
POLITECNICO DI MILANO

School of Industrial and Information Engineering
Department of Electronics, Information and Bioengineering

Master of Science in Computer Science and Engineering

---

# Noir: Design, Implementation and Evaluation of a Streaming and Batch Processing Framework

---

*Supervisor:*
Prof. Alessandro Margara

*Co-Supervisor:*
Prof. Gianpaolo Cugola

*Master Thesis by:*
Marco Donadoni, 945780
Edoardo Morassutto, 940106

Academic Year 2020–2021

# *Abstract*

Nowadays, datasets have become so huge that it is impossible to analyze them using only the resources of a single computer. To be able to process them in a timely manner, computations need to be distributed on clusters of multiple machines.

Unfortunately, programming distributed software systems is very difficult. Since the advent of distributed computing, researchers and practitioners have been striving to devise abstractions that are flexible and easy to use, yet efficient and scalable. On one extreme, the option is to implement ad-hoc solutions for each processing task, exploiting low-level facilities to handle the communication and the coordination between the different machines. This approach is very flexible, and it is able to extract as much performance as possible from the available resources. However, the development of these custom solutions is usually time-consuming, and the resulting code can be very complex and hard to maintain.

To address these drawbacks, many data processing frameworks, such as Apache Spark and Apache Flink, were developed in the recent years. These systems automatically handle the parallelization of the computations, providing to the user a rich set of features that can be used to easily implement processing pipelines. These frameworks, however, are not able to provide performance on par with that of ad-hoc solutions.

This thesis presents Noir, a novel stream-processing framework implemented in Rust. Its objective is to fill the gap existing between ad-hoc solutions and distributed processing frameworks, providing better performance than the latter while maintaining their simplicity and ease of use. Even providing similar expressiveness to Apache Flink, our evaluation shows that Noir is able to achieve up to 30× its throughput, and it rivals custom MPI solutions in some workloads.

# *Sommario*

Al giorno d'oggi, i dataset sono così immensi che è impossibile analizzarli usando solo le risorse di un singolo computer. Per poterli elaborare in modo tempestivo, è necessario distribuire la computazione su un cluster di più macchine.

Purtroppo, programmare correttamente un sistema distribuito è estremamente difficile. Dall'avvento del *distributed computing*, ricercatori e sviluppatori si sono impegnati per ideare astrazioni flessibili e facili da usare, ma che siano allo stesso tempo anche efficienti e scalabili. Da un lato, l'opzione è quella di implementare soluzioni ad-hoc per ciascun task, sfruttando interfacce di basso livello per gestire la comunicazione e il coordinamento tra le diverse macchine. Questo approccio è molto flessibile ed è in grado di ottenere le massime prestazioni possibili dalle risorse disponibili. Tuttavia, lo sviluppo di queste soluzioni personalizzate richiede solitamente molto tempo e il codice risultante può essere molto complesso e difficile da mantenere.

Per sopperire a questi inconvenienti, negli ultimi anni sono stati sviluppati molti framework di elaborazione dati, come Apache Spark e Apache Flink. Questi sistemi gestiscono automaticamente la parallelizzazione, fornendo all'utente un ricco insieme di funzionalità che possono essere utilizzate per implementare facilmente le pipeline di elaborazione. Questi framework, però, non sono in grado di fornire prestazioni alla pari di soluzioni ad-hoc.

Questa tesi presenta Noir, un nuovo framework per lo stream-processing implementato in Rust. Il suo obiettivo è quello di colmare il divario esistente tra soluzioni ad-hoc e framework di elaborazione distribuita, fornendo prestazioni migliori di questi ultimi pur mantenendo la loro semplicità e facilità d'uso. Anche fornendo un'espressività simile ad Apache Flink, la nostra analisi mostra che Noir è in grado di raggiungere fino a 30× il suo throughput, riuscendo a competere con MPI in alcuni casi.

# *Acknowledgements*

First, I would like to thank Professor Alessandro Margara and Professor Gianpaolo Cugola for their guidance and the advices they gave us during our work on this thesis. Many thanks also go to Alessio Fino, who helped us in the beginning of our project, and to Lorenzo Romanò, who used Noir for his thesis work and provided us with valuable feedback.

Of course, I have to thank my dear friend Edoardo, whose hard work made it possible to reach this incredible accomplishment. It has been a pleasure working with you. You are one of the many exceptional people I have met while participating in the Italian Olympiads in Informatics, which has been a life changing experience.

I would like to thank all of my friends, both the ones I met a long time ago and those I have made during my stay at Politecnico di Milano. I am deeply grateful for all the beautiful moments we spend together.

Last but not least, all of this would have been impossible without the love and support of my family: my parents, Fabio and Nicoletta, and my sister Chiara. Thank you for all the encouragement you have given me during my studies.

*– Marco*

First, I would like to thank Professor Alessandro Margara and Professor Gianpaolo Cugola for their invaluable help and support in working on this thesis. Working with them has been a great time, full of inspiration, encouragement and good laughs.

Then, I would like to thank Marco, without whom this experience would not have been the same. We spent many hours working together, and I am so lucky to have been able to work with him.

Thanks also to my parents, Monica and Manuele, and to my brother Elia, who have been my support and inspiration during all these years.

As an offsite student, I would like to thank all the friends from my hometown, they have been really close to me even though I was far away from them. Thanks also to all the friends I made in Milan, they have been fundamental during these years far from home.

From our working group, I would like to thank Alessio Fino who worked on RStream, and Lorenzo Romanò who tried and helped testing Noir.

Finally, I would like to thank the Italian Olympiads in Informatics and all the wonderful people who worked with me, who always believed in me and tirelessly motivated me to do my best.

*– Edoardo*

# Contents

# 1 Introduction

In today's world, technology usage is widespread and is shaping our professional and personal lives. An immense amount of data is being produced every day, at a rate that keeps increasing year after year. In the last few decades, the term *Big Data* has been widely used to represent datasets that are so big that they become very difficult to manage, store and analyze. They are characterized by the so-called "3Vs": volume, variety and velocity. *Volume* clearly refers to the huge size of the dataset: in many cases this means that the dataset cannot be stored in a single machine, but more complex systems are needed to reach the necessary storage capacity and performance. *Variety* refers instead to the fact that these datasets can contain very different types of data coming from different sources, ranging from structured records to videos, images and tweets. Finally, *velocity* refers to both the very fast rate of data generation, but also to the need of being able to process and analyze these datasets in a timely manner, for example when dealing with strict temporal constraints.

In any case, datasets are usually not useful on their own, but they need to be processed in order to extract knowledge. For example, while data is a very important asset for companies, it is useful only if it can be analyzed to extract the necessary strategic information needed to make business decisions. Processing of huge amounts of data is also needed when considering research experiments in many scientific areas, like High Energy Physics (HEP) [2]. Just like big datasets are difficult to store, they are for the same reasons difficult to process. Even when considering the latest technology improvements, the processing power provided by a single machine is not nearly enough to handle such workloads, let alone executing them in an appropriate amount of time. The only solution is to use a cluster of interconnected machines, moving from a shared-memory model to a distributed-memory model. This poses new challenges for these processing pipelines, making their development much more difficult and time-consuming.

The first and oldest way to develop these kinds of distributed workflows is to design ad-hoc solutions, by manually handling the communication, synchronization and data sharing between the various machines. This can be facilitated by the use of libraries that provide some abstractions over the low-level facilities provided by the operating system. One of the most successful is MPI (Message Passing Interface), which represents the de facto standard protocol used for distributed high performance computing. It provides many facilities that can be used to communicate between different processes, including both point-to-point and collective communication, such as the broadcast, scatter and gather primitives. It can also be used to

synchronize the various processes, either implicitly by sending and receiving special messages, or explicitly using barriers.

This approach provides high flexibility in the implementation of the distributed computation, which also makes it possible to achieve very high performance. On the flip side, developing these kinds of systems is a very hard task that takes a lot of time and is very error-prone. The developers have the burden to decide how to parallelize and distribute the computation, in which way the various processes communicate, but they also need to make sure that these computations are correctly synchronized. This results in much more complex code, which can quickly become difficult to maintain, debug and tune to achieve optimal performance.

To overcome these limitations many distributed processing frameworks were developed in the last years, such as Apache Spark and Apache Flink. These systems provide a high-level interface that makes the development of distributed and parallel computations much easier. Workflows are defined as a series of transformations on a given dataset, abstracting away all the challenges of running them in a distributed environment. In particular, the cooperation and interactions of the various processes are automatically handled. Thus, programmers are free to focus only on implementing the logic of the application without having to deal with the communication and the synchronization between the various machines. These frameworks also provide bindings to many popular languages like Java, Python and Scala, which make their adoption even more convenient. Even though these systems provide many advantages, in many cases they are not able to provide the same performance level that can be achieved with ad-hoc solutions developed using lower-level primitives.

Given these considerations we propose Noir, a new distributed processing system written in Rust [32] whose objective is to fill the performance gap between the development of custom solutions and the use of already existing frameworks, while keeping their ease of use and expressiveness. Noir is influenced by both Apache Flink and MPI, and it can be considered the successor to RStream [6], which tried to achieve this same goal. However, Noir differs from all of them in many key areas. With respect to Flink, it is able to achieve much better performance while still providing a high-level interface which supports many of Flink's features. On the other hand, Noir is much more expressive than both MPI and RStream, as the former only provides communication primitives and the latter is very limited in the transformations that can be made to the dataset.

Even providing a very high-level interface, Noir offers much better performance than Flink, achieving up to 30× its throughput. Noir is consistently faster than Flink in all the benchmarks we have tried, always ending at, or pretty close to, the lead.

Chapter 2 will introduce the main concepts of the *dataflow model*, the main abstraction Noir is based on.

Chapter 3 will present some of the most popular frameworks that have been developed for distributed processing, outlining their strengths and weaknesses.

Chapter 4 will discuss the architecture of Noir, presenting the internal structure and its components.

Chapter 5 will describe the API of Noir, which can be used to define distributed computations.

Chapter 6 will analyze the performance of Noir, presenting the results of the benchmarks and a comparison between Noir and the other frameworks.

Chapter 7 will draw the conclusions and suggest some future work.

# Dataflow Paradigm 2

In this chapter we introduce the *dataflow model* [1], an abstraction in which computations are described as directed graphs, where each node is an operator and data flows on the edges between them. This paradigm is the foundation on which Noir is built: it is inherently parallel, and it can handle bounded and unbounded datasets while providing both high throughput and low latency.

## 2.1 Introduction

Imperative programming is the usual way in which programs are written. In this model, programs are defined as a sequence of statements executed in a specified order. Since each of the statements can modify the state of the program, it is fundamental that they are executed in the correct order. This is one of the reasons why this paradigm is not very well suited to describe parallel programs that need to be executed in distributed environments, as most of the time statements cannot be reordered and executed in parallel. Another factor to take into consideration is that, when using multiple threads of execution running concurrently, synchronization mechanisms are needed to make sure that there are no conflicts when accessing the shared state of the program. This further limits the parallelization potential of the program.

The dataflow paradigm, instead, describes the program in terms of operators and focuses its attention on how data is exchanged between them. These operators are arranged in a directed graph and data elements, sometimes called tuples, flow on the connections between them, starting from a source operator and ending in a sink one. Unlike imperative programming, in the dataflow paradigm there is not a single global state of the program, but each operator has its own independent local one. This is very important, because it makes it possible to execute each operator in parallel and independently of the others. This abstraction is also able to handle unbounded streams of data, since each operator is just a black box that receives some data, makes some computation and generates new data to be sent to the operators downstream in the graph. It is also possible to achieve very low end-to-end latency, given that computations are done in a *streaming* fashion. This approach is different from the so called *micro-batch processing*, which divides the unbounded stream in very small batches of tuples that are then scheduled and processed as if they were bounded datasets. Note that the batches are very small so that the latency of the computation is not affected much by the waiting needed to fill the batch.

## 2.2 Sources, Operators and Sinks

Tuples are generated and injected into the system by special operators called sources, which are operators that do not have any incoming connections. From the abstraction's point of view, it is not important from where sources read the data elements; for example, they can read them from the network, from a local file, from an external system like a Kafka topic and so on.

Sinks, instead, are the exact opposite of sources, since they are operators which do not have any outgoing connections. As for sources, it is not important what sinks do with the tuples they receive: they can be written to a disk, sent over the network, displayed on the screen and so on.

In general, operators receive a stream of tuples from upstream operators, use them to make some calculations and generate new data elements that are sent to downstream operators. In other words, operators can be considered as transformations that map a stream of elements into another one. Operators are *stateless* if they consider one element at a time, and the calculation they make on each element does not depend on elements of the stream received previously. However, some operators need to keep a local state, so they are called *stateful*. This is the case, for example, of operators that need to reduce the stream of elements into a single tuple. In addition, operators are called *blocking* if they need to consume the whole input stream of data before being able to generate the elements of the output stream. Note that blocking operators are always stateful, but the opposite is not always true.

## 2.3 Partitioning

When using operators that aggregate elements into a single value, it is sometimes useful to partition the stream into multiple substreams by dividing its elements into multiple groups. This means that, instead of reducing the whole stream into a single value, each partition is aggregated independently of the others. To achieve this, we can use a special grouping operator that divides the tuples based on the value of a *key*. In particular, the user can specify a way to obtain a key from each data tuple, and tuples having the same key are grouped together.

After performing a grouping operation, each downstream operator can consider each stream partition independently of the others. For our purposes, this is particularly interesting because it can be exploited to extract more parallelism. Since each substream divides the operators' work into multiple independent subtasks, they can be executed in parallel without any synchronization. This is also valid for operators that, when considering non-grouped streams, would be difficult to parallelize.

## 2.4 Timestamps

Given the ability to handle unbounded data streams while maintaining low latency, the dataflow paradigm is ideal for event processing, which is the processing of real-time events to extract complex information. In this case, each element of the stream is composed of some data that describes the event, together with a timestamp that refers to when the event occurred.

There are two time domains that we are generally interested in: *event time* and *processing time*. When considering event time, the timestamp associated with each event is the time at which the event actually occurred, and it is assigned by the external system that generated the event. In the case of processing time, instead, the timestamp refers to the time at which the event is observed during the computation, based on the local system time. Given a single event, the main difference between the two time domains is that event time is assigned once and never changes, while processing time varies every time the data flows through the pipeline.

## 2.5 Watermarks

When dealing with tuples timestamped in the event-time domain, sources are not required to generate them with timestamps that are in the correct order, for example to deal with *late events*. Moreover, network latencies can cause the clocks on the various hosts to be out of sync, causing the relative order of the timestamps to be incorrect. Therefore, the system needs to put in place a mechanism to deal with unordered timestamps, because processing time and event time advance independently, and they can be skewed. There is not a unique way to track the progress of time, see for example Timely Dataflow in Section 3.2.3. In any case, the mechanism we are interested in is that of *watermarks*.

Watermarks are periodic markers that flow along the pipeline, together with data tuples, and each of them has an associated timestamp. In general, a watermark tracks the event time of the stream it is flowing in. When a watermark with timestamp $t$ is received, then the operator is sure that all the following elements in that stream will have a timestamp greater than $t$. An example of that can be found in Figure 2.1.

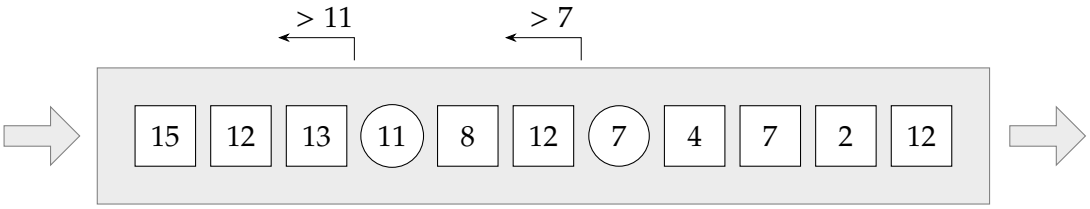



FIGURE 2.1: Watermarks (the circles) are elements of the stream. The value inside an element is its timestamp. All the elements following a watermark must have a timestamp greater than the watermark's one.

Watermarks are generated by the various sources. When an operator has multiple incoming streams, it will receive watermarks from each one of them and, in general, each stream will be at a different point in event time. In this case, the operator's event time is the same as the minimum event time of the incoming streams. When an operator receives a watermark that advances its event time, it will also forward the watermark downstream. This makes sure that even if the incoming streams are not synchronized in event time, the outgoing streams will contain coherent watermarks.

## 2.6 Cyclic computations

To add more advanced facilities, the dataflow model has been extended by the various processing systems that implement its features. In particular, iterations and

cyclic computations are one of the most common extensions.

While different systems provide different semantics, the main difference between iterations and all the other operators is that they create *cycles* in the dataflow graph. This means that the same data tuple can flow through the same operator multiple times before reaching a sink. When cycles are present in the dataflow graph, special care is needed to avoid never-ending or incorrect computations. In particular, there should be mechanisms that ensure that no tuple can get stuck in a cycle. Furthermore, special attention should be paid to how watermarks interact with cyclic computations.

# Background and Related Work 3

We already passed the point where it was enough to scale vertically, that is using more powerful hardware, to analyze very big datasets. All things considered, there is the need to scale horizontally exploiting the computational power of multiple machines.

There are already many solutions to this problem, including some that embrace the dataflow paradigm, and others that do not. Over the years, many researchers devised and implemented a vast variety of tools, libraries, and frameworks to help software engineers develop applications that are able to store and process huge amounts of data in a timely manner.

Expressiveness is a very important property of a platform, as you may not be able to solve your problem if you rely on a platform that is not expressive enough. Also, performance and scalability are key to a successful system, as they reduce the computational requirements and therefore the costs needed for the infrastructure. One must also not forget about the *ease of use*: an easier to use platform requires less code, which usually means less programming mistakes. Furthermore, developing on an easier platform makes it possible to deploy more quickly, experiment more with the code, and debug more easily.

Expressiveness, performance and ease of use are difficult to achieve together. In this section some of the most popular frameworks are presented, from industry standard frameworks such as MPI, Apache Spark, and Apache Flink, followed by Timely Dataflow, an emergent new framework written in Rust, and concluding with RStream, the framework that kick-started Noir.

## 3.1 Low-level solutions: MPI and OpenMP

Message Passing Interface (MPI) [12] and OpenMP [4] are two technologies that enable C, C++ and Fortran programs to achieve high level of parallelism: the former allows using multiple hosts in a cluster, while the latter allows exploiting local parallelism inside a single host. They are a very popular solution for HPC (High Performance Computing), offering APIs for executing parallel code using multiple threads, and providing communication primitives for a multi-host deployment.

Being pretty low level, MPI and OpenMP offer the highest flexibility to the developer, without forcing any parallel programming paradigm. In fact, MPI and OpenMP are not based on the dataflow model as the other systems described in this chapter. This allows the developers to write very efficient and optimized code, tuned for the specific application and cluster architecture. One notable difference with systems

based on the dataflow model is that, due to manual memory management, it is possible to modify the dataset in-place. In some cases, this might lead to a big performance improvement, given that this avoids doing allocations and copies of the memory. Instead, in the dataflow paradigm, each data stream is immutable and operators perform computations by transforming one data stream into a new one. Benchmarks (see Chapter 6) show how MPI and OpenMP are able to beat the other systems pretty much in every benchmark.

High performance comes at the cost of a much steeper learning curve, forcing the developer to implement from scratch most aspects of the application. The developer must think about the cluster topology, the data distribution, the communication, the parallelism, the synchronization and so on. Usually MPI and OpenMP programs are longer to write and debug, it is hard to tune them to achieve the best performance possible and they are also more difficult to maintain. Not only that, but C and C++ are considered to be harder languages to use, also because of manual memory management.

RStream and Noir took inspiration from how MPI spawns processes. In particular, the same MPI program is executed on multiple hosts, and runtime checks based on the *process rank* determine the behavior of the program. Similarly, in Noir the same program is executed, the same execution graph is built, but different hosts execute different parts of the program.

## 3.2 Dataflow-based Frameworks

This section describes various frameworks which provide many abstractions that can make the development of distributed computations much easier. These systems do not have exactly the same feature sets, but they are all inspired by the dataflow paradigm described in Chapter 2. In particular, they model the computations as networks of operators, where each of them transforms one dataset into another.

### 3.2.1 Apache Spark

Apache Spark [13] is an open-source system for large-scale distributed data processing. It is implemented in Scala, and it provides APIs for Java, Scala, Python, and R. It provides an abstraction based on the concept of *Resilient Distributed Dataset* (RDD).

An RDD is a dataset whose elements are partitioned and distributed across multiple machines in a cluster; it is created from a file saved in supported file systems or from collections in the driver program. Two types of parallel operations can be performed on RDDs: *transformations*, which generate a new RDD from a given one (e.g. `map`), and *actions*, which run a computation on the dataset and return some results (e.g `reduce`).

Whenever a new operation is invoked on a RDD, Spark creates a new task which is then scheduled on the workers present in the cluster. Each task is divided in subtasks which process each partition of the RDD.

Unlike other systems like *MapReduce* [5], Spark supports complex graph computations and provides many high-level operations including joins and iterations. In particular, iterations are achieved by reusing the same RDD multiple times. To make

the execution of iterative programs faster, RDD can be persisted to memory, instead of being recreated and recomputed each time they are accessed.

While initially supporting only batch-processing on RDDs, Spark has since been expanded with many more functionalities. In particular, Spark also provides an interface to define streaming computations: by default it uses a *micro-batching processing engine*, but from version 2.3 it also provides a new *Continuous Processing* mode which is able to reach much lower latencies, sometimes as low as 1 ms. Queries can also be defined using the SQL language, thanks to the Spark SQL module. Finally, the *MLlib* and *GraphX* libraries include ready-to-use algorithms for machine learning and graph processing.

Given that both Spark and Flink provide similar features, that they are both written in languages that run on the *Java Virtual Machine* (JVM), and that they have similar performance [7, 9], we decided not to consider Apache Spark during the evaluation of Noir and its comparison with other distributed data processing systems.

### 3.2.2 Apache Flink

Apache Flink [3] is an open-source stream-processing and batch-processing framework. It follows the dataflow paradigm just like RStream and Noir. In fact, RStream design was heavily influenced by Flink, and many of Flink's concepts are still present in Noir.

Flink is written in Java and Scala, offering APIs for Java, Scala and Python. Batching tasks are expressed via the DataSet[1] API, while streaming tasks are defined using the DataStream API. It is also possible to use relational APIs expressing queries using an SQL dialect, or using the Table API.

Java is a compiled language, so most of the programming mistakes are caught early on. The type system is able to enforce that the type of the tuples is consistent. Unfortunately, it is still pretty easy to build an invalid topology that compiles well, but fails to execute. Furthermore, sometimes it is also necessary to help the type system with annotations [18] (see Listing 1).

```
1 env.fromElements(1, 2, 3)
2     .map(i -> Tuple2.of(i, i))
3     .returns(Types.TUPLE(Types.INT, Types.INT))
4     .print();
```

LISTING 1: In this simple example, the only operator is a `map` that creates a tuple with two integers using a lambda function. Unfortunately, the Java type system is not able to expose the lambda's return type to Flink, so the user needs to explicitly state it. Note that without line 3 the snippet compiles but throws an `InvalidTypesException`.

