even though it is still in a prototype phase, ensuring that it will be possible for us to still improve performances by providing special allocators to reduce dynamic memory allocation contentions, one the current performance issue in PiCo (Chapter 8).

With this thesis we aimed at bringing some order to the confused world of Big Data analytics, and we hope the readers will agree that we have reached our goal—at least in part. We believe that PiCo makes a step forward by giving C++ a chance of entering into a world almost completely Java-dominated—often considered more user-friendly. Starting from our preliminary work, we can envision a complete C++ framework that will provide all expected features of a fully equipped environment and that can be easily used by the data analytics scientists community.

Appendix A

Source Code

In this Appendix we provide source code snapshots reporting classes related to the core operations of each application.

PiCo

Stock Pricing

```
Pipe stockPricing((ReadFromFile()));
stockPricing
  .to(blackScholes).add(PReduce<StockAndPrice>([]
  (StockAndPrice p1, StockAndPrice p2){
    return std::max(p1,p2);}))
  .add(WriteToDisk<StockAndPrice>([](StockAndPrice kv){
    return kv.to_string();
  }));

/* execute the pipeline */
stockPricing.run();
```

Listing A.1: Batch Stock Pricing C++ pipeline in PiCo.

```
size_t window_size = 8;
Pipe stockPricing(ReadFromSocket('\n'));
stockPricing
  .to(varianceMap).add(PReduce<StockAndPrice>([]
    (StockAndPrice p1, StockAndPrice p2) {
      return std::max(p1,p2);
    })
  .window(window_size))
  .add(WriteToStdOut<StockAndPrice>([](StockAndPrice kv)
    {return kv.to_string();}
  ));

/* execute the pipeline */
stockPricing.run();

```

Listing A.2: Stream Stock Pricing C++ pipeline in PiCo.

Flink

Word Count

```
public class WordCount {
  // set up the execution environment
  final ExecutionEnvironment env =
  ExecutionEnvironment.getExecutionEnvironment();
  // get input data
  DataSet<String> text = env.readTextFile(params.get("input"));

  DataSet<Tuple2<String, Integer>> counts =
  // split up the lines in pairs (2-tuples) containing: (word,1)
  text.flatMap(new Tokenizer())
  // group by the tuple field "0" and sum up tuple field "1"
  .groupBy(0)
  .sum(1);

  // emit result
  if (params.has("output")) {
    counts.writeAsCsv(params.get("output"), "\n", " ");
    env.execute("WordCount Example");
  } else {
    System.out.println("Printing result to stdout. Use --output to specify
     output path.");
    counts.print();
  }

  /**
  * Implements the string tokenizer that splits sentences into words as a user-
     defined
  * FlatMapFunction. The function takes a line (String) and splits it into
  * multiple pairs in the form of "(word,1)" ({@code Tuple2<String, Integer>}).
  */
  public static final class Tokenizer
  implements FlatMapFunction<String, Tuple2<String, Integer>> {
    @Override
    public void flatMap(String value, Collector<Tuple2<String, Integer>> out)
    {
      // normalize and split the line
      String[] tokens = value.toLowerCase().split("\\W+");
      // emit the pairs
      for (String token : tokens) {
        if (token.length() > 0) {
          out.collect(new Tuple2<String, Integer>(token, 1));
        }
      }
    }
  }
}
```

Listing A.3: Word Count Java class in Flink.

Stock Pricing

```
public class StockPricing {
  // set up the execution environment
  final ExecutionEnvironment env =
    ExecutionEnvironment.getExecutionEnvironment();
  DataSet<String> text = env.readTextFile(params.get("input"));
  DataSet<Tuple2<String, Double>> max_prices =
    // parse the lines in pairs containing: (stock,option)
    text.map(new OptionParser())
    // compute the price of each stock option
    .map(new BlackScholes())
    // group by the tuple field "0" and extracts max on tuple field "1"
    .groupBy(0).max(1);

  // emit result
  if (params.has("output")) {
    max_prices.writeAsCsv(params.get("output"), "\n", " ");
    env.execute("StockPricing Example");
  } else {
    System.out.println("Printing result to stdout. Use --output to specify
     output path.");
    max_prices.print();
  }
}
```