In Listing 2 there is a simple implementation of the word count algorithm. Just like the example shown for RStream (see Listing 3), this code reads the input file and processes it all in parallel. With respect to the same example in RStream, in some cases the Flink's code is more verbose. It might be required to define a new class

[1] Starting from Flink 1.14, the DataSet API will be unified with the DataStream API.

even for the most simple operators (`FlatMapFunction` in this case), or to explicitly specify the return type. On the other hand, some operators like `groupBy` and `sum` are more convenient to use since they are better integrated with tuples.

```
MultipleParameterTool params = MultipleParameterTool.fromArgs(args);
ExecutionEnvironment env =
 ↪  ExecutionEnvironment.getExecutionEnvironment();
env.getConfig().setGlobalJobParameters(params);

DataSet<String> text = env.readTextFile(params.get("input"));
DataSet<Tuple2<String, Integer>> counts = text
    .flatMap(new FlatMapFunction<String, Tuple2<String, Integer>>() {
        @Override
        public void flatMap(String value, Collector<Tuple2<String,
         ↪  Integer>> out) {
            for (String word : value.split(" ")) {
                out.collect(word);
            }
        }
    })
    .groupBy(0) // group by word
    .sum(1);    // sum the counts
counts.count();
```

Listing 2: A simple word-count implementation in Apache Flink.

Even if the feature sets of Spark and Flink are quite similar, there is a fundamental difference in the way they parallelize and distribute the computation. Considering batch-processing and micro-batch stream-processing, Spark creates a task each time an operation needs to be executed, which is then scheduled on the available machines. On the other hand, Flink directly schedules *pipelines* on the various nodes, which are composed of instances of consecutive operators. This means that each instance of each operator is started at the beginning of the computation, and it keeps running until the very end.

Flink is very mature and its public repository received contributions from nearly a thousand contributors. It offers many advanced features, like optional fault tolerance, exactly-once semantics, and a rich web interface. Unfortunately its performance is the worst of all the systems analyzed (see Chapter 6).

### 3.2.3 Timely Dataflow

Timely Dataflow is a framework based on Naiad [11], which processes data in a dataflow manner. Naiad's first implementation was in C#, but it was later ported to Rust with the name of *Timely Dataflow* [34].

It is a very efficient dataflow framework, but it is not a good fit for every application. Its API exposes very few low-level operators, which makes this framework not always easy to use in real-world applications.

A more expressive system is Differential Dataflow. Differential Dataflow [10] is an advanced dataflow framework for differential computations written on top of Timely Dataflow. It exposes a few more operators, but at the cost of losing some performance due to the fact that it is also able to perform differential computations.

Timely Dataflow tracks the progress of time in a way that is similar to watermarks (see Section 2.5), but differs from that approach in some key aspects. Instead of explicit messages flowing in the stream, each operator keeps a set of *capabilities*, and a *frontier* of the upstream operators' capabilities. A capability of $t$ is a token that allows an operator to send messages with a timestamp greater or equal to $t$. When the operator knows that no more messages will ever be sent with a given timestamp, it can drop or *downgrade* the capability. Internally Timely Dataflow keeps track of all the capabilities alive and will update the frontiers of the downstream operators accordingly. This time tracking mechanism is very powerful, but it makes it difficult to implement new operators.

Looking at performance and ease of use, Timely Dataflow lays between MPI and Noir. In fact, Timely Dataflow usually offers marginally better performance than Noir, but it is not as easy to use. In particular, given that it exposes only few primitives, many operators need to be implemented from scratch.

In Section 6.3.1 we will see how Timely Dataflow performs against the other systems in one of the presented benchmarks. Its lack of an extensive API makes it less suitable for our benchmarks. Indeed, there is no out-of-the-box support for either windows or join operations.

### 3.2.4 RStream

RStream [6] is a very recent framework based on the dataflow paradigm, developed by Alessio Fino at Politecnico di Milano. It fully embraces the dataflow paradigm, supporting sources, operators and sinks of various types. Noir is inspired by RStream and can be considered its successor, as it shares some internal architectural design choices and their APIs are sometimes very similar. There are also many important differences, both in the architecture and API (see Section 4.10).

Both RStream and Noir are written in Rust [32], a pretty recent programming language with many interesting features:

- It is really fast, usually with comparable performance to C and C++: it is compiled by the `rustc` compiler, which internally uses the LLVM backend.
- It guarantees memory safety at compile-time with strict borrow-checking rules, drastically reducing the number of runtime crashes.
- It is convenient to use: there are many battle-tested tools and the "ecosystem" is very rich and active.
- It is easily extendable: it is very easy to use external libraries, called *crates*, as dependencies, thanks to the publicly accessible crate registry [16], and the language can be extended using *procedural macros*.
- It is evolving: the language itself is actively developed, as well as its ecosystem.

- It is expressive: it provides language constructs that help developers, but that are not too abstract.

The main features of RStream are its ease of use, and its very good performance. It can compete head-to-head with MPI and OpenMP while being very simple to use and very concise. RStream can handle both batching tasks (i.e. with a finite dataset) and streaming tasks (i.e. with an unbounded amount of data). Unfortunately, its expressiveness is not great: many essential operators like `join` are missing, and it does not support complex computation graphs which include nested iterations, side inputs and concurrent streams. In any case, both RStream and Noir lack the support for fault tolerance: if an operator or a host crashes, then the entire job fails, and it has to be restarted from the beginning. The objective of Noir is to extend RStream and improve its expressiveness, while keeping comparable performance and ease of use.

In Listing 3 there is a simple implementation of the word count algorithm. This is a complete program that is able to read the dataset, split each line into words and count how many times each word appears, distributing the computation on multiple hosts available in the cluster. The same algorithm is implemented in Listing 4, but this time using the Noir framework. In Chapter 6 the performance of a similar algorithm is compared between all the presented platforms.

```
fn main() {
    let path: String = env::args()
        .nth(1)
        .expect("Pass the dataset path as an argument");
    let word_count: Vec<(String, u32)> = Stream::from_readlines(&path)
        .flat_map(|line| line.split_whitespace())
        .group_by_reduce(
            |(word, _count)| word.clone(),
            |(word, count1), (_word, count2)| (word, count1 + count2),
        )
        .collect_vec();
    finalize();
    println!("{}", word_count.len());
    Ok(())
}
```

Listing 3: A simple word-count implementation in RStream. This program takes as argument the path to a text file, it reads it in parallel on all hosts, and counts how many times each word appears.

```rust
fn main() {
    let (config, args) = EnvironmentConfig::from_args();
    let mut env = StreamEnvironment::new(config);
    env.spawn_remote_workers();

    let path = &args[0];
    let source = FileSource::new(path);
    let result = env
        .stream(source)
        .flat_map(move |line| line.split_whitespace())
        .group_by_count(|word| word.clone())
        .collect_vec();
    env.execute();
    if let Some(res) = result.get() {
        eprintln!("{}", res.len());
    }
}
```

Listing 4: The same program of Listing 3, but implemented using Noir.

# 4 Architecture

This chapter shows the internal structure of Noir, analyzing its architecture and design choices. In the end there will be a discussion on the differences between RStream and Noir.

## 4.1 Job Graph and Execution Graph

The *Job Graph* is a directed graph in which the nodes are the operators, and the arcs are the connections induced by the inputs and outputs of the operators. If an operator $B$ receives data from an operator $A$, then there will be an arc connecting $A$ to $B$.

Let us consider a simple example of a stream formed by a source of integers, an operator that doubles each number, and finally a sink. The resulting Job Graph is very simple (Figure 4.1), there are just 3 nodes, one for the source ($A$), one for the operator ($B$) and one for the sink ($C$). The operator receives data from the source, so there is an arc from ($A$) to ($B$); the operator sends data to the sink, so there is an arc from ($B$) to ($C$).

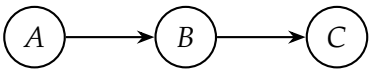



FIGURE 4.1: A simple Job Graph.

Many graph topologies are supported, including graphs with cycles when the stream contains iterations (see Section 4.7). Each node can have zero, one or more predecessors and zero, one or more successors. A source has no predecessors, while a sink has no successors. There exist operators with more than one predecessor (e.g. `join`), and operators with more than one successor (e.g. `split`). In Figure 4.2 there is an example of a complex Job Graph that includes an iteration loop ($C - D - E - F$), a `split` operator ($B$), and a `join` operator ($E$). Note that ($E$) receives data from ($G$) as an iteration side-input.

To achieve higher parallelism, most operators are instantiated multiple times, and each copy of an operator is called *instance*. The Job Graph does not consider how operators are partitioned, so the instances are not represented. This means that each operator is represented only once in the graph, even if during the execution it will be instantiated on many hosts. To represent the actual connections between instances we introduce the Execution Graph.

The *Execution Graph* is built from the Job Graph. Each operator, that is each node of the Job Graph, is duplicated as many times as the number of its instances in

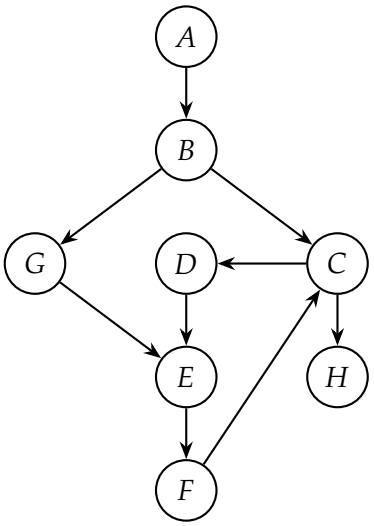



FIGURE 4.2: A more complex Job Graph.

the system. The connections between nodes in the Execution Graph show how the data is transferred between the instances. These connections are derived from the ones present in the Job Graph, considering also the *forward strategy* of each operator, which will be later described in Section 4.2. From the Execution Graph it is possible to see how the operators are partitioned into independent subtasks, and see how they are assigned to the various hosts. Noir provides tools to analyze these graphs and the load of the network caused by them (see Appendix B).

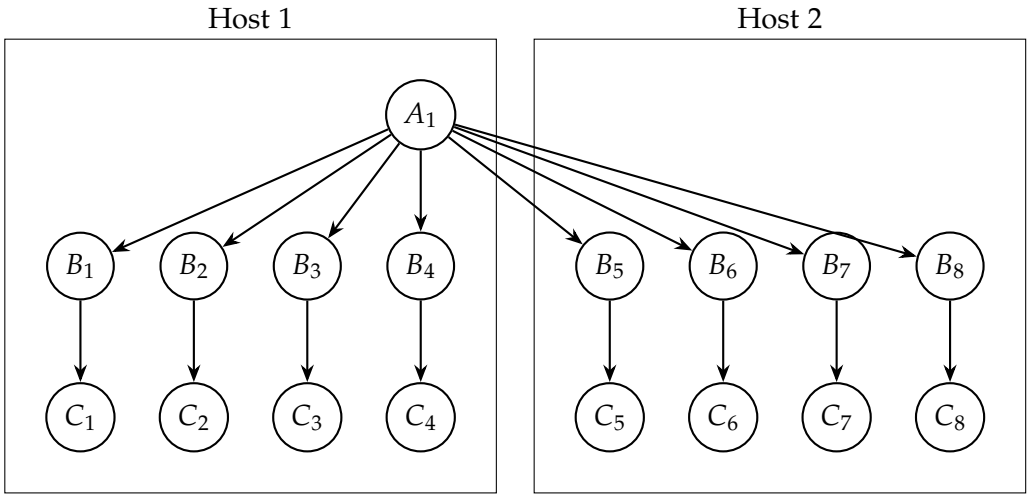



FIGURE 4.3: A possible Execution Graph relative to the Job Graph in Figure 4.1.

As an example, consider the previous Job Graph: assuming that the cluster has 2 hosts with 4 cores each, Figure 4.3 shows a possible Execution Graph. In this example, let us assume that the source cannot be parallelized (for example because the numbers are generated by a not parallelizable function), but the sink can. Therefore, the execution graph will contain only one instance of the source operator, 8 instances of the intermediate operator and 8 instances of the sink.

Note that the Job Graph does not depend on the number of hosts nor on the number

of instances of each operator. On the other hand, the Execution Graph heavily depends on those factors, as well as the scheduling algorithm that determines on which host each instance is spawned.

## 4.2 Forward Strategy

When an element has been processed by an operator, it has to decide where to go next. Which operator to go to is decided by the links of the Job Graph, instead which instance of that operator is decided by the Forward Strategy. There are a few Forward Strategies an operator has to choose among.

**Random (Figure 4.4)** Each element will select a random instance where to go to. This can be used to reshuffle the data between the partitions of the stream, balancing the load. The next instance is selected according to a uniform distribution.

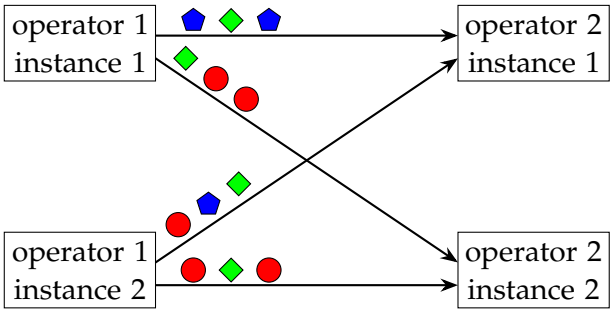



Figure 4.4: Random Forward Strategy.

**GroupBy (Figure 4.5)** Each element is processed by a *keying function* that extracts a hash from the value of an element. This value is then used to deterministically select the instance of the next operator where the element will go to. This makes sure that elements with the same key will end up in the same instance. Therefore, this Forward Strategy can be used for implementing the operators in the *group by* family (see Section 5.6.4).

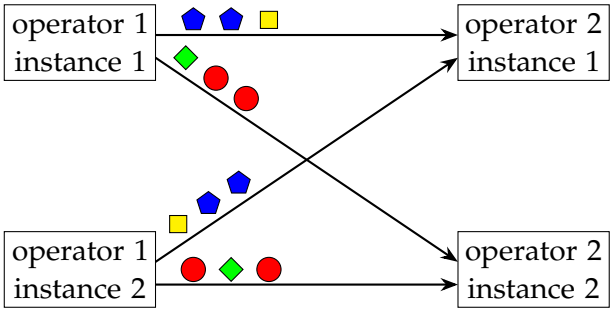



Figure 4.5: GroupBy Forward Strategy.

**OnlyOne (Figure 4.6)** Send each element to the *only* sensible destination. This strategy can be used only under specific conditions.

- If the next operator has only one instance, all the instances will send the messages to the *only one*.

- If the next operator is instantiated *exactly like* the current one (same number and same host allocation), each instance will send only to the corresponding instance in the same host.
- All the other topologies are not valid and this strategy cannot be used.

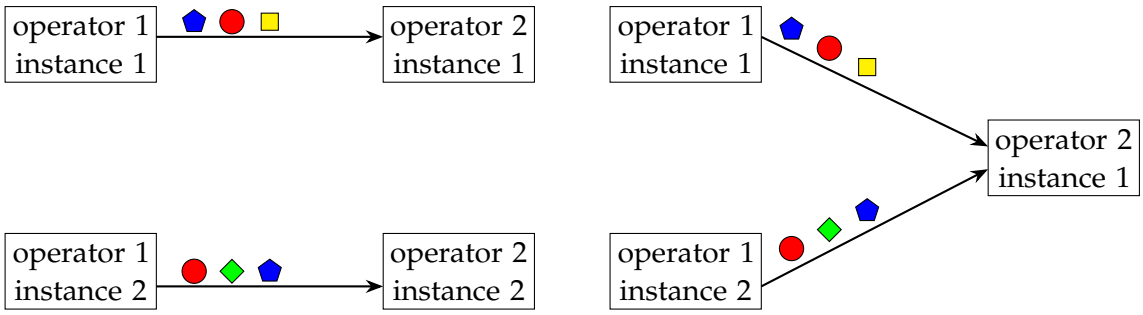



FIGURE 4.6: OnlyOne Forward Strategy.

**All (Figure 4.7)** Each element will be sent to *all* the instances of the next operator. This means that, unlike the other strategies, each message will be copied and sent to more than one destination. This is useful to implement broadcasting.

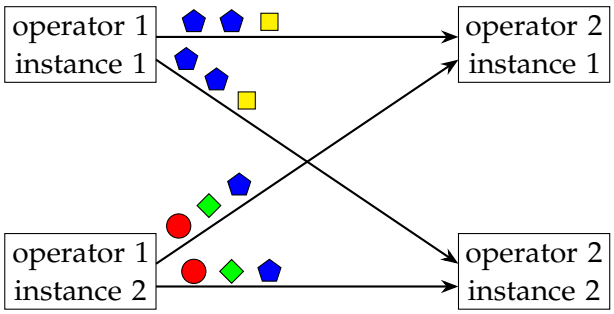



FIGURE 4.7: All Forward Strategy.

## 4.3 Block Structure

A *block* is a sequence of contiguous operators that are grouped for optimization purposes. The idea of fusing contiguous operators is not new, it is a well-established technique in query optimization [14].

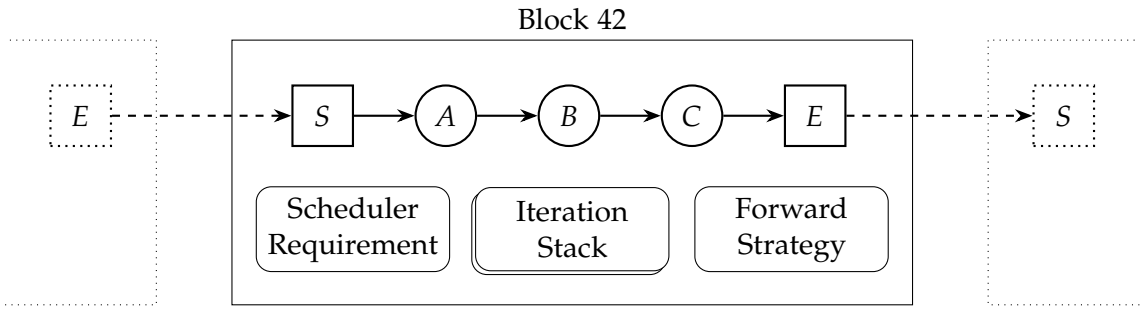



FIGURE 4.8: Overview of the internal structure of a Block.

In Figure 4.8, a high level overview of the block structure is shown. The block contains 3 operators ($A$, $B$ and $C$), with two structural operators $S$ and $E$, respectively the `StartBlock` and `EndBlock`. The purpose of those two additional operators is explained in the following section. Other than the operators, the block also stores some information for the scheduler and the runtime, such as the Scheduler Requirements, the block's Forward Strategy, and a stack with the iteration scopes (see Section 4.7).

An integer identifier, called *block ID*, is associated to each block. This identifier is unique inside the environment, and all the hosts in the cluster assign these identifiers in a deterministic way.

The main optimization obtained by grouping the operators is that all the communication overhead between those operators can be avoided. In fact, the operators inside a block rely on the Iterator pattern [22] to exchange data. An operator can ask for some data to the previous one simply by calling a function (see Section 5.4). This means that all the data exchanged in a block is passed as function return values; no synchronization and no network is required to do so.

Not all operators can be grouped, if an operator requires the network or does not use the OnlyOne Forward Strategy (see Section 4.2), the block has to be split into two. However, all the simple operators that just transform the incoming data do not require doing so.

To allow for such *function calls* to the previous operator, each operator keeps an *Operator Chain*. An Operator Chain is a recursive structure (see Figure 4.9) in which an operator contains the previous one, up to the first operator in the block. When the last operator of the chain (`EndBlock`) is asked for the next element, it can request some data to the previous one in the chain. This request can bubble up the chain recursively up to the first operator (`StartBlock`), which can fetch the data from the network.

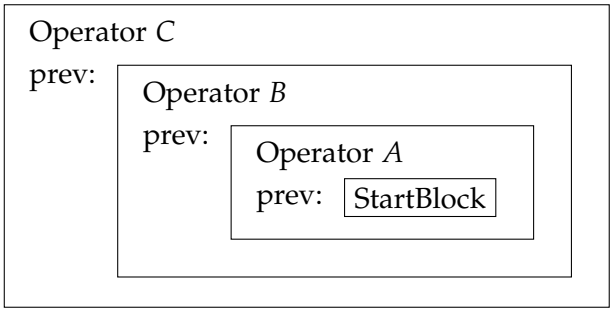



FIGURE 4.9: How operator $C$ from the example above stores its Operator Chain.

The Rust [32] compiler is able to *inline* most of those function calls, thanks to the *monomorphization* of the operators. This is possible only if the Operator Chain is part of the type of the operator; using *dynamic dispatch* (e.g. by using pointers) prevents this optimization. This inlining leads to huge performance improvements (see Section 4.9.1).

### Extra block operators

In Figure 4.8 there are 2 special operators inside the block, in the example called $S$ and $E$. Those operators abstract away the logic for the network from the actual operators ($A$, $B$ and $C$ in the example).

The first one is called `StartBlock` and its job is to read from the network and manage the state of the block and of the following operators. It receives the messages from the network, unpacks them into multiple elements, and lets them flow downstream one at the time. It also keeps a *watermark frontier*: before letting a watermark pass downstream, it has to wait for all the watermarks coming from the other instances of the previous blocks. This is required to make sure that the watermark property is respected (see Section 2.5): if watermarks are simply let through, they can end up in the wrong order. This can happen, for example, when an instance falls behind the others, since they are not synchronized. For an example on how it is done see Figure 4.10.

| | $t = 1$ | $t = 2$ | $t = 3$ | $t = 4$ |
|---|---|---|---|---|
| instance 1 | 1 | 2 | 3 | |
| instance 2 | 1 | | 3 | |
| instance 3 | | 1 | | 3 |
| frontier | 0 | $\underline{1}$ | 1 | $\underline{3}$ |

FIGURE 4.10: Watermark frontier: at $t = 1$ the frontier cannot advance to 1 since the third instance has not sent watermark 1 yet. At $t = 2$ all the instances sent a watermark greater or equal to 1, so the frontier can advance to 1. At $t = 3$ instance 2 skipped watermark 2 and jumped to 3, still the frontier cannot advance due to instance 3. At $t = 4$ all the instances sent a watermark greater or equal to 3, so the frontier can advance to 3. The frontier will emit a watermark every time it changes value (values underlined).

`StartBlock` also keeps track of the *iteration state lock*, a data structure that prevents out-of-sync instances from processing elements from the wrong iterations (see Section 4.7). Additionally, it may also keep track of the cache of the incoming elements to be replayed by the `replay` operator (see Section 5.6.8).

After all the operators of a block there is another special operator called `EndBlock`. Its job is to collect the elements produced by the block, group them into batches and send them to the correct instance of the next block. The Batch Mode (see Section 4.4) defines how elements are grouped into batches, and the Forward Strategy (see Section 4.2) describes which instances will receive them.

According to the Batch Mode, the `StartBlock` can insert into the stream a control element called `FlushBatch` (see Section 5.5), that will cause the corresponding `EndBlock` to stop batching the elements and flush the current batch, even if it is not full. This element is generated by the `StartBlock` instead of the `EndBlock` since it is much more efficient to put a timeout on the receiving side of a channel (i.e. a network timeout) instead of tracking the time at the end of the block. As described

in Section 4.9.5, that would have required adding new threads, since that operator is blocked on the call to get new elements from the operator chain.

## 4.4 Batch Mode

The `EndBlock` operator, after applying the Forward Strategy to determine where an element is directed, batches the elements into messages. Each message will contain one or more elements that will be sent to a specific instance.

The Batch Mode controls how the messages are buffered, in particular it determines when a batch should be closed and sent to the instance. There are two batch modes, Fixed and Adaptive.

**Fixed($N$)** Each batch will contain at most $N$ elements. In practice, each batch will contain exactly $N$ elements, except the last batch which may contain fewer elements. This mode is the most lightweight of the two, since it does not depend on the passing of time or on timeouts. The biggest downside is that if the stream has a low throughput and $N$ is set to a value too high, the stream can be slowed down increasing the overall latency. In fact, the maximum latency of a block is not bounded; for example, if the source stops for an arbitrarily long time, the batch will not be flushed and its elements will be stuck in the queue.

**Adaptive($N$, $t$)** Similarly to Fixed($N$), Adaptive($N$, $t$) produces batches of at most $N$ elements. Furthermore, it tries to limit the time an element is stuck waiting in a batch. Formally, a batch is flushed either when it reached $N$ elements, or when its elements waited more than $t$ time. Checking the timing condition is more complex than checking the one of Fixed($N$). There is an optimization in place to measure time very quickly: on supported platforms, time intervals are measured using `TSC` instructions [33] and the `coarsetime` [27] crate is used to avoid expensive syscalls. This limits the precision of the measured time, but it is much faster; experimentally, the time resolution is about 3–4 ms. Compared to Fixed($N$), this mode keeps the maximum latency to a bounded value, at the expense of a slightly higher overhead.

In Section 6.3.10 we explore in more detail the impact of the batch mode on the overall latency.

## 4.5 Block Allocation

This section will describe how the execution plan is computed, passing from the Job Graph to the actual Execution Graph.

All the hosts in the cluster have to agree on the same Execution Graph, otherwise if different hosts construct different Execution Graphs the computation will not be correct. There are two ways to achieve this: either hosts use a deterministic algorithm to build the execution plan, or it is computed on a single host and it is later shared with the others. We opted for the former approach, since it avoids the synchronization step due to the communication between hosts. This approach also allows for an online scheduling approach, that is to say building the Execution Graph while building the Job Graph.

When the Job Graph is being built, the scheduler's job is to keep track of all the operators present. When an operator requires the block to be split, the scheduler is notified, it takes the new block and schedules it. Firstly, it determines the number of instances to spawn by looking at the scheduler requirements of the block. Then, it selects *deterministically* where to allocate those instances.

Where to allocate the instances of a block is determined following the lexicographic order of the hosts. Each host can accept up to as many instances of a block as it has cores (defined in the configuration file). Note that each host can hold, between all blocks, more instances than cores. This resource *over commitment* helps to balance the load between the blocks: if two blocks *alternate* their computational requirements, they can share the same CPU core. This approach is very similar to the one used by Flink with the *Task Slots* [19]. The job of balancing the CPU time between different instances is left to the kernel scheduler. Some (failing) attempts have been made to help the kernel scheduler optimize better the CPU time allocation by using the `sched_setaffinity` [31] system call (see Section 4.9.2).

In Figure 4.11 you can find an example allocation.

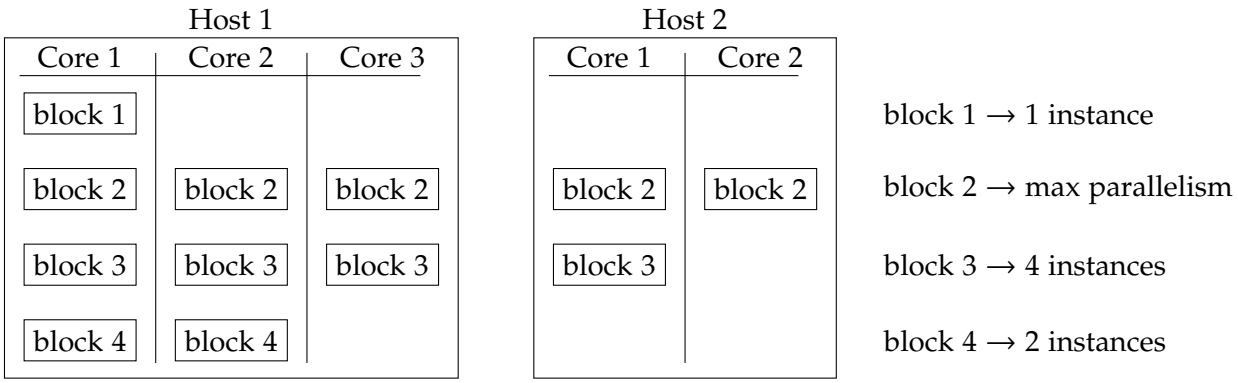



FIGURE 4.11: Example of instance allocation. There are two hosts with 3 and 2 cores respectively. The first block is a bottleneck, the second is instantiated as much as possible, the third and forth have a limited parallelism. All the instances are assigned lexicographically from host 1/core 1 to host 2/core 2.

This algorithm is a trade-off between simplicity and efficiency.

- The algorithm is as simple as it gets, and it is deterministic given a deterministic assignment of host IDs.
- The first hosts in the list may tend to be the most loaded of the cluster, but on the plus side the locality of the data is higher than with a better distributed load. In practice, in our benchmarks this non-uniform distribution of load is not evident.

In practice, most of the time, a block is either instantiated exactly once (e.g. reduce on non-keyed stream), causing a bottleneck, or as much as possible. This opens the possibility for more sophisticated scheduling algorithms. Consider the case where a "bottleneck" block is preceded and followed by blocks with maximum parallelism; for sure, there will be a network shuffle both before and after the block. Under these conditions, we can allocate these "bottleneck" blocks in a round-robin fashion. Doing so, the load is spread better without changing the locality of the data.

Unfortunately, this is not doable online, since it is not possible to know if the block will be followed by a block with maximum parallelism at the time it is scheduled.

When the instances of a block have been allocated to some hosts, each of those hosts will spawn their instances and start them. Inside each host there is only one process, and each instance is run by a single *thread*. This means that a host can handle more than one instance just by running more threads in parallel. This allows for better in-memory communication between instances inside the same host: in-memory channels are used wherever possible, avoiding serialization and the usage of the network stack.

## 4.6 Network Topology

After the blocks have been instantiated and allocated to the hosts, they have to communicate. To do so, Noir provides some communication primitives for sending and receiving messages.

Each instance has one or more Receiver Endpoints. A Receiver Endpoint is an identifier of a connection from a block to an instance. In fact, each instance can receive messages from all the instances of its previous blocks. The messages coming from the instances of the same block are grouped together and will go through the same Receiver Endpoint. Therefore, a Receiver Endpoint contains the identifier of the sender block, and the coordinate of the receiving instance.

The identifier of the sender block is important because it determines the type of the serialized messages going through the channel. In fact, a block may have more than one preceding block, and they can send elements of different types (e.g. a join can merge streams of different types).

The operators do not directly control the network layer, they simply use some high level APIs for sending and receiving messages. Internally, the network layer manages the connections between instances and the message serialization/deserialization. This layer uses two kinds of network channels: local and remote.

Local channels are used for all the connections between instances inside the same host. Since the instances are threads of the same process, they can communicate by sending messages using in-memory channels. This avoids completely the kernel network stack and skips the serialization/deserialization step. They are implemented using either flume [29] or crossbeam [28]; the user can change which implementation to use with a compiler flag. Crossbeam offers better performance when the channel is not saturated, while the performance of flume is more consistent (see Section 4.9.3). These in-memory channels are multiple-producers single-consumer queues. When an instance wants to send a message to another instance, it simply enqueues the message to the queue of the recipient. An example of this structure can be seen in Figure 4.12.

Remote channels are more complicated. They use TCP sockets between the hosts, but to avoid an excessive number of open sockets, some of them are aggregated. In particular, ideally there would be a direct connection from each instance to all the instances it connects to. This leads to a number of connections that is quadratic on the number of instances. To reduce them, Noir aggregates all the channels between

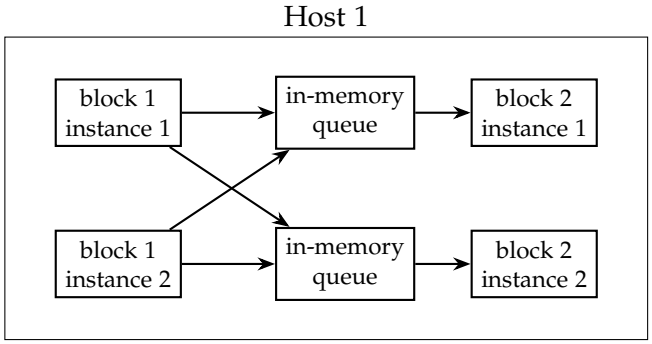



FIGURE 4.12: Communication between instances in the same host.

two blocks of a pair of hosts. In other words, all the instances of a block in a host will use the same connection for sending messages to all the instances of the next block in a given host. The number of connections is now quadratic in the number of hosts (which is significantly smaller than the number of instances). A high level overview of this topology can be found in Figure 4.13.

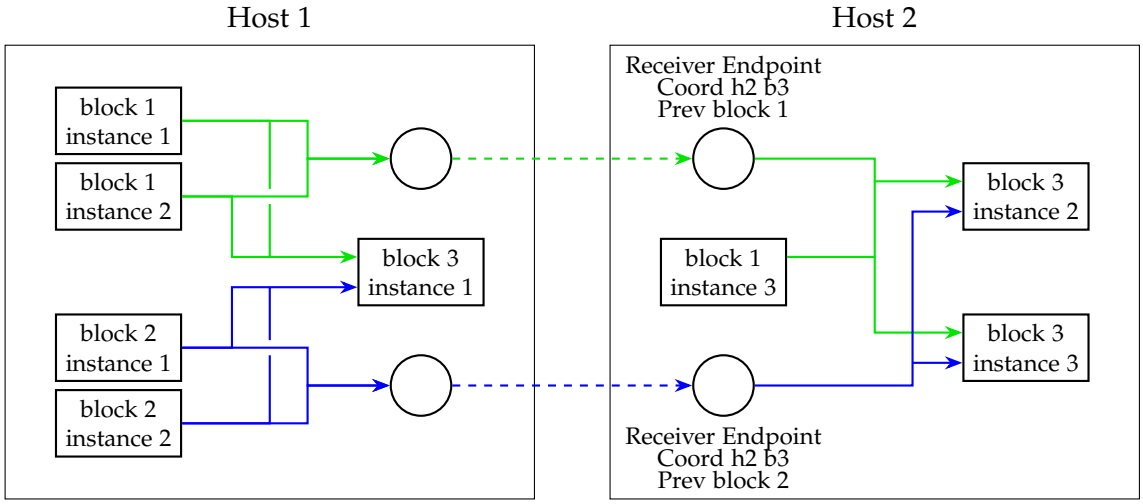



FIGURE 4.13: High level overview of the network topology: there are two hosts and 3 blocks. Block 1 and 2 send data to block 3. The first block has 3 instances (2 in the first host, 1 in the second), the second block has two instances in the first host, and the third block has 1 instance on Host 1 and 2 instances on Host 2. Colors identify the type of elements in the channel, dashed arrows are serialized messages. This image only represents the connections from Host 1 to Host 2 (the connections from Host 2 to Host 1 are not represented).

Reducing the number of channels may lead to an increased contention of those channels. If the load is heavily unbalanced between the partitions, the overload of one instance may cause back pressure. If this happens, the back pressure will propagate through the channels backwards, and when it reaches a "shared" channel it may impact other instances. Indeed, that channel will be saturated by the messages sent to the slow instance, while the other instances will starve from the lack of incoming bandwidth. This effect is represented in Figure 4.14. This problem can be mitigated by increasing the size of the receiving buffers, but it cannot be fully avoided. In general, a system that suffers from back pressure may be slowed down not only

where the pressure is generated, but also in multiple unrelated parts.

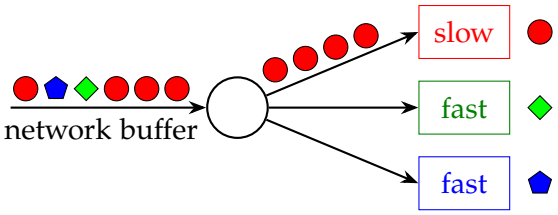



FIGURE 4.14: The topmost instance cannot keep up with the incoming messages, causing them to queue up. This leads to the saturation of the network receiving buffer, the other instances have to wait for the filled buffer.

In Figure 4.13 and Figure 4.14 there are components, represented with circles, that handle the network connections. They are called Multiplexers and Demultiplexers. Internally, they are made of few components that handle the network, the queues and the message serialization. Figure 4.15 shows in the details how these components are structured.

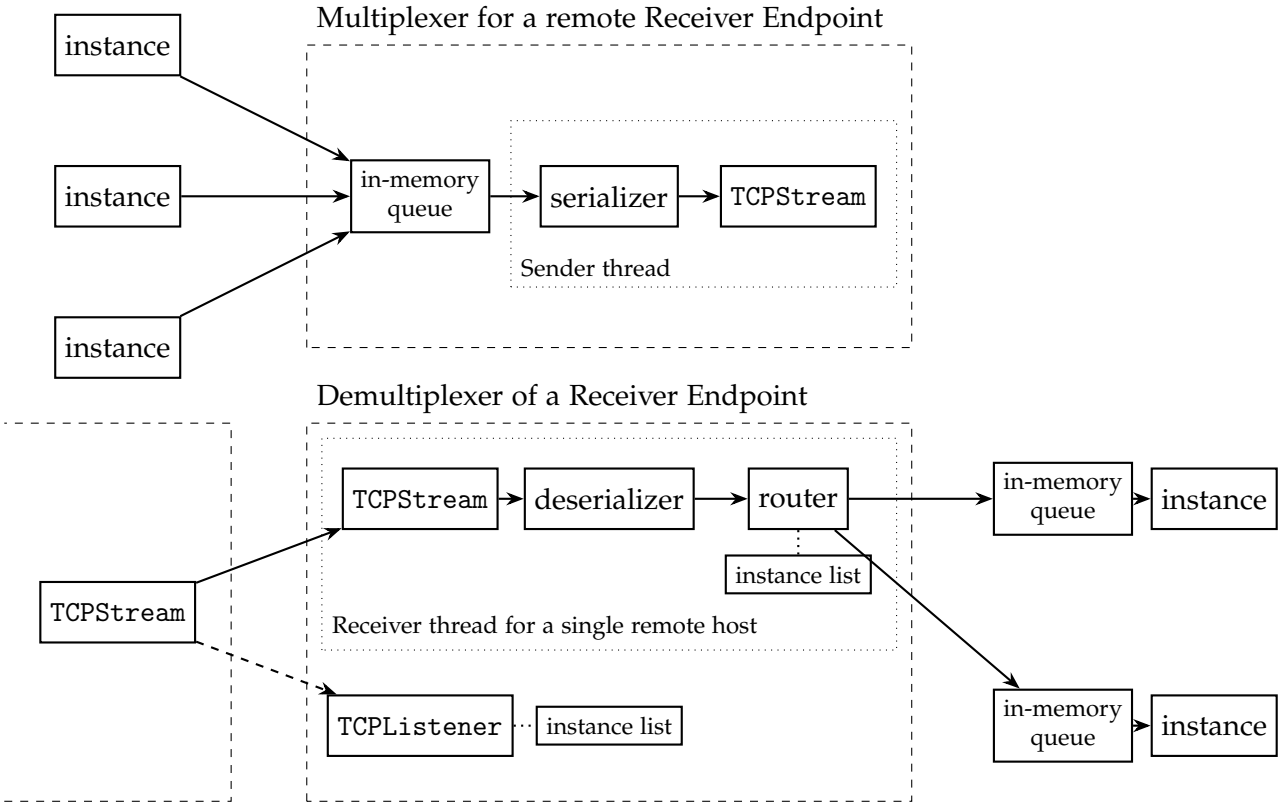



FIGURE 4.15: Internal structure of multiplexer and demultiplexer.

The Multiplexer receives the messages to send via an in-memory queue from the local instances. This queue has the same characteristics of the ones used for the local connections between instances, so the multiplexer is transparent to the senders and it acts like a local instance. There is a consumer thread that reads from the queue, serializes the messages and sends them via a TCP stream.

The TCP stream of the sender connects to the TCP listener on the receiving side. Each demultiplexer listens on a TCP socket whose port is deterministically computed,

therefore all the hosts know all the port assignments. These ports are chosen sequentially from a *base port* that is set in the configuration file.

When a multiplexer connects to the demultiplexer, a new thread is spawned for handling the socket. This thread reads from the TCP socket and deserializes the messages. The messages are then routed based on their actual destination, and sent to the recipient in-memory queue. This is the same queue used for the local connections between instances, therefore also the demultiplexer is transparent to the instances.

The router needs to know the list of local instances, together with their local in-memory queues. The demultiplexer main thread and the host threads keep these lists up-to-date sending messages to each other. This synchronization is required because the network stack starts concurrently with the local instances, so a remote connection can arrive before all the local instances are ready. When an instance is ready to receive messages, it is added to these lists.

The serializers and deserializers use the bincode [25] crate for serializing the messages into binary format. The actual protocol is very simple: each network message is composed of a header followed by the actual message (payload). The header has fixed length and contains the length of the payload, together with some information about the recipient. The payload is serialized using variable length integer encoding [26], as this heavily reduces the network usage while not compromising by much the performance of the system (see Section 6.4).

## 4.7 Iterations

Noir has a good support for iterations, see Section 5.6.8 for a detailed description of the API. In this section it is described how they are implemented, since they require special care to make sure the correct semantics are preserved.

Similarly to *imperative programming loops*, Noir's iterations crate a new *scope*. A scope wraps a group of operators that are executed repeatedly. The scope can be seen as a new stream whose input is fed to the iteration body, and its output is processed according to the kind of iteration, eventually feeding it back for the next iteration. Inside a scope there can be any type of operator, including more iterations, achieving this way *nested* iterations.

There are two kinds of iterations: `iterate` and `replay`. In the former the output of an iteration is the input of the following iteration. In the latter the same input stream is fed to the iteration body repeatedly, but the output is not fed back.

To signal the end of an iteration, a `FlushAndRestart` (see Section 5.5) is sent in the stream. This is used to inform all the operators in the loop body that the iteration is over, and the next elements will be part of the next iteration. Aggregating operators are therefore able to flush their content and stop the current aggregation, before resetting for the next one.

### State variables

The iterations support *state* variables, which are global variables that are shared between iterations. A state variable can be used to keep track of the progress of the iteration, or to keep track of partial results of the computation. They are similar to *broadcast variables* in Flink.

The stream inside a scope has read-only access to the state variable. At the end of each iteration there is a two-phase reduction for updating the state variable. Each stream item of a partition is locally reduced to a `DeltaUpdate`. The state variable is then updated once per iteration using the delta updates of that iteration. The state is updated in a dedicated operator called `IterationLeader`. The `IterationLeader` is a special operator that is not duplicated: for each scope there is exactly one leader. The leader is responsible for updating the state variable, checking the *loop condition* and coordinating the various iteration operators.

When all the partitions have completed an iteration, they send their delta updates to the leader. The leader computes the new state, and checks if a new iteration should start. Then, it communicates to all the instances the new state and whether a new iteration is necessary. When the instances receive the new state, they will begin the new iteration, if needed. It is important to synchronize the start of the iterations between the partitions; to do so Noir uses special locks, as described in the following section.

### Iteration State Lock

Consider the case of an iteration with two hosts. Inside the scope there are two blocks, each with one instance in each host. The iteration leader is in the first host, and the network connection to the second host has a high latency (see Figure 4.16).

Each partition has an operator that knows to wait for the leader before starting the next iteration. Unfortunately, the following blocks in the stream do not have such an operator.

Now consider this scenario:

- An iteration ended, all the operators are informed because they received the `FlushAndRestart` element;
- The delta updates have been sent to the leader;
- The leader has updated the state variable and checked the loop condition;
- The leader has sent the new state to the instances, but the second host has not received it yet;
- The first host has received the new state, it starts the computation;
- The first block of the first host produced some data and sent it to the second block of the second host (the red link in Figure 4.16);
- The second block of the second host must not start processing that message since its state is not yet updated;

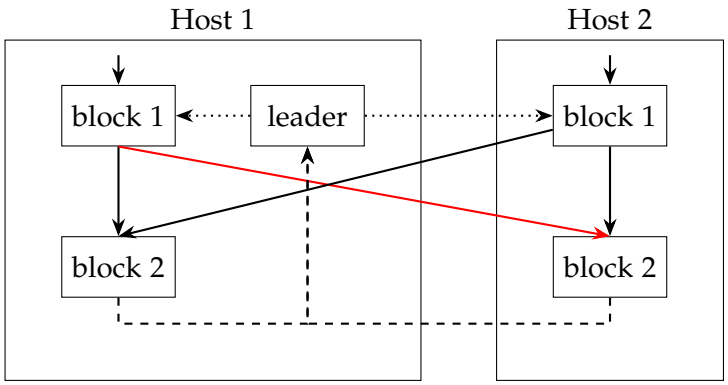



FIGURE 4.16: Example topology of a problematic iteration. The two Block 1 contain the iteration operators that communicate with the leader. The dashed links transport delta updates, the dotted ones transport the new state, and the red link is the one causing the problems.

There is the need to synchronize the start of the blocks in a host, otherwise the incoming data from other hosts trigger the start before the state is updated. There are two main ways to do this:

- Using a distributed lock between all the instances and the leader;
- Using a local lock on the state variable.

The first approach consists in implementing two-phase commit: firstly the leader sends the new state to the instances, they update it and inform the leader, and then the leader confirms it starting the next iteration. This way no instance can start the iteration before the state is globally known. On the flip side, this adds a synchronization step that can be costly and does not scale well. Furthermore, all the instances have to wait for the slowest host, instead of immediately starting.

With the second approach it is possible to achieve lower latencies, and it scales better. Each `StartBlock` (which is present in all the blocks) can detect if the state is out-of-date, and if so, it waits before receiving messages from the other hosts. To do so it knows the current iteration number, and when the iteration ends it waits for the state update before starting to receive. This way, when a state update is received, the instance can immediately start processing the new messages, without waiting for the other hosts. Noir uses this solution.

## 4.8 Cluster Spawning

Noir supports both local-only and distributed computations. The first is especially useful for developing and debugging the algorithm, while the second allows you to exploit the combined computational power of a cluster of machines. Starting Noir locally is very simple: since for each host there is only one process, it's enough to start the compiled binary on the local machine, and the computation will be parallelized using threads.

Spawning a job on multiples machines requires more steps. First of all, every host must have access to a copy of the binary executable. Then, this binary should

be launched on each host individually. Finally, each process in the cluster needs information about every other running instance, including their network addresses, port numbers and number of cores.

Noir uses an approach similar to the one of RStream [6], which is inspired by MPI [12]. There is a single binary executable that is run by each host, and its internal behavior changes according to the identifier of the host. This means that all the hosts will agree on the same Job Graph and since the scheduler is deterministic, also the Execution Graph will match exactly.