Listing A.4: Batch Stock Pricing Java class in Flink.

```
// get the execution environment
final StreamExecutionEnvironment env = StreamExecutionEnvironment.
     getExecutionEnvironment();
env.setBufferTimeout(20);

// get input data by connecting to the socket
DataStream<String> text = env.socketTextStream("localhost", port, "\n");

DataStream<Tuple2<String, Double>> max_prices =
  // parse the lines in pairs containing: (stock,option)
  text.map(new OptionParser())
  // compute the price of each stock option
  .map(new varianceMap())
  // group by the tuple field "0" and extracts max on
  // tuple field "1"
  .keyBy(0).countWindow(8).max(1);

// print the results with a single thread, rather than in parallel
max_prices.print().setParallelism(1);
```

Listing A.5: Stream Stock Pricing Java class in Flink.

Spark

Word Count

```
public final class WordCount {
  JavaRDD<String> lines = spark.read().textFile(args[0]).javaRDD();
  JavaPairRDD<String, Integer> words =
    lines.flatMapToPair(new PairFlatMapFunction<String, String, Integer>() {
      public Iterator<Tuple2<String, Integer>> call(String s) {
        List<Tuple2<String, Integer>> tokens =
          new ArrayList<Tuple2<String, Integer>>();
        for (String t : SPACE.split(s)) {
          tokens.add(new Tuple2<String, Integer>(t, 1));
        }
        return tokens.iterator();
      }
    });

JavaPairRDD<String, Integer> counts =
  words.reduceByKey(new Function2<Integer, Integer, Integer>() {
    public Integer call(Integer i1, Integer i2) {
      return i1 + i2;
    }});

  List<Tuple2<String, Integer>> output = counts.collect();
  for (Tuple2<?, ?> tuple : output) {
    System.out.println(tuple._1() + ": " + tuple._2());
  }
}
```

Listing A.6: Word Count Java class in Spark.

Stock Pricing

```
JavaRDD<String> lines = spark.read().textFile(args[0]).javaRDD();

JavaPairRDD<String, OptionData> stock_options =
  lines.mapToPair(new PairFunction<String, String, OptionData>() {
    public Tuple2<String, OptionData> call(String value) {
      // tokenize the line
      String[] tokens = value.split("[\t ]");
      int i = 0;
      // parse stock name
      String name = tokens[i++];
      // parse option data
      OptionData opt = new OptionData();
      // parsing options..
      return new Tuple2<String, OptionData>(name, opt);
    }
  });

JavaPairRDD<String, Double> stock_prices=stock_options.mapToPair(new
    BlackScholes());

JavaPairRDD<String, Double> counts =
  stock_prices.reduceByKey( new Function2<Double, Double, Double>() {
    public Double call(Double i1, Double i2) {
      return i1 + i2;
    }
  });

List<Tuple2<String, Double>> output = counts.collect();
for (Tuple2<?,?> tuple : output) {
  System.out.println(tuple._1() + ": " + tuple._2());
}

```

Listing A.7: Batch Stock Pricing Java class in Spark.

```
// Create a DStream that will connect to hostname:port
JavaReceiverInputDStream<String> lines = jssc.socketTextStream("localhost",
    4000, StorageLevels.MEMORY_AND_DISK_SER);


@SuppressWarnings("serial")
JavaPairDStream<String, OptionData> stock_options = lines //
  .mapToPair(new PairFunction<String, String, OptionData>() {
    public Tuple2<String, OptionData> call(String value) {
    // tokenize the line
    String[] tokens = value.split("[\t ]");
    int i = 0;

    // parse stock name
    String name = tokens[i++];

    OptionData opt = new OptionData();
    // parse option data ...
    return new Tuple2<String, OptionData>(name, opt);
  }
});

JavaPairDStream<String, Double> stock_prices = stock_options //
  .mapToPair(new varianceMap());

JavaPairDStream<String, Double> counts = stock_prices.reduceByKey(new Function2<
    Double, Double, Double>() {
  public Double call(Double i1, Double i2) {
    return Math.max(i1, i2);
  }
});
```

Listing A.8: Stream Stock Pricing Java class in Spark.

Bibliography

[1] IEEE standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008.

[2] M. Abadi et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. http://tensorflow.org/, 2015.