RStream and MPI spawn a cluster job using an extra program (`streamrunner` and `mpirun`, respectively). Noir embeds this program directly into the program binary. Doing so simplifies the cluster spawning and the deployment. The resulting binary is therefore able to be executed locally as well as remotely without the need of recompiling; the behavior can be changed via command line arguments.

To start a cluster job, the user has to execute the compiled binary in any host that is able to connect to all the other hosts of the cluster using SSH. The binary detects that the user wants to start a cluster job and, instead of running the computation locally, the process performs these actions:

- Connects via SSH to each remote host;
- Sends via SCP the binary executable to a temporary directory;
- Starts the binary passing the configuration via environment variables (e.g. a copy of the configuration file, the identifier of the remote host, extra Rust logging configurations);
- Waits for the remote process to finish;
- Collects debugging metrics (if enabled);
- Cleans up the remote resources (remove the temporary executable).

Using the suggested argument parser, it is possible to spawn the same binary providing `--local <number of cores>` or `--remote <config.yaml>`. In the second case, `config.yaml` is the configuration file that contains the cluster information. It uses the YAML [36] format that is both human-readable and very flexible. This file includes the list of hosts in the cluster, and optionally some extra configuration (e.g. SSH configuration and debugging settings). Listing 5 shows an example of a configuration file. In the hosts list, `address` is the hostname or IP address of the host, `base_port` specifies the first TCP port to use, and finally `num_cores` is the number of CPU cores available in the host.

## 4.9 Design Choices and Implementation Details

Some design choices and implementation details are analyzed in this section. Most of the choices described here are data-driven, and whenever possible the performance measurements are provided. This section explores many aspects of the design, so this will go throughout language features, networking aspects, library selection, custom allocators and even kernel parameter tweaking.

```yaml
hosts:
- address: host1    # either a domain name or an IP address
  base_port: 9500   # first port to use in this host
  num_cores: 32     # number of cores of this host
- address: host2
  base_port: 18000
  num_cores: 32
- address: host3
  base_port: 9500
  num_cores: 16
```

Listing 5: Minimal example YAML configuration file.

Even though this section is pretty vast, there is still plenty of room for improvements and performance optimizations.

### 4.9.1 Closures inside `Arc`

As described in Chapter 5, many operators need to receive and store some closures to execute computations defined by the user. For example, these include the closure used by the `map` operator to transform each item, the one used by `fold` to aggregate the elements of the stream, or the one needed by the `group_by` operator to extract the key from each tuple. Note that each closure is created once on each host in the `main` function, and then needs to be shared or cloned to be used by the various local instances.

Initially, these closures were stored inside an `Arc`, which is a thread-safe reference-counting pointer, using *trait objects*. This made the development of the various operators a little easier, since there was no need to add a new generic type parameter to the operators' definitions. For example, the closure used by `map` was initially saved as a `Arc<dyn Fn(Out) -> NewOut + Send + Sync>`, where `Out` is the type of the input items and `NewOut` is the type of the output. The closure was shared by the multiple local instances, but this did not represent a problem, given that the closure is stateless.

At some point during the development of Noir, we removed the `Arc` from every operator, adding a new type parameter to their definitions. Each closure also had to implement the `Clone` trait, so that it could be cloned once for each local instance. By doing so, we have seen unexpected performance improvements of more than 20% (see Table 4.1).

There are multiple possible causes for these performance improvements. First, not using `Arc` means that atomic operations for reference-counting are not needed anymore. However, this is unlikely to be the cause of the performance improvements, since the atomic operations are very few and are only done at the beginning and at the end of the computation. Furthermore, there was also no need to use trait objects and dynamic dispatch anymore. In any case, most of the improvements probably came from the monomorphization of each operator, which resulted in better inlining

| Hosts | Cores | With `Arc` | Without `Arc` | Improvement |
|---|---|---|---|---|
| 1 | 8 | 73.3 s | 58.5 s | 20.2 % |
| 2 | 16 | 37.8 s | 29.4 s | 22.2 % |
| 3 | 24 | 25.6 s | 19.9 s | 22.3 % |
| 4 | 32 | 19.7 s | 15.0 s | 23.9 % |

TABLE 4.1: Performance improvements due to the removal of `Arc`. The results refer to the associative *Wordcount* benchmark (see Section 6.3.1).

and helped the compiler to make more optimizations for each unique instantiation of the operators.

### 4.9.2 Core Affinity

Before the beginning of the development of Noir, we experimented with *core affinity*, which can be used to instruct the kernel's scheduler to bind a process or thread to a specific set of cores. In particular, we were interested in knowing if pinning each thread to a different core could result in better performance due to less context switches.

To do so, we implemented an ad-hoc version of the *Wordcount* benchmark in Rust that exploited only local parallelism through the use of multiple threads. To communicate between threads, we used the multi-producer multi-consumer channels provided by the `crossbeam` [28] crate. Each thread was then pinned to a specific CPU core using the `affinity` [24] crate. Under the hood, this crate uses the `sched_setaffinity` [31] syscall of the Linux kernel.

Table 4.2 reports the results of this benchmark, using a CPU with four physical cores and eight logical cores. In the best case, there is basically no difference in using core affinity or not. However, when there are more threads than CPU cores, and their number is not a multiple of the number of cores, then the performance is much worse. This is what happens, for example, when we try to use five threads. Without core affinity, the execution time is 6.2 s, very similar to the one with four threads. However, with core affinity enabled, the running time is 23.9 s, which is worse than the one using only a single thread. As expected, the scheduler of the Linux kernel is *very* good at its job and, given the underwhelming results we obtained, we decided not to use core affinity in Noir.

| Threads | With Core Affinity | Without Core Affinity |
|---|---|---|
| 1 | 22.1 s | 21.6 s |
| 2 | 11.6 s | 11.7 s |
| 3 | 8.3 s | 8.4 s |
| 4 | 6.2 s | 6.5 s |
| 5 | 23.9 s | 6.2 s |

TABLE 4.2: Execution time of an ad-hoc version of the *Wordcount* benchmark, with and without setting the core affinity of each thread.

### 4.9.3 Selection of the Channels Library

In Noir we heavily rely on multi-producer single-consumer channels to make instances communicate with each other on the same host. For this reason, we had to choose the right library that provided good performance, while also having all the features we need, such as selection between multiple channels.

Initially, we relied on the channels provided by the `crossbeam` crate. In fact, its homepage claims that *"This crate is an alternative to std::sync::mpsc with more features and better performance"* [28]. To check this claim, we run the same ad-hoc implementation of the *Wordcount* benchmark of Section 4.9.2, using both the channels provided by `crossbeam` and by the standard library.

| Threads | `crossbeam` | `flume` | `std::sync::mpsc` |
|---|---|---|---|
| 1 | 6.3 s | 6.1 s | 6.2 s |
| 2 | 3.4 s | 3.5 s | 3.3 s |
| 3 | 2.5 s | 2.6 s | 2.4 s |
| 4 | 2.0 s | 2.2 s | 1.9 s |
| 5 | 1.8 s | 1.9 s | 1.9 s |
| 6 | 2.0 s | 1.7 s | 1.7 s |
| 7 | 3.1 s | 1.8 s | 1.7 s |
| 8 | 5.6 s | 2.0 s | 1.7 s |

TABLE 4.3: Execution time of an ad-hoc version of the *Wordcount* benchmark, using either the channels of the `crossbeam` crate, or the ones from `flume`, or of the standard library.

The results of this simple test are presented in Table 4.3. When dealing with few threads, the `crossbeam` crate provides good performance, being on par or faster than the channels of the standard library. However, we got unexpected results when many threads are used: in this case, the standard library performance is still good, while `crossbeam` gets worse the more threads are used. For example, the execution time when using `crossbeam` with eight threads is 5.6 s, which is much worse than the 1.7 s of the standard library. This behavior might be related to the issue #366 on `crossbeam`'s GitHub repository, in which a user claims that *"sending values relatively quickly through a crossbeam channel results in a lot [of] yields and increased CPU usage compared to std::sync::mpsc"* [37].

Given the inconsistent performance of `crossbeam` and that some features needed by Noir are not present in the channels of the standard library, we opted for the `flume` [29] library, as it provides good and consistent performance. It is still possible to use the `crossbeam` channels by passing the right feature flags at compile time.

In Figure 4.17 we can see that, when using `crossbeam`, most of the time is overhead due to the channels. With the other implementations we can see that the majority of the time is spent allocating strings and tokenizing words.

### 4.9.4 Async-await

Since version 1.39, Rust provides a way to define asynchronous functions that can return the control of the execution flow to the runtime when they need to wait

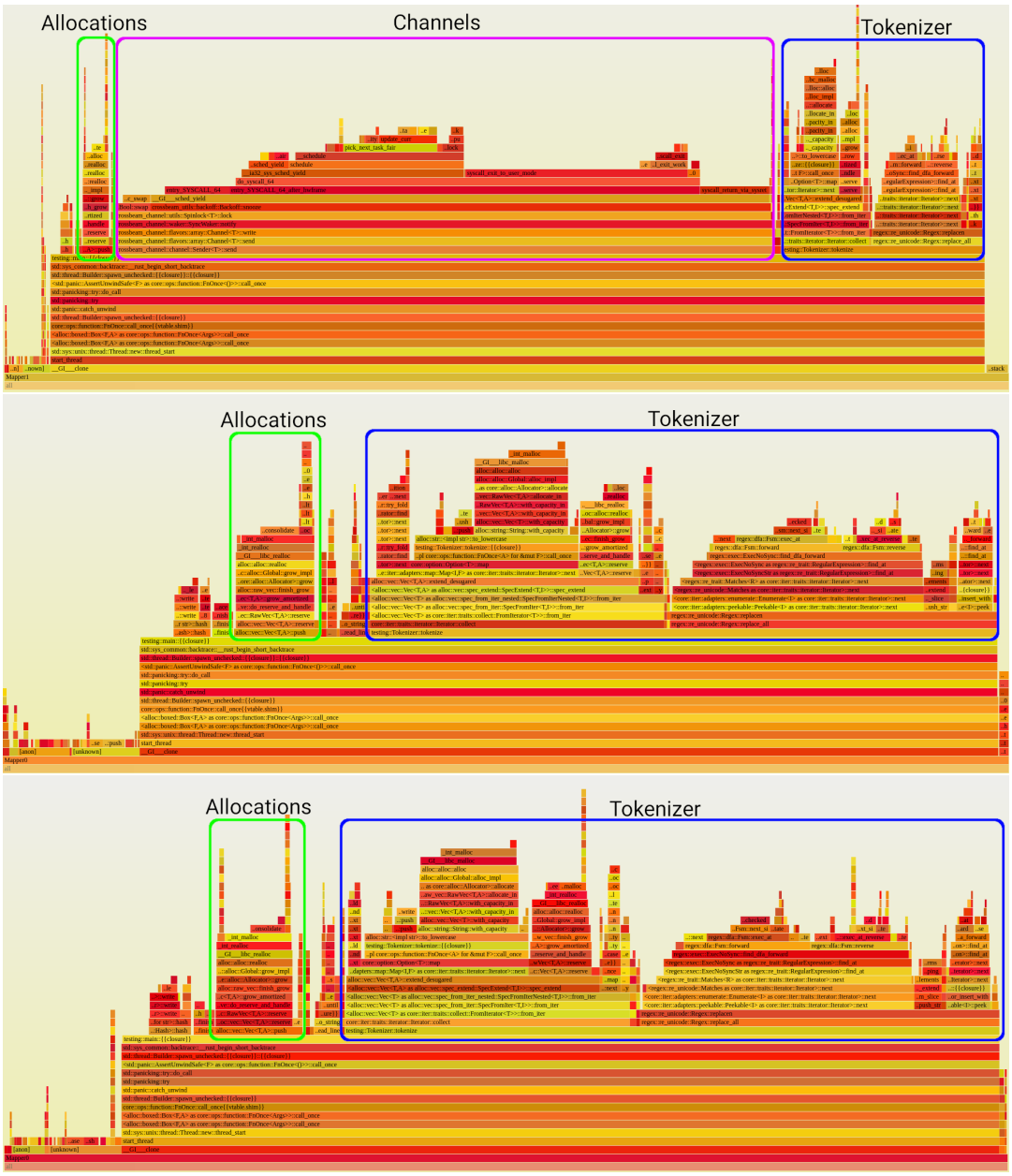



FIGURE 4.17: Flamegraphs of the three channels implementations when using 8 cores (`crossbeam` on the top, `flume` in the middle, `std::sync::mpsc` on the bottom).

for some event to happen. This is mainly used for asynchronous I/O, and it is similar to the `async-await` constructs of languages like JavaScript, even if they behave quite differently. In particular, Rust uses a "poll" model, so that futures can be implemented as *zero-cost abstractions*: functions are not executed until they are `await`-ed.

A prototype of Noir was developed using asynchronous functions for both the network layer and the execution of the operators. However, we were not able to achieve good performance and, given that `async-await` in Rust was still in its early days, we decided not to use this feature.

It might be interesting to investigate an hybrid approach similar to the one used in

RStream, exploiting asynchronous I/O to implement only the network layer, while using blocking functions for the execution of the operators.

### 4.9.5 Batching without threads

Noir supports two *batch modes*, as described in Section 4.4. The *batchers* are the components in charge of handling the batching of the items.

The *Fixed(N)* variant is very simple to implement, since the batcher only needs to keep track of how many tuples are present in the batch under construction. As soon as there are $N$ items in the batch, it can be sent to the recipient instance.

The *Adaptive(N, t)* variant is harder to implement efficiently, as we have to deal with timeouts. Initially, the batchers acted as proxies between the `EndBlock` and the network. They were run using multiple threads, whose only job was to wait for items coming from the corresponding `EndBlock` through in-memory channels, by exploiting the `recv_timeout` facility. However, this required to spawn a lot of threads and to use many in-memory channels, one for each pair of instances that need to communicate. This hindered the performance of the system, as it can be seen from Table 4.4.

| Slots | With Threads | Without Threads | Improvement |
|---|---|---|---|
| 2 | 89.9 s | 40.2 s | 55.3 % |
| 4 | 59.0 s | 21.1 s | 64.2 % |
| 8 | 39.3 s | 13.2 s | 66.4 % |
| 12 | 40.0 s | 11.7 s | 70.1 % |
| 16 | 42.8 s | 11.1 s | 74.1 % |

Table 4.4: Execution time of *Wordcount*, using and not using threads to handle the batching of the items.

One possible solution would be to batch the items exploiting the already existing threads used for handling the TCP streams. However, this meant losing batching when the instances communicate locally using channels.

This problem was solved by making the batcher a part of the `EndBlock` and dealing with the timeout both in the `StartBlock` and in the `EndBlock`. In particular, each time the batcher receives an item, it checks if the timeout has expired; if so, it sends the incomplete batch to the recipient instance. This has a big limitation: it might happen that the batcher receives some items before the expiration of the timeout, and then it does not receive any for a long time. This results in the batch waiting for a long time, longer than the desired timeout indicated by the user. For this reason, also the `StartBlock` checks the expiration of the timeout, by exploiting the `recv_timeout` method on the in-memory channels it uses to receive the messages read from the network. If it does not receive messages for more than the timeout, it sends a control element (`StreamElement::FlushBatch`, see Section 5.5) downstream; it will be received by the batcher, which will notice the timeout has expired and will flush the current batch.

### 4.9.6 MiMalloc

Many allocations and deallocations are done during the execution of a computation. This is the case, for example, when stashing received items inside operators, when reading from and writing to the network, when using hash maps to handle partitioned state and so on. For this reason, we investigated whether changing the default allocator can improve the performance of Noir.

Rust, by default, uses the `System` allocator, which on Linux is based on `malloc` usually from `glibc`. However, starting from version 1.28, Rust also provides an easy way to change the global default allocator (see Listing 6).

```
1 // Change the global default allocator to MiMalloc
2 #[global_allocator]
3 static GLOBAL: mimalloc::MiMalloc = mimalloc::MiMalloc;
```

Listing 6: Change the default allocator to `mimalloc` in Rust.

In particular, we were interested in testing `mimalloc` [8], a very fast allocator that was shown to outperform many other allocators, including `tcmalloc` and `jemalloc`. To evaluate its performance, we run the *Wordcount* and the *Associative Wordcount* benchmarks using both the default allocator and `mimalloc`. Table 4.5 and Table 4.6 shows the results of these two tests.

| Hosts | Cores | Default Allocator | `mimalloc` | Improvement |
|---|---|---|---|---|
| 1 | 8 | 94.2 s | 87.4 s | 7.2 % |
| 2 | 16 | 63.7 s | 56.3 s | 11.6 % |
| 3 | 24 | 47.4 s | 38.5 s | 18.8 % |
| 4 | 32 | 43.4 s | 31.7 s | 27.0 % |

Table 4.5: Execution time of the *Wordcount* benchmark, using the default allocator and `mimalloc`.

| Hosts | Cores | Default Allocator | `mimalloc` | Improvement |
|---|---|---|---|---|
| 1 | 8 | 51.7 s | 49.9 s | 3.5 % |
| 2 | 16 | 27.3 s | 26.6 s | 2.6 % |
| 3 | 24 | 19.0 s | 18.3 s | 3.7 % |
| 4 | 32 | 14.5 s | 14.0 s | 3.4 % |

Table 4.6: Execution time of the *Associative Wordcount* benchmark, using the default allocator and `mimalloc`.

The results show that using `mimalloc` usually improves the performance of the system. As expected, we see bigger improvements in the *Wordcount* benchmark. This is due to the fact that it sends over the network many more tuples than *Associative Wordcount*, thus making more allocations and deallocations.

Noir uses by default the `System` allocator, but `mimalloc` can be chosen by setting the right feature flag during the compilation of the program.

## 4.10 Differences with RStream

Noir and RStream share many design choices, but there are also many important differences both in the architecture and in the API.

**Use of threads** This design choice heavily influenced the architecture of Noir. RStream achieves parallelism in one host by spawning many processes, while Noir uses a single process per-host and instead uses threads. Threads allow the use of better synchronization primitives, such as in-memory channels, locks and atomic variables. With those primitives Noir is able to support more advanced operators (such as joins and more advanced iterations), keeping comparable performance.

Moreover, having a single address space (i.e. not using multiple processes) helps in debugging and measuring performance. With RStream assessing the performance of the various blocks is tricky. For example, using `perf` [30] it is possible to track the impact of a single process, but aggregating these metrics it is not an easy task. With Noir it is enough to run each process under `perf` and all the threads are automatically aggregated, making it a lot easier to measure the system's performance and to spot the bottlenecks.

**In-memory channels** RStream runs each block in a separate process, therefore inter-process communication channels (such as TCP) are required. Those channels are usually much slower than the in-memory counterpart since they require each message to be serialized and then deserialized to cross the address space boundary. Furthermore, sending a message requires a syscall and the serialized message may be copied in kernel space, and it may need to traverse the network stack.

Noir is able to exploit in-process channels that avoid all those overheads. By using the network abstraction described in Section 4.6, the implementation of the operators is not impacted, and the performance is significantly improved.

**More advanced scheduler** By using threads instead of processes, it is possible to spawn a dynamic number of workers inside each host. RStream is limited to spawning as many processes as slots defined in its configuration file. Instead, Noir is able to use a much more sophisticated scheduler that is aware of the various hosts and their connections. It also can perform some analysis on the Job Graph to determine where to allocate each block.

Having this scheduler lifts the user the need to compute manually the number of slots of each host. In RStream the number of slots to write in the configuration file is a function of the host, the number of blocks in the Job Graph and whether the host contains sources or sinks. In Noir the user just needs to know how many CPU cores there are in each host.

```yaml
hosts:
- address: host1
  base_port: 9500
  num_cores: 8
- address: host2
  base_port: 9500
  num_cores: 8
- address: host3
  base_port: 9500
  num_cores: 8
```

```yaml
---
- hostname: host1
  slots: 9  # 8 instances + 1 source
- hostname: host2
  slots: 8  # 8 instances
- hostname: host3
  slots: 9  # 8 instances + 1 sink
```

The configuration file for Noir is on the left, the one for RStream is on the right.
Consider a stream with a single non parallelizable source, followed by a single block, and then a single non parallelizable sink. There are 3 hosts with 8 cores each.

**Concurrent streams** Noir supports a vast variety of Job Graph topologies, including ones where there is more than one independent stream running in parallel. RStream is very limited under this aspect, it allows Job Graphs that are either linear (i.e. no branches) or cyclic (i.e. the graph is a single loop). Therefore, RStream is not able to support many advanced operators (e.g. joins, nested iterations, ...) by design.

**Richer API** With a more advanced scheduler and the support for concurrent streams, Noir offers many more operators than RStream and a much richer API. Noir supports a vast range of join operators (see Section 5.6.7), better iterations with support for nested loops and the possibility to keep state between different loop iterations (see Section 5.6.8), many more aggregators, concurrent streams operators (such as zip, split, concat, see Section 5.6.10) and rich operators (see Section 5.6.3).

**More modular API** While the APIs of RStream are pretty vast, their design does not scale very well. For example, the window API couples the window definition (e.g. counting, event time, sliding, tumbling) with the aggregation function (e.g. reduce, map). In particular, each possible window variant is exposed as a different function: `sliding_count_reduce`, `tumbling_count_reduce`, `tumbling_e_time_reduce`, and so on. Noir tries to be more modular, separating the APIs for defining a window from the APIs defining the aggregation function:

```
stream
    .window(CountWindow::sliding(10, 3))
    .map(|window| ...)
```

# 5 API

This chapter outlines the Application Programming Interface (API) of Noir, first by defining some important concepts and data types, and then by illustrating the most important operators provided.

## 5.1 Environment

Before being able to define and start the computation, the user must create an *execution environment*, which is used to maintain all the information needed to schedule and run the user's task. The environment is used to keep track of all the blocks and operators present in the program, including how they are connected and instantiated. The creation of the network topology is done on every host, so that each of them is able to spawn the necessary operators and to connect to the other hosts.

To create an environment, the user has to pass some configuration to the environment constructor. This configuration describes whether the program will be executed locally or remotely on multiple machines. It is also used to enable or disable the debugging features provided by Noir, such as timing reports, logs, and whether to collect profiling data used by the visualizer. As described more precisely in Section 4.8, the environment's configuration is either specified by using some command line arguments, read from a file on disk or from the environment variables of the process.

```
1  // Create an `EnvironmentConfig` from the command line arguments
2  let (config, _args) = EnvironmentConfig::from_args();
3  // Create the execution environment
4  let mut env = StreamEnvironment::new(config);
5  // Spawn the workers on the remote hosts
6  env.spawn_remote_workers();
7  // Define one or more streams with env.stream(...)
8  [...]
9  // Execute the program
10 env.execute()
11 // Finally, read the results produced by the execution
12 [...]
```

Listing 7: Creation of an execution environment.

Note that the environment works as a kind of barrier between the graph definition, the execution of the program and the retrieval of the results. In particular, the graph of the computation can be defined only after the creation of the environment, but before the call to the `execute` method. This method starts the computation and blocks until all the operators have stopped. Finally, the results can be retrieved only after the `execute` method has completed. Listing 7 provides an example of this workflow.