[3] M. Abadi and M. Isard. Timely dataflow: A model. In *FORTE*, pages 131–145, 2015.

[4] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, Dec. 1996.

[5] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernàndez-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.*, 8(12):1792–1803, Aug. 2015.

[6] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernàndez-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and 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*, 8:1792–1803, 2015.

[7] M. Aldinucci. eskimo: experimenting with skeletons in the shared address model. *Parallel Processing Letters*, 13(3):449–460, Sept. 2003.

[8] M. Aldinucci, S. Bagnasco, S. Lusso, P. Pasteris, and S. Rabellino. The Open Computing Cluster for Advanced data Manipulation (occam). In *The 22nd International Conference on Computing in High Energy and Nuclear Physics (CHEP)*, San Francisco, USA, Oct. 2016.

[9] M. Aldinucci, S. Campa, P. Ciullo, M. Coppola, S. Magini, P. Pesciullesi, L. Potiti, R. Ravazzolo, M. Torquati, M. Vanneschi, and C. Zoccolo. The implementation of ASSIST, an environment for parallel and distributed programming. In H. Kosch, L. Böszörményi, and H. Hellwagner, editors, *Proc. of 9th Intl Euro-Par 2003 Parallel Processing*, volume 2790 of *LNCS*, pages 712–721, Klagenfurt, Austria, Aug. 2003. Springer.

[10] M. Aldinucci, M. Coppola, M. Danelutto, M. Vanneschi, and C. Zoccolo. ASSIST as a research framework for high-performance grid programming environments. In J. C. Cunha and O. F. Rana, editors, *Grid Computing: Software environments and Tools*, chapter 10, pages 230–256. Springer, Jan. 2006.

[11] M. Aldinucci and M. Danelutto. Skeleton based parallel programming: functional and parallel semantic in a single shot. *Computer Languages, Systems and Structures*, 33(3-4):179–192, Oct. 2007.

[12] M. Aldinucci, M. Danelutto, M. Drocco, P. Kilpatrick, **Claudia Misale**, G. Peretti Pezzi, and M. Torquati. A parallel pattern for iterative stencil + reduce. *Journal of Supercomputing*, pages 1–16, 2016.

[13] M. Aldinucci, M. Danelutto, and P. Kilpatrick. Towards hierarchical management of autonomic components: a case study. In D. E. Baz, T. Gross, and

F. Spies, editors, *Proc. of Intl. Euromicro PDP 2009: Parallel Distributed and network-based Processing*, pages 3–10, Weimar, Germany, Feb. 2009. IEEE.

[14] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati. An efficient unbounded lock-free queue for multi-core systems. In *Proc. of 18th Intl. Euro-Par 2012 Parallel Processing*, volume 7484 of *LNCS*, pages 662–673, Rhodes Island, Greece, Aug. 2012. Springer.

[15] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati. Fastflow: high-level and efficient streaming on multi-core. In S. Pllana and F. Xhafa, editors, *Programming Multi-core and Many-core Computing Systems*, Parallel and Distributed Computing, chapter 13. Wiley, Oct. 2014.

[16] M. Aldinucci, M. Danelutto, and P. Teti. An advanced environment supporting structured parallel programming in Java. *Future Generation Computer Systems*, 19(5):611–626, July 2003.

[17] M. Aldinucci, M. Drocco, G. Peretti Pezzi, **Claudia Misale**, F. Tordini, and M. Torquati. Exercising high-level parallel programming on streams: a systems biology use case. In *Proc. of the 2014 IEEE 34th Intl. Conference on Distributed Computing Systems Workshops (ICDCS)*, Madrid, Spain, 2014. IEEE.

[18] M. Aldinucci, M. Meneghin, and M. Torquati. Efficient Smith-Waterman on multi-core with fastflow. In M. Danelutto, T. Gross, and J. Bourgeois, editors, *Proc. of Intl. Euromicro PDP 2010: Parallel Distributed and network-based Processing*, pages 195–199, Pisa, Italy, Feb. 2010. IEEE.

[19] M. Aldinucci and M. Torquati. *FastFlow website*, 2009. http://mc-fastflow.sourceforge.net/.