## 5.2 Stream Types

After the creation of an environment, the user can define new streams. In particular, the `stream` method of the environment creates a new `Stream` from a given source (see Listing 8).

```
1 // Define a new source
2 let source = ...;
3 // Create a new `Stream`
4 let stream = env.stream(source);
```

Listing 8: Stream creation from a source.

There are four different stream types in Noir:

**Stream** the simplest type of stream, it is not partitioned and it does not have any window applied to it.

**KeyedStream** a stream whose elements are partitioned by key; this is the type of stream returned, for example, by the `group_by` operator.

**WindowedStream** a stream whose elements are grouped into windows; this is the stream returned by the `window_all` operator.

**KeyedWindowedStream** a stream whose elements are partitioned by key, but they are also grouped into windows; this is the stream returned by the `window` operator, when called on a `KeyedStream`.

Each stream type has many methods, which represent the different operators that can be applied to it. When applying one of the operators, the stream is consumed and a new one is returned.

## 5.3 Data traits

In Noir, the data tuples of a stream are not limited to a given set of types, but they can be whichever custom type the user wants. However, these custom types must implement some specific traits, so that values can be moved between different instances, both locally or using the network.

Items that need to be moved from one operator to the next one locally, without using the network, must implement the `Data` trait. The `ExchangeData` trait, instead, is used for items that need to be sent over the network: for this reason, it is a subtrait

of `Data`, but it also requires traits used for the serialization and the deserialization of the items.

Instead, when dealing with partitioned streams, the keys' types must implement the `DataKey` trait: it is a subtrait of `Data`, but it also requires that the instances of these types can be hashed and that they can be compared one with the other, so that we are able to find out if two keys are equal or not. When keys need to be sent over the network, then they must implement the `ExchangeDataKey` trait.

The formal definitions of these traits are shown in Listing 9.

```
pub trait Data: Clone + Send + 'static {}
impl<T: Clone + Send + 'static> Data for T {}

pub trait ExchangeData: Data + Serialize + for<'a> Deserialize<'a> {}
impl<T: Data + Serialize + for<'a> Deserialize<'a> + 'static>
    ExchangeData for T {}

pub trait DataKey: Data + Hash + Eq {}
impl<T: Data + Hash + Eq> DataKey for T {}

pub trait ExchangeDataKey: DataKey + ExchangeData {}
impl<T: DataKey + ExchangeData> ExchangeDataKey for T {}
```

Listing 9: Definition of the `Data`, `ExchangeData`, `DataKey` and `ExchangeDataKey` traits.

## 5.4 Operator trait

As seen in Chapter 2, operators are the basic building blocks used to describe the computation that needs to be performed by the system. In Noir, operators are defined by implementing the `Operator` trait (Listing 10), which is used to describe how the operator behaves, that is how it transforms the input stream into the output one. This trait is usually implemented on an appropriate `struct`, which maintains all the information regarding the operator, including its state, the parameters defined by the user (e.g. closures) and its operators chain.

```
pub trait Operator<Out: Data>: Clone + Send {
    fn setup(&mut self, metadata: ExecutionMetadata);
    fn next(&mut self) -> StreamElement<Out>;
    fn to_string(&self) -> String;
    fn structure(&self) -> BlockStructure;
}
```

Listing 10: Definition of the `Operator` trait.

First of all, the `Operator` trait has a generic type parameter `Out`, which implements the `Data` trait: this is the type of the elements of the stream. For example, if we

are dealing with a stream of strings, then `Out` will be `String`. This generic type parameter is needed to have a single definition of the `Operator` trait that can be used for any kind of stream.

The `Operator` trait also defines few methods that need to be implemented by each operator. As the name implies, the `setup` method is called before the start of the computation, and it is used to configure the operator. It receives as parameter some metadata associated with the operator's block, including the coordinates of the current instance, the list of instances that execute this same block and a global identifier of the current instance. This information can be used to make the operator behave differently based on which instance it is running on. This is useful, for example, when reading from a file parallelly: by using the global identifier contained in the metadata, the different instances of the operator can read different chunks of the file. Finally, this method must always propagate the execution metadata backward in the chain of operators, by recursively calling the `setup` method on the previous operators. This means that operators in a block are configured from the bottom up, starting from the `EndBlock` operator up to the `StartBlock` one.

The actual logic of the operator is implemented in the `next` method, whose functioning is heavily inspired by the `Iterator` trait in the Rust standard library [22]. Each time this method is called, it returns the next element in the operator's output stream. To do so, the operator can of course call the `next` method recursively on the previous operator. This starts a chain of method calls that reaches the `StartBlock`, which will return elements read from the network and coming from other blocks. Note that it is not mandatory to have a one-to-one mapping from incoming to outgoing elements: when the `next` method is called, the operator can either return elements that were stashed in previous calls to that method, or it can call the `next` method on the previous operator one or more times, even up to the end of the stream. This provides a great flexibility in implementing both simple stateless operators that just need a single element at a time or stateful operators that need the whole stream to produce some output (e.g. operators that reduce the stream to a single value).

Finally, the `to_string` and `structure` methods are used for debugging and visualization purposes. The former returns a textual description of the operator, which is used to represent the operator in the logs of the system. The latter returns more structured information about the block, which is used by the visualizer (see Appendix B).

For a sample implementation of a simple `map` operator, see Appendix A.

## 5.5 StreamElement

When dealing with a stream of a particular type, for example a stream of integers or strings, one would expect that the elements generated by the operators have the same type as the ones of the stream. However, looking closely at the definition of the `Operator` trait in Listing 10, the `next` method returns a value of type `StreamElement<Out>` instead of `Out`, which is the type of the stream's data tuples. This is needed because there is some additional information other than the actual data tuples that need to be preserved and propagated downstream.

In particular, `StreamElement<Out>` is an enumeration with the following variants:

`Item(Out)` a simple data tuple of type `Out`.

`Timestamped(Out, Timestamp)` a data tuple of type `Out`, with its associated timestamp. This is used when dealing with events in either event time or processing time domain (see Section 2.4).

`Watermark(Timestamp)` a watermark with its associated timestamp (see Section 2.5).

`FlushBatch` when using the adaptive batch mode, this control element indicates that the batch needs to be flushed, even if it is not full yet (see Section 4.4).

`FlushAndRestart` is a control element used to signal that the stream has ended and that each operator needs to restore their state to the initial one. This is also used to mark the end of an iteration (see Section 4.7).

`Terminate` after receiving a `FlushAndRestart` element, this control element indicates that the computation has ended and that each operator should stop as soon as possible.

Using the *regular expression* notation[1], the elements of a stream can be described as follows:

$$((\texttt{Item} \mid \texttt{Timestamped} \mid \texttt{Watermark} \mid \texttt{FlushBatch})^* \ \texttt{FlushAndRestart})^+ \ \texttt{Terminate}$$

So, the stream is a non-empty sequence of sequences of `Item`, `Timestamped`, `Watermark`, and `FlushBatch`, each followed by a `FlushAndRestart`, and all terminated with a `Terminate` control element.

## 5.6 Operators

In this section we will describe the main operators provided by Noir. These include sources, simple stateless operators, operators that handle the partitioning and aggregation of the stream, windows, joins, iterations and sinks.

### 5.6.1 Sources

Sources are special operators that are needed to create new streams. For this reason, the `Source` trait extends the `Operator` trait, requiring the additional method `get_max_parallelism` (see Listing 11). This method should return the maximum number of instances of the source that the system can spawn. This is needed, for example, when dealing with non-parallelizable sources like `IteratorSource`.

Once a source is defined, a stream can be created with the help of the `stream` method of the environment (see Listing 8).

[1] $a|b$ means one of $a$ or $b$, $a^*$ means zero or more times $a$, and $a^+$ means one or more times $a$.

```rust
pub trait Source<Out: Data>: Operator<Out> {
    fn get_max_parallelism(&self) -> Option<usize>;
}
```

Listing 11: Definition of the `Source` trait.

### Iterator Sources

In Rust, iterators are used to abstract the concept of a series of items. Given that iterators are very similar to a stream of data, in Noir it is possible to use them as sources (see Listing 12). However, iterators are usually non-parallelizable, so by default `IteratorSource` will not be instantiated multiple times.

When a parallel source is needed, then `ParallelIteratorSource` can be used. Instead of accepting an iterator, this source requires a closure that, given the identifier of a instance and the total number of instances of the source, returns the appropriate iterator (see Listing 13).

It is also possible to implement custom sources by simply implementing custom iterators that, for example, read the data from an external system or resource.

```rust
// In this example, the iterator used by the source is `(0..100)`.
// This will generate all the integers between 0 and 100 (excl.).
let source = IteratorSource::new((0..100));
```

Listing 12: Example of `IteratorSource`.

```rust
// This source will generate all the integers between 0 and 100
// (excluded), using multiple instances.
let n = 100;
let source = ParallelIteratorSource::new(move |id, num_instances| {
    let chunk_size = (n + num_instances - 1) / num_instances;
    let remaining = n - n.min(chunk_size * id);
    let range = remaining.min(chunk_size);

    let start = id * chunk_size;
    let stop = id * chunk_size + range;
    start..stop
});
```

Listing 13: Example of `ParallelIteratorSource`.

### File Source

Noir provides a source that reads data from a file. In this case, each data tuple produced by the source is a single line of the file. The file is divided in multiple chunks, and it is read in parallel by many instances. For this reason, the file must be available on every host, at the same location.

```rust
// In this example, the source will read the file "/data.txt"
let source = FileSource::new("/data.txt");
```

Listing 14: Example of `FileSource`.

#### CSV Source

Just like `FileSource`, this source reads a CSV file in parallel. However, each record in the file is deserialized into a user-provided type. This source supports multiple customizations, including specifying whether the CSV file has a header row or setting a custom delimiter character (see Listing 15).

```rust
// Custom type used to deserialize the records
#[derive(Clone, Deserialize, Serialize)]
struct MyType {
    value: String,
    count: u64,
}

// In this example, the source will read the file "/data.csv",
// deserializing each record into the `MyType` type
let source = CsvSource::<MyType>::new("/data.csv")
    .delimiter(b';')
    .has_headers(false);
```

Listing 15: Example of `CsvSource`.

### 5.6.2 Simple operators

After defining a source and constructing a stream, it is possible to apply multiple operators to the stream, in order to achieve the desired computation graph. In this section, we describe the most simple operators that Noir offers, which are the stateless operators.

These operators process one element at a time, using the user-provided closures. Being stateless operators, they only accept closures which implement the `Fn` trait, that is to say closures which only take immutable references to captured variables. When these operators are applied to a keyed stream, a tuple with both the key and the value of each element is passed to the closure. However, the key is passed as an immutable reference, since changing the key might result in a stream where elements are not partitioned by key anymore. Due to their simplicity, these operators do not need to move data between different instances and, for this reason, they can be added to an already existing block of operators, without the need to create a new one.

### Map

The `map` operator is one of the most used operators. As the name implies, this operator is used to "map", that is to "translate", each incoming data tuple into a new one. In order to do so, the `map` operator accepts one closure as a parameter, with a single input and a single output. This closure is called repeatedly on every incoming element, and the outgoing stream is composed of all the results returned by it. Listing 16 shows an example in which each element of the stream is multiplied by ten.

```
// `map` operator for `Stream`
stream.map(|val| val * 10);

// `map` operator for `KeyedStream`
stream.group_by(...).map(|(_key, val)| val * 10);
```

Listing 16: Examples of usage of the `map` operator.

### Filter

The `filter` operator is used to remove some elements from the stream. Just like the `map` operator, it takes a closure as parameter. However, this closure must return a boolean: the function is called once for each element, and that element is removed from the stream if the closure evaluates to `false`. Listing 17 shows how to keep only the multiples of three from a stream of numbers.

```
// `filter` operator for `Stream`
stream.filter(|&n| n % 3 == 0);

// `filter` operator for `KeyedStream`
stream
    .group_by(...)
    .filter(|&(_key, n)| n % 3 == 0);
```

Listing 17: Examples of usage of the `filter` operator.

### Flatten

In some cases, the items of a stream might be collections of values, for example when elements are vectors of numbers or strings. The `flatten` operator is used to remove one level of indirection, which means that the resulting stream will be composed of all the values contained in all the elements of the incoming stream (see Listing 18).

```
// `flatten` operator for `Stream`
stream              // Stream<Vec<String>>
   .flatten();      // Stream<String>

// `flatten` operator for `KeyedStream`
stream.group_by(...).flatten();
```

Listing 18: Examples of usage of the `flatten` operator.

### FilterMap

The `filter_map` operator is used to fuse a `map` and a `filter` operation together. The closure used by this operator must return an `Option::Some` that contains the translated value, or `Option::None` if the element needs to be discarded. As an example, in Listing 19 odd numbers are removed from the stream and even numbers are multiplied by three.

```
// `filter_map` operator for `Stream`
stream
    .filter_map(|n| if n % 2 == 0 { Some(n * 3) } else { None });

// `filter_map` operator for `KeyedStream`
stream
    .group_by(...)
    .filter_map(|(_, n)| if n % 2 == 0 { Some(n * 3) } else { None });
```

Listing 19: Examples of usage of the `filter_map` operator.

### FlatMap

One of the main limitations of the `map` operator is that each incoming element must be transformed into exactly one output element. Instead, the `flat_map` operator can be used when we need to return zero, one or more elements from a single incoming element in the stream. This is achieved by fusing together a `map` with a `flatten` operation: the former returns a collection of values derived from a single element, which is then flattened by the latter. The `flat_map` operator can be used, for example, to duplicate each element of the stream (see Listing 20).

```
// `flat_map` operator for `Stream`
stream.flat_map(|n| vec![n, n]);

// `flat_map` operator for `KeyedStream`
stream
    .group_by(...)
    .flat_map(|(_key, n)| vec![n, n]);
```

Listing 20: Examples of usage of the `flat_map` operator.

### 5.6.3 Rich operators

Sometimes, simple stateless operators like `map` are not powerful enough to tackle all the tasks we would like to execute. In some of these cases, we can use the so-called "rich" operators, which are equivalent to the `map`, `flat_map` and `filter_map` operators, except that they can have a state that is local to each instance.

State is handled automatically by the fact that these operators accept closures that implement the `FnMut` trait, which is less restricting than the `Fn` trait required by the simple operators seen in Section 5.6.2. This means that closures can capture user defined variables by value, instead of by reference, which can be used to maintain the necessary state. The closures must also implement the `Clone` trait, so that they can be cloned once for each instance or for each partition of the stream.

As stated before, the state is local to each instance, and it is not shared between them, so sometimes it might be necessary to change the parallelism of the operator in order to get the expected results (see Section 5.6.11). For example, `rich_map` can be used to implement an operator that calculates the prefix sums of the elements in the incoming stream (see Listing 21). This can be done by keeping the partial sum of all the elements processed since the beginning of the stream in a state variable, updating it when new elements are received. However, if this operator is instantiated multiple times, each instance would receive and process only a subset of all the elements and the results would be wrong. This problem is not present when dealing with a keyed stream, because each partition has its own private state, and all the elements with a given key are processed by the same instance.

In general, rich operators should be avoided if the computation can be carried out by using stateless operators, as they are generally faster. When working with keyed streams, for example, each partition needs to have its own private state. This means that the user-provided closure needs to be cloned once for each key and stored in a hash map, so that elements with the same key are always processed by the same closure. This introduces a lot of overhead with respect to stateless operators, which can share the closure between different partitions of the stream.

```
1 // `rich_map` operator which calculates the prefix sums
2 stream.rich_map({
3     let mut sum = 0;
4     move |x| {
5         sum += x;
6         sum
7     }
8 });
```

Listing 21: Example of usage of the `rich_map` operator.

```rust
// `rich_filter_map` operator which returns only the positive
// prefix sums
stream.rich_filter_map({
    let mut sum = 0;
    move |x| {
        sum += x;
        if sum >= 0 {
            Some(sum)
        } else {
            None
        }
    }
});
```

Listing 22: Example of usage of the `rich_filter_map` operator.

```rust
// `rich_flat_map` which generates all the possible pairs of
// incoming elements
stream.rich_flat_map({
    let mut elements = Vec::new();
    move |y| {
        let new_pairs = elements
            .iter()
            .map(|&x: &u32| (x, y))
            .collect::<Vec<_>>();
        elements.push(y);
        new_pairs
    }
})
```

Listing 23: Example of usage of the `rich_flat_map` operator.

### 5.6.4 Partitioning

In this section we describe operators that are used to shuffle data between different instances. Since they need to transfer data between instances using the network, they need to create a new block (see Section 4.3).

#### Shuffle

The `shuffle` operator is used to randomly distribute the elements of a stream between all the different instances of a block. This can be useful when dealing with an operator that creates an uneven quantity of tuples on different instances, for example when dealing with unbalanced partitioning or join. They also come in handy when using operators that cannot be instantiated multiple times. By applying the `shuffle` operator, it is possible to evenly distribute the data coming from these operators between all the hosts, thus restoring the maximum level of parallelism that the cluster can offer.

```
// shuffle the data between all the different instances
stream.shuffle()
```

Listing 24: Example of usage of the `shuffle` operator.

### GroupBy

The `group_by` operator is used to partition the stream into logically independent substreams. Elements are divided based on a *key*, which is a special value that can be extracted from every element; elements that have the same key are grouped together and are sent to the same instance. The key is extracted from each element by using a user-provided closure. This operator can be used to apply other operators, such as aggregations and reductions, independently on each partition. The `group_by` operator can be applied to a `Stream`, and it returns a `KeyedStream`.

As an example, in Listing 25, elements are divided into groups based on their parity and then summed together. Thus, the resulting stream will contain two elements, the sum of the even numbers and the sum of the odd numbers. Note that since the sum is an associative operation, the associative variant of the `fold` operator (`group_by_fold`) is much more efficient (it is described later in this section).

```
// divide the numbers in two groups based on their parity
// and calculate their total value
stream
    .group_by(|&n| n % 2)
    .fold(0, |acc, value| *acc += value);
```

Listing 25: Example of usage of the `group_by` operator.

### KeyBy

The `key_by` operator is very similar to the `group_by` one, with the exception that it does not actually send tuples with the same key to the same instance. This avoids splitting the block and using the network, but the user must make sure that the data tuples are already partitioned correctly on the various instances. This means that, if the user is not careful, the resulting `KeyedStream` might be malformed and behave unexpectedly. In Listing 26, the tuples are already partitioned correctly thanks to the usage of `group_by`, so the keyed stream will work as intended after using the `key_by` operator.

```
// divide the numbers in two groups based on their parity
stream
    .group_by(|&n| n % 2)
    .unkey()                // simply forgets about the key
    .key_by(|&n| n % 2);
```

Listing 26: Example of usage of the `key_by` operator.

### 5.6.5 Aggregations

As suggested by their name, aggregators are used to accumulate the values of a stream into a single element. These are useful to compute aggregated results over a dataset, such as the average or the total of a given set of values.

#### Fold

The `fold` operator is used to aggregate all the elements of a stream into a single value, which can be of any type. This operator requires two parameters: the initial value of the accumulator and the closure used to perform the aggregation. The closure is executed on each item of the stream, passing both the value and a mutable reference to the current partial accumulator. The closure updates the accumulator with the value of the item. After processing all the elements, `fold` emits the accumulator. In Listing 27, the `fold` operator is used to sum all the numbers in a stream.

Note that using the `fold` operator usually means that many elements are transferred over the network. In fact, if it is applied to a non-keyed stream, then the operator has only one instance, which receives all the elements of the stream. Instead, if it is used on a keyed stream, then the elements are first partitioned and sent to the correct instance based on their key, and then each partition is aggregated independently. If the folding operation is associative, then it is better to use the associative variants that will be presented later.

```
1 // sum all the numbers in the stream
2 stream.fold(0, |acc, value| *acc += value);
3
4 // sum all the numbers in each partition of the stream
5 stream
6     .group_by(...)
7     .fold(0, |acc, value| *acc += value);
```

Listing 27: Examples of usage of the `fold` operator.

#### Reduce

Sometimes, a stream has to be reduced into a value that has the same type of the data tuples of the incoming stream. When this happens, it is usually easier to use the `reduce` operator, instead of `fold`. This is due to the fact that the `reduce` operator does not require an initial value to perform the aggregation. In particular, if the stream contains only one element, then that element is returned. Instead, if it contains two or more elements, then the first two of them are passed to the user-provided closure to compute the first partial result. From the third element onward, the `reduce` operator behaves exactly as `fold`. Listing 28 implements the same computation of Listing 27, using the `reduce` operator instead of `fold`.

Just like the `fold` operator, also `reduce` usually requires to move a lot of data tuples between the different instances. Also in this case, if the reducing operator is associative then using the associative variants described later usually results in better performance.

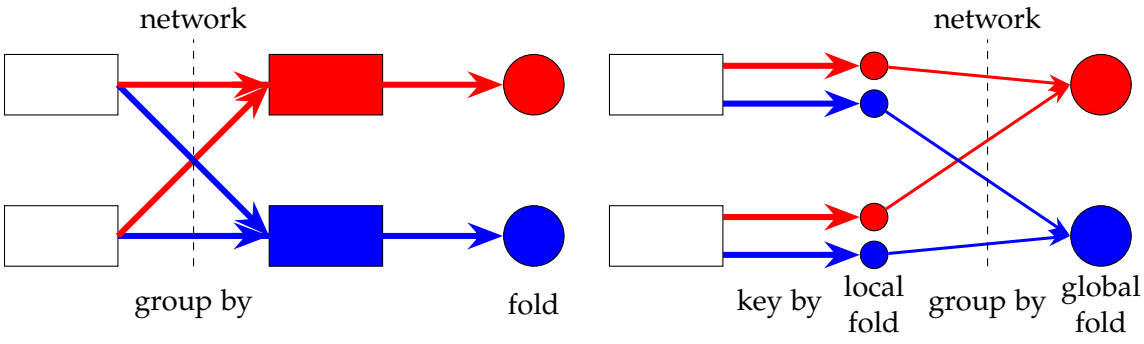
```
// sum all the numbers in the stream
stream.reduce(|acc, value| *acc += value);

// sum all the numbers in each partition of the stream
stream
    .group_by(...)
    .reduce(|acc, value| *acc += value);
```

Listing 28: Examples of usage of the `reduce` operator.

### Associative variants of Fold and Reduce

When the aggregating operation is associative, the `fold` and `reduce` operators can be implemented much more efficiently. In particular, considering both keyed and non-keyed streams, the elements can be partially accumulated before being sent over the network. Then, these partial results will be combined into a single value by the receiving instance. This greatly reduces the network usage and, in the case of non-keyed streams, also provides a greater degree of parallelism. In fact, only the partially reduced values are sent over the network; Figure 5.1 shows how this is achieved.




(a) `group_by(...).fold(...)` (b) `group_by_fold(...)`

Figure 5.1: The two variants of `group_by` and `fold`, (b) is the associative version of (a). Rows represent different instances, while colors different partitions. Rectangles represent where other operators can be placed.

When considering `reduce`, its usage does not really change, except that for non-keyed streams `reduce_assoc` is used instead of `reduce`. Considering keyed streams instead, `group_by(...).reduce(...)` is replaced by the single operator `group_by_reduce`. Listing 29 shows the same computation of Listing 28 but using the associative variant of the `reduce` operator.