[20] M. Aldinucci, M. Torquati, C. Spampinato, M. Drocco, **Claudia Misale**, C. Calcagno, and M. Coppo. Parallel stochastic systems biology in the cloud. *Briefings in Bioinformatics*, 15(5):798–813, 2014.

[21] 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, and D. Warneke. The stratosphere platform for big data analytics. *The VLDB Journal*, 23(6):939–964, Dec. 2014.

[22] Apache Software Foundation. *Hadoop*, 2013. http://hadoop.apache.org/.

[23] Apache Software Foundation. *HDFS*, 2013. http://hadoop.apache.org/docs/r1.2.1/hdfs_user_guide.html.

[24] Apache Software Foundation. *Cassandra*, 2016. http://cassandra.apache.org.

[25] Apache Software Foundation. *HBase*, 2016. http://hbase.apache.org/.

[26] Apache Software Foundation. *Yarn*, 2016. https://hadoop.apache.org/docs/r2.7.1/hadoop-yarn/hadoop-yarn-site/YARN.html.

[27] Apache Software Foundation. *ZooKeeper*, 2016. http://zookeeper.apache.org/.

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

[29] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56–67, 2009.

[30] S. Bahrampour, N. Ramakrishnan, L. Schott, and M. Shah. Comparative study of caffe, neon, theano, and torch for deep learning. *CoRR*, abs/1511.06435, 2015.

[31] Basho. *Riak*, 2016. http://basho.com/riak/.

[32] M. A. Beyer and D. Laney. The importance of big data: A definition. Technical report, Stamford, CT: Gartner, June 2012.

[33] R. Bhardwaj, A. Sethi, and R. Nambiar. Big data in genomics: An overview. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 45–49, Oct 2014.

[34] T. Bingmann, M. Axtmann, E. Jöbstl, S. Lamm, H. C. Nguyen, A. Noe, S. Schlag, M. Stumpp, T. Sturm, and P. Sanders. Thrill: High-performance algorithmic distributed batch data processing with C++. *CoRR*, abs/1608.05634, 2016.

[35] H. Bischof, S. Gorlatch, and R. Leshchinskiy. DatTel: A data-parallel C++ template library. *Parallel Processing Letters*, 13(3):461–472, 2003.

[36] H.-J. Boehm. Threads cannot be implemented as a library. *SIGPLAN Not.*, 40(6):261–268, June 2005.

[37] H.-J. Boehm and S. V. Adve. Foundations of the c++ concurrency memory model. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 68–78, New York, NY, USA, 2008. ACM.

[38] J. Bonwick and S. Microsystems. The slab allocator: An object-caching kernel memory allocator. In *In USENIX Summer*, pages 87–98, 1994.

[39] E. A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, pages 7–, New York, NY, USA, 2000. ACM.

[40] D. R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[41] F. Cappello and D. Etiemble. MPI versus MPI+OpenMP on IBM SP for the NAS benchmarks. In *Proc. of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '00. IEEE Computer Society, 2000.

[42] P. Carbone, G. Fóra, S. Ewen, S. Haridi, and K. Tzoumas. Lightweight asynchronous snapshots for distributed dataflows. *CoRR*, abs/1506.08603, 2015.

[43] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. Henry, R. Bradshaw, and Nathan. FlumeJava: Easy, efficient data-parallel pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 363–375, 2 Penn Plaza, Suite 701 New York, NY 10121-0701, 2010.

[44] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, Feb. 1985.

[45] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou. Denovo: Rethinking the memory hierarchy for disciplined parallelism. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 155–166, Washington, DC, USA, 2011. IEEE Computer Society.

[46] P. Ciechanowicz, M. Poldner, and H. Kuchen. The Munster skeleton library Muesli — a comprehensive overview. In *ERCIS Working paper*, number 7. ERCIS – European Research Center for Information Systems, 2009.

[47] C. Cole and M. Herlihy. Snapshots and software transactional memory. *Sci. Comput. Program.*, 58(3):310–324, 2005.

[48] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1991.

[49] M. Cole. Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, 2004.

[50] M. Cole. *Skeletal Parallelism home page*, 2009. http://homepages.inf.ed.ac.uk/mic/Skeletons/.

[51] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, 2011.

[52] Crunch. Apache Crunch website. http://crunch.apache.org/.

[53] M. Danelutto, R. D. Meglio, S. Orlando, S. Pelagatti, and M. Vanneschi. A methodology for the development and the support of massively parallel programs. *Future Generation Compututer Systems*, 8(1-3):205–220, 1992.

[54] M. Danelutto and M. Stigliani. SKElib: parallel programming with skeletons in C. In A. Bode, T. Ludwing, W. Karl, and R. Wismüller, editors, *Proc. of 6th Intl. Euro-Par 2000 Parallel Processing*, volume 1900 of *LNCS*, pages 1175–1184, Munich, Germany, Aug. 2000. Springer.

[55] M. Danelutto and M. Torquati. Loop parallelism: a new skeleton perspective on data parallel patterns. In M. Aldinucci, D. D'Agostino, and P. Kilpatrick, editors, *Proc. of Intl. Euromicro PDP 2014: Parallel Distributed and network-based Processing*, Torino, Italy, 2014. IEEE.

[56] M. Danelutto and M. Torquati. Structured parallel programming with "core" fastflow. In V. Zsók, Z. Horváth, and L. Csató, editors, *Central European Functional Programming School*, volume 8606 of *LNCS*, pages 29–75. Springer, 2015.

[57] J. Darlington, A. J. Field, P. Harrison, P. H. J. Kelly, D. W. N. Sharp, R. L. While, and Q. Wu. Parallel programming using skeleton functions. In *Proc. of Parallel Architectures and Langauges Europe (PARLE'93)*, volume 694 of *LNCS*, pages 146–160, Munich, Germany, June 1993. Springer.

[58] J. Darlington, Y.-k. Guo, H. W. To, and J. Yang. Parallel skeletons for structured composition. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 19–28, New York, NY, USA, 1995. ACM.

[59] Databricks. Spark Tungsten website. https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html.

[60] T. De Matteis and G. Mencagli. Parallel patterns for window-based stateful operators on data streams: an algorithmic skeleton approach. *International Journal of Parallel Programming*, pages 1–20, 2016.

[61] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Usenix OSDI '04*, pages 137–150, Dec. 2004.

[62] D. del Rio Astorga, M. F. Dolz, L. M. Sánchez, J. G. Blas, and J. D. García. A C++ generic parallel pattern interface for stream processing. In *Algorithms and Architectures for Parallel Processing - 16th International Conference, ICA3PP 2016, Granada, Spain, December 14-16, 2016, Proceedings*, pages 74–87, 2016.

[63] M. Drocco, **Claudia Misale**, and M. Aldinucci. A cluster-as-accelerator approach for SPMD-free data parallelism. In *Proc. of Intl. Euromicro PDP 2016: Parallel Distributed and network-based Processing*, pages 350–353, Crete, Greece, 2016. IEEE.

[64] M. Drocco, **Claudia Misale**, G. Peretti Pezzi, F. Tordini, and M. Aldinucci. Memory-optimised parallel processing of Hi-C data. In *Proc. of Intl. Euromicro PDP 2015: Parallel Distributed and network-based Processing*, pages 1–8. IEEE, Mar. 2015.

[65] M. Drocco, **Misale, Claudia**, G. Tremblay, and M. Aldinucci. A formal semantics for data analytics pipelines. https://arxiv.org/abs/1705.01629, May 2017.

[66] J. Enmyren and C. W. Kessler. Skepu: A multi-backend skeleton programming library for multi-GPU systems. In *Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications*, HLPP '10, pages 5–14, New York, NY, USA, 2010. ACM.

[67] Flink. Apache Flink website. https://flink.apache.org/.

[68] Flink. Flink streaming examples, 2015. [Online; accessed 16-November-2016].

[69] M. J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966.

[70] M. Fowler. Kappa-Architecture website. https://www.martinfowler.com/bliki/FluentInterface.html.

[71] H. González-Vélez and M. Leyton. A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers. *Software: Practice and Experience*, 40(12):1135–1160, Nov. 2010.

[72] Google. *Google Cloud Dataflow*, 2015. https://cloud.google.com/dataflow/.

[73] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier, fifth edition, 2011.

[74] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[75] IBM. What is big data? http://www-01.ibm.com/software/data/bigdata/what-is-big-data.html, 2013.