Instead, with the `fold` operator things get a bit more complicated. Due to the fact that the resulting accumulator can have a generic type, the user also has to describe how two partial results can be accumulated into one. For this reason, the associative variants of `fold` accept two closures, instead of just one. In any case, when dealing with non-keyed streams, `fold` is replaced by `fold_assoc`. Instead, when considering keyed streams, the operator `group_by_fold` is used instead

of `group_by(...).fold(...)`. Listing 30 uses the associative variants of `fold` to implement the same computation of Listing 27.

```
// sum all the numbers in the stream, associatively
stream.reduce_assoc(|acc, value| *acc += value);

// sum all the numbers in each partition of the stream, associatively
stream
    .group_by_reduce(..., |acc, value| *acc += value);
```

Listing 29: Examples of usage of the associative variants of the `reduce` operator.

```
// sum all the numbers in the stream, associatively
stream.fold_assoc(
    0, // initial value
    // aggregate elements into partial results
    |acc, value| *acc += value,
    // combine two partial results into one
    |acc1, acc2| *acc1 += acc2
);

// sum all the numbers in each partition of the stream, associatively
stream.group_by_fold(
    ..., // key function
    0,   // initial value
    // aggregate elements into partial results
    |acc, value| *acc += value,
    // combine two partial results into one
    |acc1, acc2| *acc1 += acc2
);
```

Listing 30: Examples of usage of the associative variants of the `fold` operator.

### Common Aggregations

While `fold` and `reduce` are very flexible and they can be used to define many different computations, there are some common aggregations that are usually present in many programs. For this reason, Noir provides shortcuts to compute the minimum element, the maximum element, the sum, the average value and the number of the items of a keyed stream.

All these operators can be applied to a non-keyed stream and they accept a closure to compute the key of each tuple. Except for `group_by_count`, all the other operators also require a closure to extract a value from each item: this value will be used to compute the minimum element, the maximum element, the sum or the average value (see Listing 31). Note that this operators are implemented using the associative variants of the `fold` and `reduce` operators.

```rust
// Find the minimum element of each partition of the stream
stream.group_by_min_element(..., |&n| n);

// Find the maximum element of each partition of the stream
stream.group_by_max_element(..., |&n| n);

// Find the sum of all the elements of each partitions of the stream
stream.group_by_sum(..., |&n| n);

// Find the mean value of the elements of each partition of the stream
stream.group_by_avg(..., |&n| n as f64);

// Count the number of elements of each partition of the stream
stream.group_by_count(...);
```

Listing 31: Examples of usage of some common aggregators.

### 5.6.6 Windows

Windows are used to divide a stream into finite groups that evolve over time, which are then processed independently. This is needed when dealing with unbounded streams, since blocking operators like aggregations and joins need to consume all the data to complete their computations. This is clearly a problem when the stream is infinite and when dealing with low-latency constraints, as the operator would never generate any output. In case of finite streams, instead, windows are optional: if not used, the processing will be equivalent to those of batch processing systems.

There are two different policies used by windows to divide elements into groups: based on their timestamp (time-based windows) and based on the tuples arrival order (tuple-based windows). Note that, if windows are overlapping, each tuple can end up in multiple groups. In general, each tuple might also not be part of any window, but this case is rarely interesting as some data is lost. Windows can be *aligned* if the same window is applied to the whole dataset or *unaligned* if there are different windows for different subsets of the stream, for example per key after a grouping operation.

Time-based windows group elements by their timestamp, either in the event time domain or in the processing time domain. There are three different types of time-based windows:

**Sliding Windows** (Figure 5.2) are aligned windows that are defined by a *size* and a *step* period. This means that each window has the same size, and it starts *step* amount of time after the previous one. Given that the step period can be shorter than the size of the window, these windows may be overlapping.

**Tumbling Windows** (Figure 5.3) sometimes called fixed windows, are aligned windows that are defined by a fixed length that determines both the *size* and the *step* period of the window. For this reason, tumbling windows are non overlapping.

**Session Windows** (Figure 5.4) are unaligned windows that are defined by a *gap* period. They group elements based on their activity: two consecutive elements belong to the same window if they are at most a *gap* amount of time apart. Like tumbling windows, session windows are non overlapping.

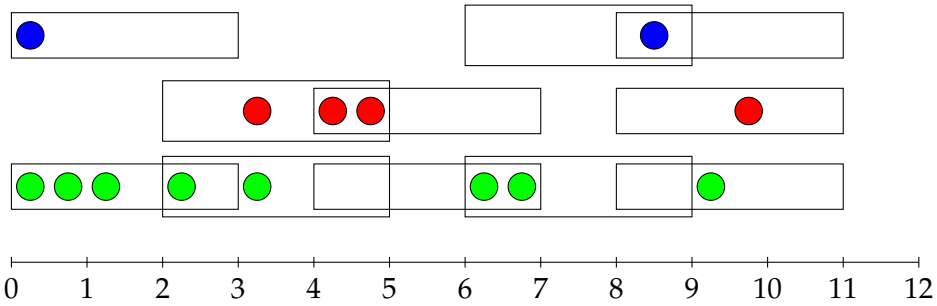



FIGURE 5.2: Sliding Window with *size* 3 and *step* 2.

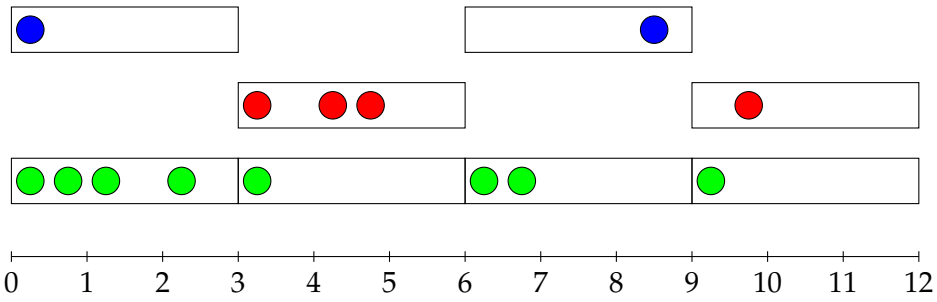



FIGURE 5.3: Tumbling Window with *size* 3.

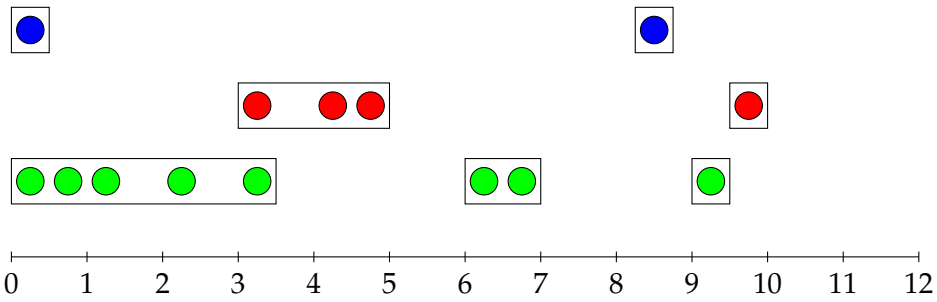



FIGURE 5.4: Session Window with *gap* 2.

Tuple-based windows divide elements based on their arrival order. They are sometimes called count windows, since they group elements by counting them. There are two different types of counting windows:

**Sliding Count Windows** (Figure 5.5) are defined by a *size* and a *step* length. The first window contains the first *size* elements and from then on each window contains the same elements of the previous one, except that the oldest *step* elements are replaced with new ones. These windows may be overlapping.

**Tumbling Count Windows** (Figure 5.6) are defined by a *size* length. They are equivalent to sliding count windows where the *size* is equal to the *step* length. These windows are non overlapping.

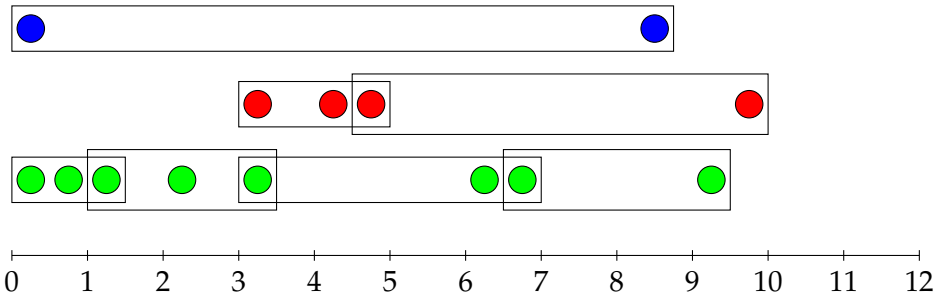



FIGURE 5.5: Sliding Count Window with *size* 3 and *step* 2.

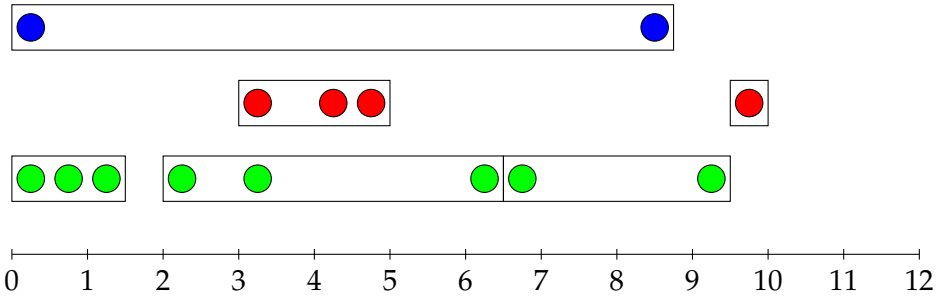



FIGURE 5.6: Tumbling Count Window with *size* 3.

Noir supports windowing on both `Stream` and `KeyedStream`. With the latter the window is applied on a key-by-key basis, while on the former all the tuples are considered together. To do so, on a `Stream` the operator to use is `window_all`, which cannot be parallelized.

```
// Count windows
CountWindow::sliding(3, 2)
CountWindow::tumbling(3)

// Event time windows
EventTimeWindow::sliding(Duration::from_secs(120),
↪  Duration::from_secs(10))
EventTimeWindow::tumbling(Duration::from_secs(120))
EventTimeWindow::session(Duration::from_secs(120))

// Processing time windows
ProcessingTimeWindow::sliding(Duration::from_secs(120),
↪  Duration::from_secs(10))
ProcessingTimeWindow::tumbling(Duration::from_secs(120))
ProcessingTimeWindow::session(Duration::from_secs(120))
```

LISTING 32: The supported windowing types to be passed to the `window` operator.

After defining the windowing criteria, the next operator to use is an aggregation. On windowed streams the supported operators are `first`, `fold`, `join`, `map`, `max`, `min`, and `sum`.

The operators `max`, `min`, and `sum` are supported only if the type of the tuples support

the corresponding operation; each window is reduced to a single tuple using the corresponding mathematical operator.

```
stream
    .group_by(...)
    .window(CountWindow::tumbling(2))
    .max()
```

Listing 33: Examples of usage of the `window` operator for a *Tumbling Count Window* of size 2, whose aggregation operator is `max`.

When using `map`, the provided closure is called with a reference to the items of each window, and it returns the new item to be emitted.

```
stream
    .group_by(...)
    .window(CountWindow::tumbling(2))
    .map(|window| *window.last().unwrap())
```

Listing 34: Examples of usage of the `window` operator for a *Tumbling Count Window* of size 2, whose aggregation operator is `map`.

Given two streams windowed in the same manner, the `join` produces the Cartesian product of the two streams, on a window-by-window basis.

```
let keyed1 = stream1.group_by(...);
let keyed2 = stream2.group_by(...);
keyed1
    .window(EventTimeWindow::tumbling(Duration::from_millis(2)))
    .join(keyed2)
```

Listing 35: Examples of usage of the `window` operator for a *Tumbling Event Time Window* of size 2 ms that joins two streams.

It is important to note that the elements produced by a window are timestamped as well. This makes it possible to chain windows, potentially with different windowing criteria. Chaining is especially useful when windows are aligned, so that they refer to the same range of events of the original stream. With time-based windows, the timestamp of each element produced is the ending time of the corresponding window. Considering instead tuple-based windows, the timestamp is the maximum one of items contained in the window, adjusted if watermarks with greater timestamps have already passed.

### 5.6.7 Join

Join transformations are used to combine two streams, which are usually called *left* and *right* stream, into a stream composed of pairs of values. These pairs are created by matching elements coming from the first stream with elements coming from the second one. In particular, a key value is assigned to each element of both streams,

and each element of the first stream is matched with the elements of the second one that have the same key.

There are three join variants, that determine which tuples will be present in the output stream:

**Inner Join** elements of any of the two streams that have no matching values in the other stream are not present in the output. This is the default variant.

**Left Join** elements of the left stream are always present in the output, even if they are not matched to any element in the right stream. For elements of the right stream, the behavior is the same as the inner join.

**Outer Join** elements of both streams are always present in the output, even if they do not have matching elements.

Note that the *right join* variant does not exist, since it is equivalent to a left join where the two input streams are exchanged.

The type of the data tuples in the output stream changes based on the chosen variant. If the left stream has tuples of type `L` and the right stream of type `R`, then the output stream will have tuples of type `(L, R)` in the case of an inner join, `(L, Option<R>)` in the case of a left join, and `(Option<L>, Option<R>)` in the last case. `Option` is used to handle the case in which an element does not have matching tuples in the other stream: in that case, one of the two sides of the tuple will be `None`.

When joining two streams, two different strategies, called *ship strategies*, can be used to distribute the elements between the different instances:

**Hash Repartition** elements with the same key are sent to the same instance; this is the same behavior of the `group_by` operator.

**Broadcast-Forward** also called *Broadcast-Right*, elements of the left side are kept local to each instance; elements of the right side are broadcast to each instance. This is recommended when the left stream is very big, while the right stream is composed by very few tuples.

Finally, two different *local strategies* can be used to generate the output tuples:

**Hash Join** the tuples are built using hash tables. In particular, two hash tables are kept, one for each of the input streams, in which elements are indexed by key. Whenever a new incoming element needs to be processed, it is matched with the already received tuples of the other streams. This can be done efficiently by simply accessing the hash table using the key of the element that is currently being processed. After generating all the tuples, the new element is inserted in the hash table corresponding to the input stream it came from. As soon as one of the two streams reaches the end, the hash table corresponding to the *other* input is dropped, since it is not useful anymore.

**Sort and Merge** the two input streams are stored in memory and then sorted by key. The output tuples are generated by using an approach similar to the *two pointers technique*.

The *Hash Join* strategy is chosen as the default one, since it has several advantages. First of all, this strategy works well also when dealing with streaming workflows.

Indeed, unlike the *Sort and Merge* strategy, it does not need to consume all the input streams to start producing output elements. Furthermore, it can also use much less memory, since it does not need to store all the elements of both streams in memory in some cases.

Noir provides all the possible combinations of join variant, ship strategy and join strategy, except *outer* joins do not support the *Broadcast-Forward* ship strategy.

By default, the `join` operator is an *inner* join with *Hash Repartition* ship strategy and *Hash Join* local strategy. There are also the `left_join` and `outer_join` shortcuts, that use the same strategies as `join`. Finally, the `join_with` operator can be used to choose between all the possible variants and strategies. Note that all these operators require three parameters: the right stream and two closures to extract the keys from the elements of the two input streams. Listing 36 shows these operators in action.

```
// inner join / hash repartition / hash join strategy
left_stream.join(right_stream, ..., ...);

// left join / hash repartition / hash join strategy
left_stream.left_join(right_stream, ..., ...);

// outer join / hash repartition / hash join strategy
left_stream.outer_join(right_stream, ..., ...);

// customizable join
left_stream.join_with(right_stream, ..., ...)
    .ship_hash()    // or .ship_broadcast_right()
    .local_hash()   // or .local_sort_merge()
    .inner();       // or .left() or .outer()
```

Listing 36: Examples of usage of the `join` operator and its variants.

### Interval Join

When the items of the streams are timestamped, then also the `interval_join` operator can be used. This operators joins two streams together, pairing the elements based on their timestamps. In particular, this operator accepts two time durations called `lower_bound` and `upper_bound` (see Listing 37). An item of the left stream with timestamp `t` will be paired with all the tuples on the right side that have timestamp `q` such that $t - \texttt{lower\_bound} \leq q \leq t + \texttt{upper\_bound}$ (see Figure 5.7).

This operator only support the *inner join* variant and it can be applied to both keyed and non-keyed streams. When dealing with non-keyed streams, the operator will not be instantiated multiple times.

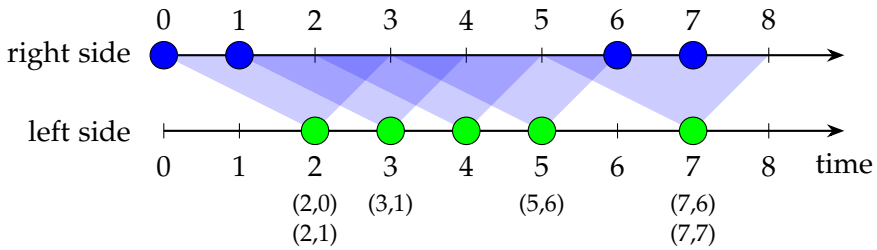



FIGURE 5.7: An example of the `interval_join` operator in action. The *lower bound* is set to 2, while the *upper bound* is set to 1.

```
left_stream.interval_join(
    right_stream,
    Duration::from_secs(5), // lower_bound
    Duration::from_secs(3), // upper_bound
);
```

LISTING 37: Examples of usage of the `interval_join` operator. Each tuple of the left side will be paired with items from the right side whose timestamps are at most five seconds behind or three seconds ahead of the left stream's tuple timestamp.

### 5.6.8 Iterations

Noir supports iterative streams: streams that process data repeatedly. The iteration body, the part in which the data recirculates, is defined inside a closure within a new scope (see Section 4.7). The body is executed with maximum parallelism, and it can access a scope-local state variable. The input stream must not have a limited parallelism specified. At the end of each iteration, the operators of the loop body are reset, so that the stream starts from a clean state each time. Inside the loop there can be any operator, including nested iterations.

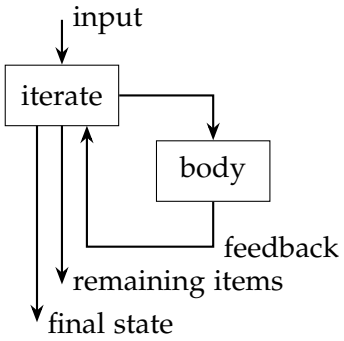
Noir offers two iteration operators: `iterate` and `replay`. Both operators execute the body stream one or more times, taking as input a stream of elements. Additionally, it is also possible to have side-inputs, which are streams that are injected into the loop from outside its scope. In particular, only streams from the *global scope* (i.e. streams not built inside any iteration) can be injected. The items of the side-inputs are replayed at each iteration.

Neither of those operators support streams with timestamped data. This means that iterations are supported only for streams that are bounded. Note that, while Flink supports iterations on unbounded streams, the watermark contract is broken and some operators will not work as expected. For this reason, Noir will reject any iteration on an unbounded stream.

#### Iterate

Adding an `iterate` operator to a stream creates a loop whose first iteration's items are the items of the stream. When the input stream ends, the output of the loop is fed back, and it is the input of the next iteration.

The `iterate` operator takes many parameters:




- `num_iterations` The maximum number of iterations to perform.
- `initial_state` The initial value of the scope-local state variable.
- `body` A closure that builds the body of the loop. The closure takes as parameters the input stream and a handle to the state variable, and should return the feedback stream.
- `local_fold` A closure that is used to aggregate the items that are fed back to produce a `DeltaUpdate`.
- `global_fold` A closure that updates the state variable using the delta updates.
- `loop_condition` A closure that, given the state variable, returns `false` if the loop should be stopped.

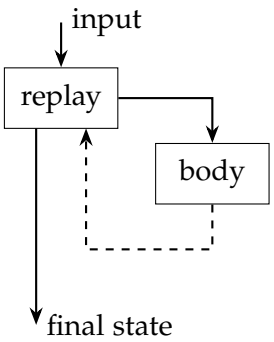
The `iterate` operator produces two different streams:

- The first stream yields a single item: the final state of the iteration;
- The second stream yields all the items of the last iteration (the feedback stream).

The type of the state and of the delta update can be chosen by the user. This operator can be used to iteratively compute values with the help of a state. For example, Listing 38 shows how to find the number which has the longest Collatz sequence.

#### Replay

With `replay`, the input stream is replayed and used as the input for the loop body at each iteration. The items coming out of the loop are only used to update the state and they are not fed back.




The `replay` operator takes the same parameters of `iterate`, but it does not return the feedback stream. This operator only returns a stream that produces a single element: the final state of the iteration.

Just like `iterate`, this operator can be used to compute some values with the help of a scope-local state, but the computation always uses the same input data at each iteration.

```rust
let source = IteratorSource::new(1..limit);
let (state, remaining) = env
    .stream(source)
    .shuffle()
    .map(|n| (0, n, n)) // (# of steps, current val, original val)
    .iterate(
        1000,   // number of iterations
        (0, 0), // initial state: (max steps, starting value)
        // loop body
        |s, _state| {
            s
                .filter(|(_c, cur, _n)| *cur != 1) // remove 1s
                .map(|(c, cur, n)| {
                    // Collatz
                    if cur % 2 == 0 {
                        (c + 1, cur / 2, n)
                    } else {
                        (c + 1, cur * 3 + 1, n)
                    }
                })
        },
        // build a DeltaUpdate: get the maximum number of steps and
        // relative initial value
        |delta: &mut (i32, i32), x| *delta = (x.0, x.2).max(*delta),
        // update the global state with the highest number of steps
        |state, delta| *state = delta.max(*state),
        // loop condition: do all the 1000 iterations
        |_state| true,
    );
```

Listing 38: Example of usage of the `iterate` operator. It considers all the integers between 1 and `limit` and finds the one that has the longest Collatz sequence, keeping it in the iteration state.

```rust
let state = stream
    .replay(
        num_iterations,
        initial_state,
        |s, state| {
            ...
        },
        |delta: &mut DeltaUpdate, x| *delta = ...,
        |state, delta| *state = ...,
        |_state| ...,
    );
```

Listing 39: Example of `replay` operator.

### 5.6.9 Sinks

Sinks are special operators used to consume the stream and to collect the results of the computation. Since they do not generate any new output, they implement the `Operator<()>` trait and cannot have any downstream operators.

#### CollectVec

The `collect_vec` sink is used to collect the elements of the stream into a Rust's `Vec`. In particular, `collect_vec` returns a `StreamOutput<Vec<Out>>` value, which has a `get` method that can be used to obtain the vector of elements. That method should be called only after calling the `execute` method on the environment. Note that this vector is populated with data only on one "master" host: all the other ones will not receive any data (see Listing 40). Beware that this operator will store all the elements of the incoming stream in memory, which can potentially end up causing out-of-memory errors.

```
// collect all the elements generated by the source
let res = env.stream(source).collect_vec();
// execute the computation
env.execute();
// get the values
if let Some(values) = res.get() {
    // only the "master" host will get here
    ...
}
```

Listing 40: Example of usage of the `collect_vec` sink.

#### ForEach

Instead of storing all the items of the stream in memory, it is sometimes useful to execute a given action once for each item. This can be achieved by using the `for_each` sink. It accepts a closure that is applied to every item received by the operator. This provides great flexibility; for example, it can be used to write the items to disk or to an external system (e.g a database). The `for_each` operator is usually instantiated multiple times on multiple hosts, so that items can be consumed in parallel. As an example, Listing 41 prints all the items of the stream to the standard output.

```
// print all the items to the standard output
stream.for_each(|n| println!("Item: {}", n));
// execute the computation
env.execute();
```

Listing 41: Example of usage of the `for_each` sink.

### 5.6.10 Multiple streams

With respect to RStream, one of the main advantages of Noir is the support for multiple streams executing concurrently. In particular, multiple streams can be merged into a single one or, conversely, a single stream can be split into multiple ones.

#### Split

The `split` operator is used to generate multiple streams from a single one. It accepts as parameter the number of duplicated streams to be generated. This operator returns a vector containing as many streams as requested. These streams will all contain the same elements of the input stream. The `split` operator can be useful in many cases, for example to make self-joins, that is a join between a stream and itself (see Listing 42).

```
1 // duplicate the stream into two which have the same elements
2 let streams = stream.split(2);
3 let s1 = streams.pop().unwrap();
4 let s2 = streams.pop().unwrap();
5 // self-join
6 s1.join(s2, ..., ...);
```

Listing 42: Example of usage of the `split` operator.

#### Concat

The `concat` operator is the opposite of `split`. It is used to merge two streams into one that contains the elements of both of them. However, the two input streams must have the same type, the same degree of parallelism and it is not possible to merge timestamped streams with non-timestamped ones. When used on keyed streams, also the type of the keys must match.

```
1 // merge the two streams into one
2 let stream = stream1.concat(stream2);
```

Listing 43: Example of usage of the `concat` operator.

#### Zip

Just like `concat`, the `zip` operator is used to merge two streams into a single one. However, in this case the output stream does not simply contain the items coming from both the input streams. Instead, it will contain pairs of items, where the first element of the pair comes from the first input stream and the second element is from the second stream. Note that if the two input streams do not have the same number of items, then items coming from the bigger stream that are not paired will be discarded. Finally, the `zip` operator is not instantiated multiple times, so it can become the bottleneck of the system.

The same limitations to the two input streams imposed by `concat` are also required by the `zip` operator. This means that it is not possible to mix timestamped and non-timestamped streams and the two input streams must have the same degree of parallelism. It is not necessary, however, that they have the same type of items.

```
// "zip up" the two streams
let stream = stream1.zip(stream2);
```

Listing 44: Example of usage of the `zip` operator.

### 5.6.11 Miscellaneous

The operators in this last section do not fall in any of the previous categories, they can be used for customizing the general behavior of the streams.

#### AddTimestamps

The `add_timestamps` operator is used to add a timestamp, in event-time domain, to each data tuple and to generate watermarks. This operator is usually added right after creating a stream from a specified source and it requires two closures (see Listing 45). The first one is called on each item, and it must return the timestamp associated to it. The second function is also called on each tuple, but it returns an optional watermark.

Note that the second closure must return watermarks that are coherent with the timestamps of the data tuples. In particular, when a watermark with timestamp $t$ is generated, the following items cannot have a timestamp that comes before $t$.

```
// create a stream and add timestamps
env.stream(source).add_timestamps(
    // each item has a timestamp equal to its value, in milliseconds
    |&n| Timestamp::from_millis(n),
    // given an item and its timestamp, generate a watermark
    // only if the item is even
    |&n, &ts| if n % 2 == 0 { Some(ts) } else { None }
);
```

Listing 45: Example of usage of the `add_timestamps` operator.

#### BatchMode

The `batch_mode` operator is used to change the batch mode of the stream (see Section 4.4). The selected batch mode will be applied to the current block and to the following ones.

```
// set the batch mode to "fixed" with given size
env.stream(source).batch_mode(BatchMode::fixed(1024));
// set the batch mode to "adaptive" with given size and timeout
env.stream(source).batch_mode(
    BatchMode::adaptive(256, Duration::from_millis(100)));
```

Listing 46: Example of usage of the `batch_mode` operator.

### MaxParallelism

Sometimes it might be useful to decide how many times a block can be instantiated. This can be done with the help of the `max_parallelism` operator. It accepts one integer, the maximum number of instances that can be spawned; a new block with this limit is then created.

```
stream
    // make sure the following block has exactly one instance
    .max_parallelism(1);
```

Listing 47: Example of usage of the `max_parallelism` operator.

Please note that this represents only an upper bound, since in some cases the resources might not be enough to spawn all the instances requested. Beware that this limit is reset to the default one when an operator splits the block into two. As an example, this operator can be used to make sure that a block is instanced only once (see Listing 47).

### Unkey and DropKey

The `unkey` and `drop_key` operators are used to remove the logical partitioning from a keyed stream. The tuples are not sent over the network, so they stay in the host they were in. These operators can be applied to a `KeyedStream` and they return a `Stream`. However, while `unkey` returns a stream composed of key-value pairs, `drop_key` returns a stream containing only the values (see Listing 48).

```
// create a partitioned stream of key-value pairs
stream1.group_by(...)
    // remove the partitioning while keeping the key-value pairs
    .unkey();

// create a partitioned stream of key-value pairs
stream2.group_by(...)
    // remove the partitioning and the keys, keeping only the values
    .drop_key(); // equivalent to .unkey().map(|(_k, v)| v)
```

Listing 48: Example of usage of the `unkey` and `drop_key` operators.

# 6 Evaluation

This chapter shows many experimental results to evaluate the performance and behavior of Noir under a number of different conditions. We analyze both streaming and batch processing workloads in synthetic and real-world scenarios, and measure the latency and throughput of the system under varying types of load. We also investigate the expressiveness and ease-of-use of each system by looking at the lines of code needed to implement each benchmark. All the results are presented with a comparison with existing frameworks, showing how Noir performs with respect to them.

## 6.1 Environment

All the benchmarks in this chapter are executed using four virtual machines provided by AWS EC2 [15]. Table 6.1 reports their main characteristics.

| | |
|---|---|
| Machine type | `c5.2xlarge` |
| Zone | `us-east-2b` |
| Operating system | Ubuntu 20.04.3 LTS |
| CPU | Intel(R) Xeon(R) Platinum 8124M CPU |
| CPU Frequency | 3.00 GHz |
| CPU Cores | 4 |
| CPU Threads | 8 |
| Cache L1i | 128 KiB |
| Cache L1d | 128 KiB |
| Cache L2 | 4 MiB |
| Cache L3 | 24.75 MiB (shared between 36 hardware threads) |
| RAM | 16 GiB |
| Network | 5 Gbps |
| Ping | 0.12 ms |

Table 6.1: Characteristics of the virtual machines used in the evaluation.

The datasets we consider are small enough to fit in the *Page Cache* memory area managed by the kernel. For this reason, the first warm-up run of each benchmark is not timed, so that the dataset can be cached to RAM. Therefore, subsequent executions of the same benchmark do not depend on the speed of the disks, given that the Linux kernel serves the file content directly from RAM and avoids accessing the disk.

Table 6.2 contains the versions of the tools used in the benchmarks. The MPI programs are executed with the following extra flags:

`-oversubscribe` Allows to run the benchmark with all the available cores, even the *hyperthreading* ones;

`-mca btl_base_warn_component_unused 0` Ignores the warning messages about the unused high-bandwidth network card installed in the machine (it is not available in our VM);

`-map-by hwthread` Map each MPI process to a single hardware thread;

`-display-map` Displays the mapping of the MPI processes to the hardware threads;

`-mca mpi_yield_when_idle 1` Disable the busy-waiting loop in the MPI library.

| | |
|---|---|
| RStream | commit `aacc085` (30/08/2021) |
| Rust | 1.54.0 |
| Flink | 1.11.2 |
| Java | openjdk 14.0.2 |
| MPI | Open MPI 4.0.3 |
| OpenMP | 4.5 |
| GCC | 9.3.0 |

Table 6.2: Versions of the used tools.

## 6.2 Lines of Code

To get a rough idea of the effort required to use each framework, we measured the amount of code needed to implement each benchmark. The results are reported in Table 6.3. The values refer to the number of lines of code, excluding comments and blank lines, as measured by `tokei` [35].

| Benchmark | RStream | Noir | Flink | MPI | Timely |
|---|---|---|---|---|---|
| Non-associative Wordcount | 38 | 50 | – | – | 95 |
| Associative Wordcount | 39 | 49 | 41 | 280 | – |
| Windowed Wordcount | 39 | 48 | 59 | 218 | – |
| Car Accidents | 155 | 220 | 159 | 790 * | – |
| Rolling Top Words | – | 193 | 226 | – | – |
| *k*-means | 157 | 167 | 200 | 419 * | – |
| Enum Triangles | – | 57 | 212 | 309 * | – |
| Connected Components | – | 94 | 161 | 152 * | – |
| PageRank | – | 83 | 203 | 144 * | – |
| Transitive Closure | – | 59 | 141 | – | – |

Table 6.3: Lines of code, excluding comments and empty lines, of all the tested benchmarks.

The cells marked with an asterisk do not include the code needed for the CSV parser. Those lines are not counted since the parser can be simply downloaded and reused. In fact, we used `fast-cpp-csv-parser` [17], which adds around 1000 lines of code.

We can see that, in the simplest benchmarks, Noir, RStream and Flink need pretty much the same amount of code. However, in the more complicated ones like *Enum Triangles*, *Connected Components*, and *PageRank*, Noir requires a lot less code than Flink. In almost all the cases, the MPI implementation is the longest, even without considering the code for the CSV parser.

## 6.3 Benchmarks

This section shows the performance of Noir in heterogenous benchmarks conceived to stress various aspects of the system. Indeed, Noir is compared with the systems introduced in Chapter 3 in both batching and streaming workloads. Furthermore, we analyzed the performance and the latency introduced by the network in the latency benchmark.

Flink, RStream and Noir share the same abstraction and their APIs are very similar, therefore our benchmarks are implemented in a comparable way, using the exact same constructs whenever possible. On the other end, MPI is a very different tool, and the implementation of the benchmarks is not as straightforward as for the other frameworks. Since usually it is not possible to imitate the other systems, we opted for writing the MPI's benchmarks in the most natural way, even if under the hood the programs behave quite differently with respect to the other frameworks.

The results of the benchmarks reported in the following sections are the average of the results of three timed runs.

### 6.3.1 Wordcount

*Wordcount* is the simplest benchmark we analyzed. The dataset is a text file containing a number of words separated by spaces and new lines. The program counts how many times each word appears in the whole document. To do so, the file is split between the instances so that each instance has a different part of the dataset. We analyzed two possible implementations, a *non-associative* and an *associative* one.

The non-associative algorithm works as follows:

- Instance $i$ reads the $i$-th chunk of the dataset;
- The lines are normalized by removing the punctuation (using a regular expression) and lower casing them;
- Each word $w$ in the chunk is sent to the instance $r = \text{hash}(w)$ % num_instances;
- Instance $r$ receives all the copies of the word $w$ and counts them;

The associative variant works as follows:

- Instance $i$ reads the $i$-th chunk of the dataset;
- The lines are normalized by removing the punctuation (using a regular expression) and lower casing them;
- The number of occurrences $n$ of each word $w$ in the chunk is counted (*local reduction*);
- The pair ($w$, $n$) is sent to the instance $r = \text{hash}(w)$ % num_instances;

- Instance *r* receives all the pairs, and aggregates the counts;

Note that, in the associative variant, the words are not immediately sent over the network. Instead, they are aggregated locally and only the partially aggregated results are sent. This results in a significant performance improvement since the network is used a lot less.

We used two different datasets: *Gutenberg* and *Randomwords*.

The Gutenberg dataset is the same used in the RStream thesis [6]. It is a 4 GB file containing a subset of the *Project Gutenberg library of Public Domain books* [21]. Being based on real-world text, there are many distinct words (100 443), but only few of them have many occurrences.

The Randomwords dataset contains randomly generated words (1 GiB, 100 443 distinct words with uniformly distributed frequency).

The main difference between the two datasets is the words distribution: in Randomwords the distribution of the load is uniform, while in Gutenberg the presence of a few *very frequent* words generates an uneven workload distribution between the instances. In the latter case, the distribution may lead to some performance degradation due to the uneven back-pressure. This can be easily seen from Figure 6.1: the instance 19 (in the third host) has more than double the words of most of the other instances.

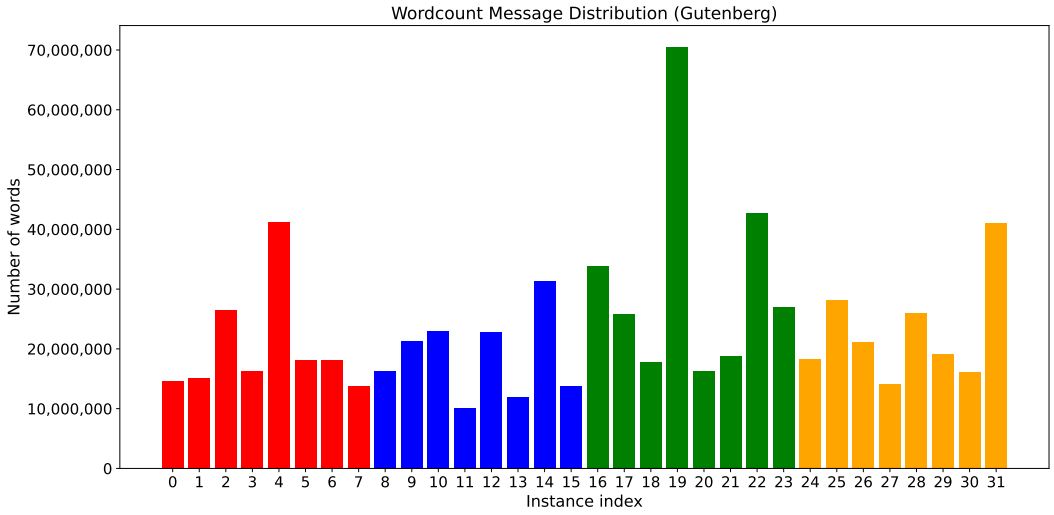



Figure 6.1: Word distribution of Gutenberg between the instances with 4 hosts.

### Non-associative Wordcount

From Table 6.4 and Figure 6.2 we can see that, with the Gutenberg dataset, RStream is slightly faster than Noir with one host, and it scales better with more hosts. This performance gap is not present with the Randomwords dataset (Table 6.5 and Figure 6.3). We suspect this is due to the fact that, with uneven back-pressure, the network stack of Noir loses performance due to multiplexing (see Section 4.6). Thus, RStream might be faster because it uses many more network channels, allowing the faster instances to advance without waiting for the slower ones.

| Hosts | Cores | RStream | Noir | Timely |
|---|---|---|---|---|
| 1 | 8 | 90.10 s (± 0.11 s) | 88.72 s (± 0.21 s) | 77.61 s (± 0.62 s) |
| 2 | 16 | 51.17 s (± 0.09 s) | 61.96 s (± 1.12 s) | 49.92 s (± 0.21 s) |
| 3 | 24 | 37.46 s (± 0.16 s) | 44.35 s (± 0.51 s) | 36.56 s (± 0.34 s) |
| 4 | 32 | 30.04 s (± 0.30 s) | 39.73 s (± 0.11 s) | 31.29 s (± 0.07 s) |

TABLE 6.4: Results of the non-associative *Wordcount* benchmark with the Gutenberg dataset.

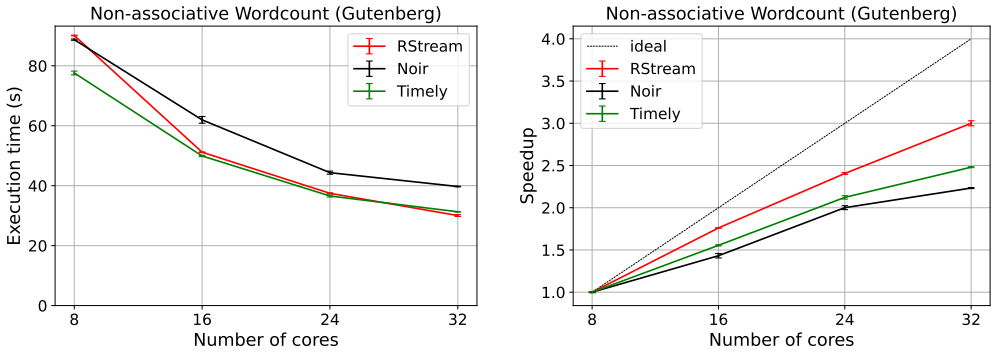



FIGURE 6.2: Execution time and speedup of the non-associative *Wordcount* benchmark with the Gutenberg dataset.

| Hosts | Cores | RStream | Noir | Timely |
|---|---|---|---|---|
| 1 | 8 | 15.39 s (± 0.11 s) | 14.34 s (± 0.07 s) | 12.61 s (± 0.00 s) |
| 2 | 16 | 8.80 s (± 0.04 s) | 9.20 s (± 0.20 s) | 7.53 s (± 0.20 s) |
| 3 | 24 | 6.27 s (± 0.00 s) | 6.42 s (± 0.07 s) | 5.41 s (± 0.08 s) |
| 4 | 32 | 5.17 s (± 0.02 s) | 4.90 s (± 0.09 s) | 4.51 s (± 0.41 s) |

TABLE 6.5: Results of the non-associative *Wordcount* benchmark with the Randomwords dataset.

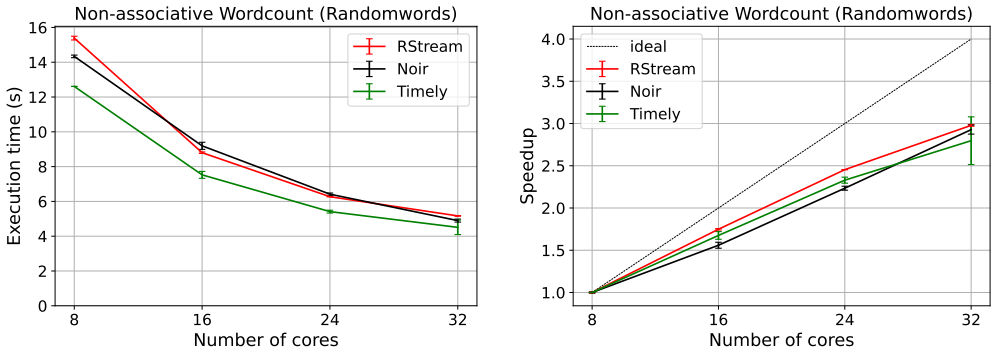



FIGURE 6.3: Execution time and speedup of the non-associative *Wordcount* benchmark with the Randomwords dataset.

Timely Dataflow, is generally the fastest of the three, even if for a small margin. On the other hand, both RStream and Noir offer a richer API than Timely, which allows them to write the associative variant of this benchmark. The associative variant is not only much faster, but also requires less network bandwidth to be fast.

#### Associative Wordcount

We tested many different implementations of the associative variant, since it is the most natural way to implement this benchmark. Not only the execution time is a lot shorter, but also the network usage is orders of magnitude smaller.

| Hosts | Cores | RStream | Noir | Flink | MPI |
|---|---|---|---|---|---|
| 1 | 8 | 49.88 s (± 0.39 s) | 50.88 s (± 0.15 s) | 226.97 s (± 1.18 s) | 10.56 s (± 0.02 s) |
| 2 | 16 | 25.97 s (± 0.02 s) | 25.74 s (± 0.12 s) | 118.15 s (± 0.24 s) | 6.20 s (± 0.05 s) |
| 3 | 24 | 17.44 s (± 0.04 s) | 17.36 s (± 0.05 s) | 80.80 s (± 1.13 s) | 4.43 s (± 0.11 s) |
| 4 | 32 | 13.28 s (± 0.02 s) | 13.14 s (± 0.09 s) | 60.01 s (± 0.06 s) | 3.63 s (± 0.09 s) |

TABLE 6.6: Results of the *Associative Wordcount* benchmark with the Gutenberg dataset.

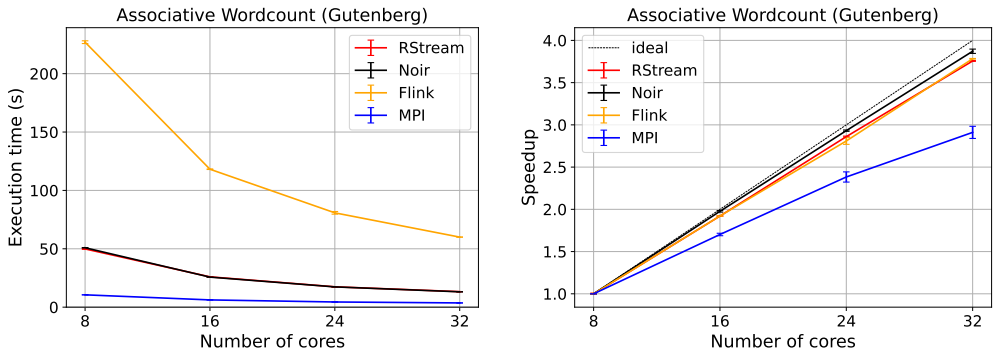



FIGURE 6.4: Execution time and speedup of the *Associative Wordcount* benchmark with the Gutenberg dataset.

This version is much faster than the non-associative one, and pretty much all implementations scale near ideally (see Table 6.6 and Figure 6.4). RStream and Noir have pretty much the exact same performance, MPI is quite a bit faster, while Flink is a lot slower by a factor of nearly 5×.

In Section 6.4 there is a detailed explanation of the network usage of this benchmark in the two versions, in which we can see that Noir is able to use the network a lot less than the other frameworks thanks to its variable length encoding for integer values.

### 6.3.2 Windowed Wordcount

*Windowed Wordcount* is a simple variation of *Wordcount* (see Section 6.3.1), which shows how to use count windows. The dataset format is the same used for the *Wordcount* benchmark: a text file, where each line contains many words separated by spaces. First, the file is read concurrently by multiple instances, and each line is split into a set of words. This stream is then partitioned by word, which means that the items of each partition are all the same word. On each partition, a sliding count window with *size* 10 and *step* 5 is applied. Each window is then reduced into a tuple $(w, n)$, where $w$ is the word corresponding to the partition and $n$ is the number of times $w$ is contained in the window. Note that, for each word, $n$ will always be equal to ten, except for the last window in each partition which might be incomplete.

We ran this benchmark with the same two datasets of Section 6.3.1. As in the non-associative *Wordcount*, also in *Windowed Wordcount* with the Gutenberg dataset there is an uneven load distribution among the hosts.

| Hosts | Cores | RStream | Noir | Flink | MPI |
|---|---|---|---|---|---|
| 1 | 8 | 104.17 s (± 0.36 s) | 105.36 s (± 1.08 s) | 272.55 s (± 3.93 s) | 27.71 s (± 0.17 s) |
| 2 | 16 | 54.99 s (± 0.11 s) | 70.66 s (± 0.09 s) | 178.49 s (± 9.09 s) | 20.38 s (± 0.11 s) |
| 3 | 24 | 41.98 s (± 0.21 s) | 52.46 s (± 0.30 s) | 115.15 s (± 11.32 s) | 14.28 s (± 0.10 s) |
| 4 | 32 | 34.06 s (± 0.20 s) | 44.91 s (± 0.27 s) | 101.76 s (± 8.91 s) | 15.36 s (± 0.01 s) |

TABLE 6.7: Results of the *Windowed Wordcount* benchmark with the Gutenberg dataset.