[76] Intel. *Intel ® C++ Intrinsics Reference*, 2010.

[77] Intel. *Intel ® AVX-512 instructions*, 2013. https://software.intel.com/en-us/blogs/2013/avx-512-instructions.

[78] Intel Corp. *Threading Building Blocks*, 2011.

[79] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[80] J. Jeffers and J. Reinders. Front-matter. In J. Jeffers and J. Reinders, editors, *Intel Xeon Phi Coprocessor High Performance Programming*, pages i – iii. Morgan Kaufmann, Boston, 2013.

[81] D. R. Jefferson. Virtual time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, July 1985.

[82] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[83] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, 1974. North Holland, Amsterdam.

[84] Kappa-Architecture. Kappa-Architecture website. http://milinda.pathirage.org/kappa-architecture.com/.

[85] Khronos Compute Working Group. *OpenCL*, Nov. 2009. http://www.khronos.org/opencl/.

[86] M. Kiran, P. Murphy, I. Monga, J. Dugan, and S. S. Baveja. Lambda architecture for cost-effective batch and speed big data processing. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 2785–2792, Oct 2015.

[87] J. Kreps, N. Narkhede, and J. Rao. Kafka: A distributed messaging system for log processing. In *Proceedings of 6th International Workshop on Networking Meets Databases (NetDB), Athens, Greece*, 2011.

[88] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, Sept. 1979.

[89] L. Lamport. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst.*, 5(2):190–222, 1983.

[90] D. Laney. 3D data management: Controlling data volume, velocity, and variety. Technical report, META Group, February 2001.

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

[92] H. Li. Introduction to Big Data. http://haifengl.github.io/bigdata/, 2016.

[93] B. C. Libraries. Boost Serialization documentation webpage. http://www.boost.org/doc/libs/1_63_0/libs/serialization/doc/serialization.html.

[94] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[95] K. Matsuzaki, H. Iwasaki, K. Emoto, and Z. Hu. A library of constructive skeletons for sequential style of parallel programming. In *Proc. of the 1st Inter. conference on Scalable information systems*, InfoScale '06, New York, NY, USA, 2006. ACM.

[96] D. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, November 2013.

[97] M. A. U. Nasir, G. D. F. Morales, D. García-Soriano, N. Kourtellis, and M. Serafini. The power of both choices: Practical load balancing for distributed stream processing engines. *CoRR*, abs/1504.00788, 2015.

[98] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-tolerant software distributed shared memory. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 291–305, Berkeley, CA, USA, 2015. USENIX Association.

[99] A. Y. Ng, G. Bradski, C.-T. Chu, K. Olukotun, S. K. Kim, Y.-A. Lin, and Y. Yu. Map-reduce for machine learning on multicore. In *NIPS*, 12/2006 2006.

[100] B. Nicolae, C. H. A. Costa, **Claudia Misale**, K. Katrinis, and Y. Park. Leveraging adaptative I/O to optimize collective data shuffling patterns for big data analytics. *IEEE Transactions on Parallel and Distributed Systems*, PP(99), 2016.

[101] B. Nicolae, C. H. A. Costa, **Claudia Misale**, K. Katrinis, and Y. Park. Towards memory-optimized data shuffling patterns for big data analytics. In *IEEE/ACM 16th Intl. Symposium on Cluster, Cloud and Grid Computing, CCGrid 2016*, Cartagena, Colombia, 2016. IEEE.

[102] NVIDIA Corp. *CUDA website*, June 2013 (last accessed). http://www.nvidia.com/object/cuda_home_new.html.

[103] S. Oaks and H. Wong. *Java Threads*. Nutshell handbooks. O'Reilly Media, 2004.

[104] Oracle. *NoSQL*, 2016. http://nosql-database.org/.

[105] P. S. Pacheco. *Parallel programming with MPI*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[106] I. Park, M. J. Voss, S. W. Kim, and R. Eigenmann. Parallel programming environment for OpenMP. *Scientific Programming*, 9:143–161, 2001.

[107] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2007.

[108] M. Poldner and H. Kuchen. Scalable farms. In *Proc. of Intl. PARCO 2005: Parallel Computing*, Malaga, Spain, Sept. 2005.

[109] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

[110] J. H. Rutgers. Programming models for many-core architectures: a co-design approach. 2014.

[111] L. M. Sanchez, J. Fernandez, R. Sotomayor, and J. D. Garcia. A comparative evaluation of parallel programming models for shared-memory architectures. In *Proceedings of the 2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, ISPA '12, pages 363–370, Washington, DC, USA, 2012. IEEE Computer Society.

[112] A. Secco, I. Uddin, G. Peretti Pezzi, and M. Torquati. Message passing on infiniband RDMA for parallel run-time supports. In M. Aldinucci, D. D'Agostino, and P. Kilpatrick, editors, *Proc. of Intl. Euromicro PDP 2014: Parallel Distributed and network-based Processing*, Torino, Italy, 2014. IEEE.

[113] J. Serot. Tagged-token data-flow for skeletons. *Parallel Processing Letters*, 11(4):377–392, 2001.

[114] D. B. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Comput. Surv.*, 30(2):123–169, June 1998.

[115] D. J. Sorin, M. D. Hill, and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 1st edition, 2011.

[116] M. Steuwer and S. Gorlatch. Skelcl: Enhancing opencl for high-level programming of multi-GPU systems. In *Proceedings of the 12th International Conference on Parallel Computing Technologies*, pages 258–272, St. Petersburg, Russia, Oct. 2013.

[117] Storm. Apache Storm website. http://storm.apache.org/.

[118] **Claudia Misale**. Accelerating bowtie2 with a lock-less concurrency approach and memory affinity. In *Proc. of Intl. Euromicro PDP 2014: Parallel Distributed and network-based Processing*, Torino, Italy, 2014. IEEE. (Best paper award).

[119] **Claudia Misale**, M. Aldinucci, and M. Torquati. Memory affinity in multithreading: the bowtie2 case study. In *Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES) – Poster Abstracts*, Fiuggi, Italy, 2013. HiPEAC.

[120] **Claudia Misale**, M. Drocco, M. Aldinucci, and G. Tremblay. A comparison of big data frameworks on a layered dataflow model. In *Proc. of HLPP2016: Intl. Workshop on High-Level Parallel Programming*, pages 1–19, Muenster, Germany, July 2016. arXiv.org.

[121] **Claudia Misale**, M. Drocco, M. Aldinucci, and G. Tremblay. A comparison of big data frameworks on a layered dataflow model. *Parallel Processing Letters*, 27(01):1740003, 2017.

[122] **Claudia Misale**, G. Ferrero, M. Torquati, and M. Aldinucci. Sequence alignment tools: one parallel pattern to rule them all? *BioMed Research International*, 2014.

[123] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[124] F. Tordini, M. Drocco, **Claudia Misale**, L. Milanesi, P. Liò, I. Merelli, and M. Aldinucci. Parallel exploration of the nuclear chromosome conformation with NuChart-II. In *Proc. of Intl. Euromicro PDP 2015: Parallel Distributed and network-based Processing*. IEEE, Mar. 2015.

[125] F. Tordini, M. Drocco, **Claudia Misale**, L. Milanesi, P. Liò, I. Merelli, M. Torquati, and M. Aldinucci. NuChart-II: the road to a fast and scalable tool for Hi-C data analysis. *International Journal of High Performance Computing Applications (IJHPCA)*, 2016.

[126] 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, and D. Ryaboy. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 147–156, New York, NY, USA, 2014. ACM.

[127] UniTo-INFN. Occam Supercomputer website. http://c3s.unito.it/index.php/super-computer.

[128] L. G. Valiant. A bridging model for parallel computation. *CACM*, 33(8):103–111, Aug. 1990.

[129] O. Villa, V. Gurumoorthi, A. Márquez, and S. Krishnamoorthy. Effects of floating-point non-associativity on numerical computations on massively multithreaded systems. In *In Cray User Group meeting*, CUG'09, 2009.

[130] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[131] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and 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'12, Berkeley, CA, USA, 2012. USENIX.

[132] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[133] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proc. of the 24th ACM Symposium on Operating Systems Principles*, SOSP, pages 423–438, New York, NY, USA, 2013. ACM.

[134] ZeroMQ. *website*, 2012. http://www.zeromq.org/.