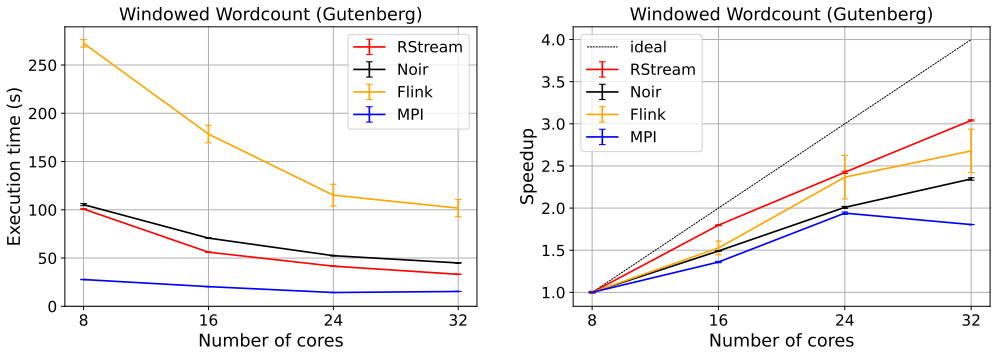



FIGURE 6.5: Execution time and speedup of the *Windowed Wordcount* benchmark with the Gutenberg dataset.

The results with the Gutenberg dataset can be found in Table 6.7 and Figure 6.5.

With the Gutenberg dataset we can see the same behavior as in the non-associative *Wordcount*, RStream is a bit faster with a single host, and the gap grows adding more hosts. As before, Flink is much slower (by at least 2×), and MPI is quite a bit faster.

| Hosts | Cores | RStream | Noir | Flink | MPI |
|---|---|---|---|---|---|
| 1 | 8 | 19.46 s (± 0.10 s) | 21.78 s (± 0.13 s) | 64.17 s (± 0.13 s) | 3.42 s (± 0.02 s) |
| 2 | 16 | 10.70 s (± 0.11 s) | 12.80 s (± 0.14 s) | 32.01 s (± 0.10 s) | 1.91 s (± 0.12 s) |
| 3 | 24 | 7.55 s (± 0.01 s) | 9.35 s (± 0.07 s) | 23.99 s (± 0.13 s) | 1.25 s (± 0.01 s) |
| 4 | 32 | 6.15 s (± 0.01 s) | 7.32 s (± 0.09 s) | 19.98 s (± 0.04 s) | 0.96 s (± 0.00 s) |

TABLE 6.8: Results of the *Windowed Wordcount* benchmark with the Randomwords dataset.

The results with the Randomwords dataset can be found in Table 6.8 and Figure 6.6.

This same behavior of the Gutenberg dataset is observed with the Randomwords dataset, and RStream is still faster than Noir. However, the performance gap is reduced: when considering four hosts, Noir is 31 % slower with the Gutenberg dataset, but only 19 % when considering the Randomwords one. We suspect that the implementation of windows in Noir is not efficient as it could be. However, it must be noted that Noir provides many more aggregators on windows with respect to RStream, and its API is much more flexible.

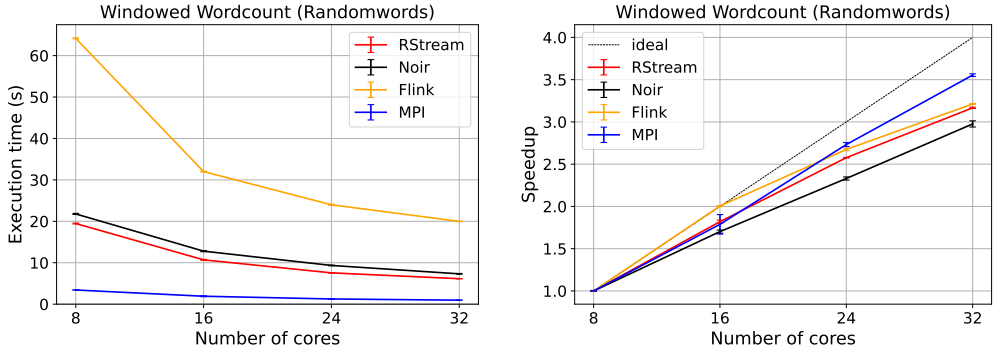



FIGURE 6.6: Execution time and speedup of the *Windowed Wordcount* benchmark with the Randomwords dataset.

### 6.3.3 Car Accidents

The *Car Accidents* benchmark executes three different queries on the dataset *Motor Vehicle Collisions* [20] of the city of New York. This is a CSV file that contains information about the crashes happened in New York, including the date and time, the borough, the number of people killed or injured, the contributing factors of the accident and the type of the vehicles involved. The dataset used to benchmark the various systems is 4.3 GB big, and it corresponds to the original dataset duplicated 25 times, for a total of 23 898 200 events.

The first query calculates the total number of fatal crashes, that is accidents in which at least one person is killed, for each week. This can be done very easily by simply partitioning the stream by week, and then aggregating each individual partition.

Considering the second query, we want to know, for each contributing factor, how many accidents were due to that factor and how many of them were also lethal. Note that each accident can have one or more contributing factors related to it. Similarly to the first query, this can be achieved by partitioning the stream appropriately and aggregating each partition independently.

The third and final query requires finding out how many accidents there were in each borough each week, but also the average percentage of lethal accidents in each borough for each week, aggregating among all years. This can be calculated in two steps. The first one calculates how many accidents there were in each week in each borough, and how many of them were lethal. The second step uses these intermediates results to calculate, for each week in a year, the total number of accidents and the total number of lethal ones. The average percentage is just the ratio of these two values.

This benchmark is very useful in checking the expressiveness and the richness of the feature set of the various systems, given that the queries get progressively more difficult to implement. In particular, one of the advantages of Noir over RStream is the ability to share the data source between multiple streams. This makes it possible to read and parse the CSV file only once, instead of doing that for each query, resulting in significant performance improvements.

In Table 6.9 and Figure 6.7, we can see again that Flink is slower than the other implementations, with RStream and Noir tied to the lead when they read the CSV

| Hosts | Cores | RStream | Noir | Noir (shared) | Flink | MPI |
|---|---|---|---|---|---|---|
| 1 | 8 | 32.97 s (± 0.33 s) | 32.38 s (± 0.10 s) | 23.87 s (± 0.13 s) | 66.02 s (± 0.13 s) | 41.70 s (± 0.11 s) |
| 2 | 16 | 17.19 s (± 0.05 s) | 16.33 s (± 0.04 s) | 12.20 s (± 0.45 s) | 33.97 s (± 0.27 s) | 21.23 s (± 0.06 s) |
| 3 | 24 | 11.68 s (± 0.02 s) | 10.99 s (± 0.09 s) | 8.21 s (± 0.05 s) | 26.33 s (± 0.17 s) | 14.91 s (± 1.20 s) |
| 4 | 32 | 9.02 s (± 0.04 s) | 8.50 s (± 0.20 s) | 6.44 s (± 0.25 s) | 24.11 s (± 2.60 s) | 10.69 s (± 0.02 s) |

TABLE 6.9: Results of the *Car Accidents* benchmark.

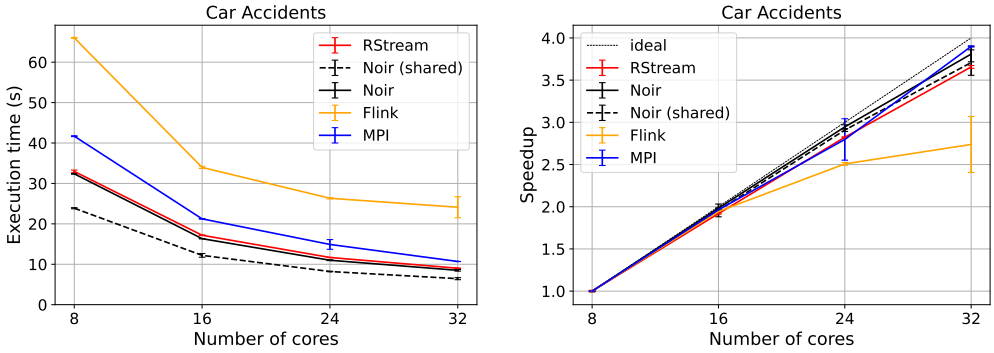



FIGURE 6.7: Execution time and speedup of the *Car Accidents* benchmark.

file once per query. However, when Noir uses a single shared source for all the queries, the speedup with respect to RStream is around 30 % considering all the four hosts.

The MPI implementation reads the dataset once per query, and it is a bit slower than RStream and Noir probably because of a poorly optimized implementation. This shows that, even when developing ad-hoc solutions using low-level primitives, good performance is not guaranteed and a lot of tuning is needed. Even without advanced optimizations, its source code has more than twice the lines of code of Noir (see Section 6.2).

From the point of view of the scalability, all the implementations but Flink's scale pretty much ideally with the number of cores, while Flink struggles with more hosts.

### 6.3.4 Rolling Top Words

While most of the benchmarks in this chapter consist of batch processing workflows, *Rolling Top Words* tests the streaming capabilities of Noir, and it represents well a very typical real-world application.

The input consists in a stream of words, called *hashtags* in this case. The generator produces a random hashtag from a set of 50 (e.g. `#love`, `#instagood`, `#fashion`) with a geometric distribution of parameter $\lambda = 0.1$: the most frequent word has probability $\lambda = 10\%$, the second $(1-\lambda)\lambda = 9\%$, the third $(1-\lambda)^2\lambda = 8.1\%$, and so on.

The output, at every half second, is the top $k = 4$ most frequent words posted in the last second. Therefore, the stream is processed with an event-time sliding window of size 1 s, that slides every 500 ms.

In Noir, the stream is processed by a first window that counts the number of occurrences of each hashtag. It is followed by a `window_all`, a non-parallelizable window, that selects the top $k$ most frequent hashtags. Since this window cannot be parallelized, it may become the bottleneck of the system. Instead, Table API is used to implement this benchmark in Flink.

The data generator produces as many words as possible, spacing them exactly 1 ms apart in event-time. This means that every window contains exactly 1000 hashtags. The test measures how many words per second are processed; in this benchmark, processing time and event time advance independently.

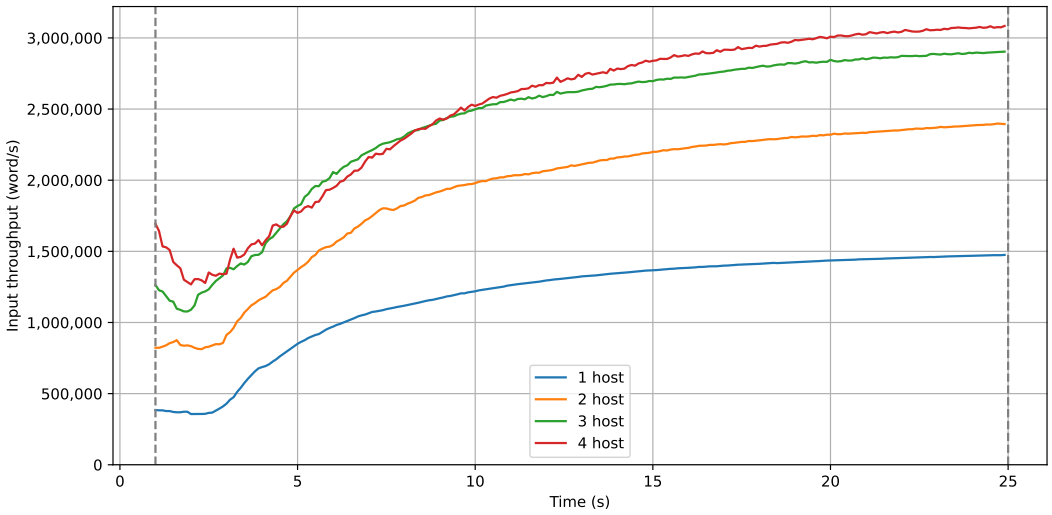



FIGURE 6.8: Input throughput of *Rolling Top Words* in Flink, not showing the first second.

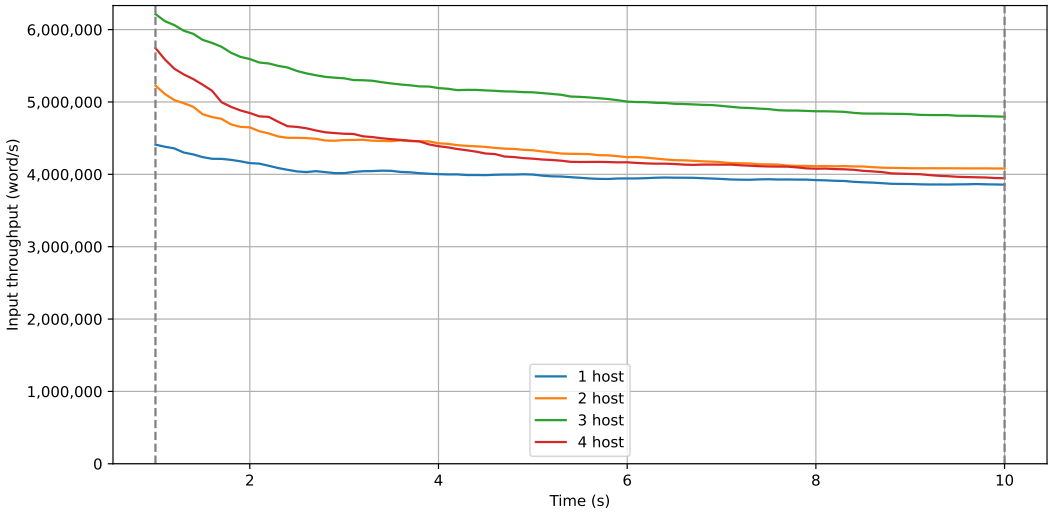



FIGURE 6.9: Input throughput of *Rolling Top Words* in Noir, not showing the first second.

Figure 6.8 and Figure 6.9 show how many words per second Flink and Noir can input into the stream. The first second of the execution is not considered, since the throughput is not yet stable.

We can notice that Flink scales pretty well from one host to three, but the forth one does not increase the throughput by much. This suggests that the bottleneck is able to handle at most three hosts before saturating. On the other hand, Noir does not scale as well; the best performance is achieved with three hosts, but adding a forth reduces the performance. However, Noir has a much higher throughput than Flink: even with a single host it is around 25% faster than Flink with four hosts.

### 6.3.5 $k$-means

The $k$-means clustering algorithm receives as input a cloud of $n$ $d$-dimentional points, and it partitions them into $k$ clusters (see Figure 6.10). To each cluster corresponds a *centroid*, and each point is part of the cluster with the nearest centroid. The algorithm tries to minimize the total distance of the points from their corresponding centroid.

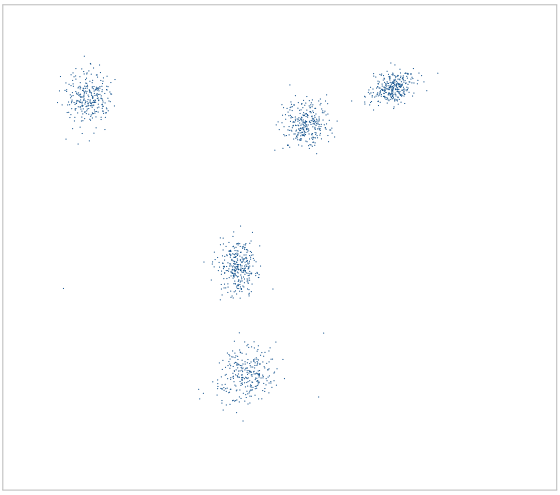


FIGURE 6.10: An example of a $k$-means dataset with $d = 2$. In this case there are $k = 5$ obvious clusters.

This problem is NP-hard, however there exist efficient heuristic algorithms that converge to a local optimum. This benchmark uses the standard (naive) $k$-means algorithm, which is an iterative algorithm.

The initialization step is done by randomly selecting $k$ points from the dataset: they will be the initial $k$ centroids used by the algorithm. Then, at each step, two operations are performed:

- Each point is assigned to the nearest centroid;
- The list of centroids is updated by computing the mean of the points in each cluster.

The algorithm terminates when the centroids do not change anymore, or when the limit of iterations $h$ is reached.

This benchmark tries to analyze the performance of an iterative algorithm. The impact of each parameter is analyzed separately by changing the value of $k$ (the number of centroids), $h$ (the number of iterations), and $n$ (the number of points). Each iteration of the algorithm considers each pair of point-centroid, therefore it has a complexity of $O(k \cdot n)$. Overall, the algorithm has a complexity of $O(k \cdot n \cdot h)$.

| Hosts | Cores | RStream | Noir | Flink | MPI |
|---|---|---|---|---|---|
| 1 | 8 | 6.26 s (± 0.04 s) | 4.68 s (± 0.03 s) | 145.90 s (± 1.02 s) | 3.58 s (± 0.00 s) |
| 2 | 16 | 3.66 s (± 0.02 s) | 2.52 s (± 0.03 s) | 87.25 s (± 1.07 s) | 1.95 s (± 0.01 s) |
| 3 | 24 | 2.89 s (± 0.03 s) | 1.89 s (± 0.12 s) | 67.34 s (± 1.11 s) | 1.40 s (± 0.01 s) |
| 4 | 32 | 2.65 s (± 0.04 s) | 1.79 s (± 0.02 s) | 53.41 s (± 0.98 s) | 1.14 s (± 0.01 s) |

TABLE 6.10: Results of the $k$-means benchmark with $k = 30$, $h = 15$ and $n = 10\,002\,000$.

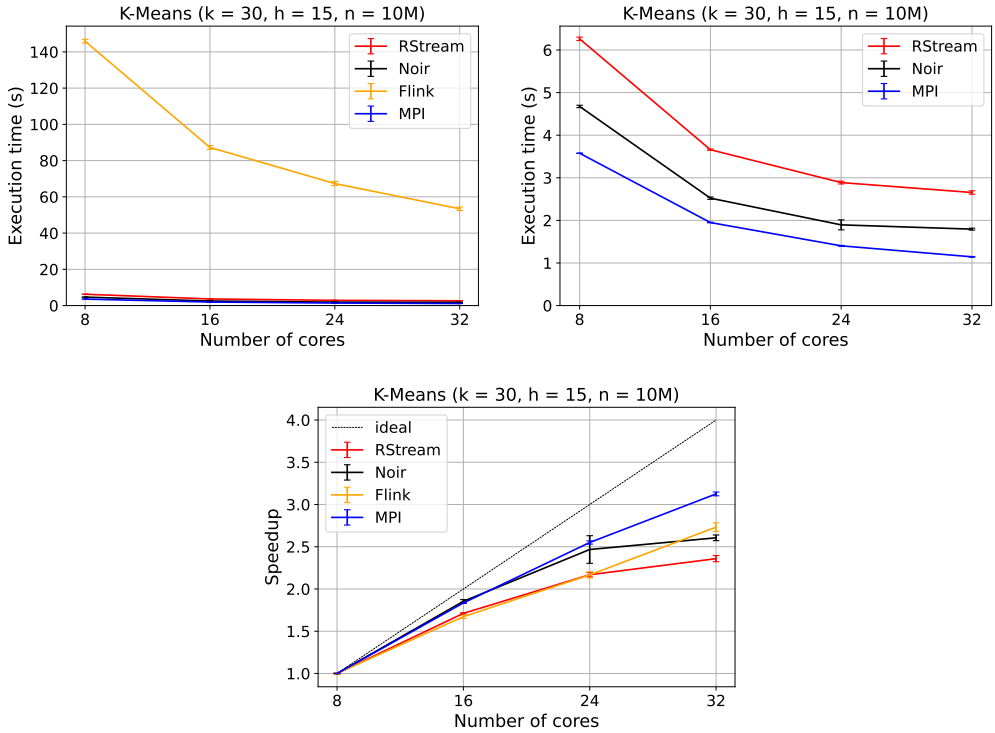



FIGURE 6.11: Execution time and speedup of the $k$-means benchmark with $k = 30$, $h = 15$ and $n = 10\,002\,000$. The second plot is the same as the first, without Flink.

In Table 6.10 and in Figure 6.11 the results are shown for $k = 30$, $h = 15$ and $n = 10\,002\,000$. These results show that the performance of Noir is better than RStream under these conditions, with MPI still being the best of all.

Flink, on the other hand, is much slower than the others, by up to a 30× factor with respecto to Noir. This appears to be caused by the *garbage collector* threads that keep stealing the CPU from the worker threads. Under a closer inspection of the code, this is probably due to a very high number of allocations of "point" objects during the iterations.

The aforementioned results are used as a baseline for the next analysis, which varies one parameter at the time, looking at its impact on the performance of each system.

In Table 6.11 there is a comparison between the baseline and the results of the benchmark with $k = 300$. The number of centroids has been increased by a factor of 10, essentially tenfolding the algorithm's complexity. However, we can notice that

| | Hosts | Cores | RStream | Noir | Flink | MPI |
|---|---|---|---|---|---|---|
| | 1 | 8 | 26.21 s (± 0.08 s) | 23.82 s (± 0.04 s) | 252.23 s (± 2.98 s) | 23.12 s (± 0.01 s) |
| * | 1 | 8 | 6.26 s (± 0.04 s) | 4.68 s (± 0.03 s) | 145.90 s (± 1.02 s) | 3.58 s (± 0.00 s) |
| | 2 | 16 | 13.58 s (± 0.01 s) | 11.99 s (± 0.03 s) | 144.71 s (± 0.07 s) | 11.82 s (± 0.00 s) |
| * | 2 | 16 | 3.66 s (± 0.02 s) | 2.52 s (± 0.03 s) | 87.25 s (± 1.07 s) | 1.95 s (± 0.01 s) |
| | 3 | 24 | 9.54 s (± 0.04 s) | 8.09 s (± 0.07 s) | 104.62 s (± 0.02 s) | 8.09 s (± 0.03 s) |
| * | 3 | 24 | 2.89 s (± 0.03 s) | 1.89 s (± 0.12 s) | 67.34 s (± 1.11 s) | 1.40 s (± 0.01 s) |
| | 4 | 32 | 7.61 s (± 0.04 s) | 6.25 s (± 0.12 s) | 83.82 s (± 1.06 s) | 6.21 s (± 0.01 s) |
| * | 4 | 32 | 2.65 s (± 0.04 s) | 1.79 s (± 0.02 s) | 53.41 s (± 0.98 s) | 1.14 s (± 0.01 s) |

TABLE 6.11: Results of the $k$-means benchmark with $k = 300$ (increased from $k = 30$), $h = 15$ and $n = 10\,002\,000$. The lines marked with an asterisk are the same as the ones in Table 6.10 (the baseline).

the running times increase by factors smaller than 10×. This is due to the fact that not all the execution time is spent making numerical operations, but some of it is lost to synchronization and communication.

The relative order between the 4 frameworks remains the same. However, MPI and Noir are now very close in term of performance, while RStream is a little bit slower. This is expected, given that increasing the number of centroids increases primarily the number of numerical operations, which do not require any synchronization. Instead, the cost for the communication remains quite similar, so its relative weight with respect to the whole computation decreases. In any case, Flink is still the worst performer.

| | Hosts | Cores | RStream | Noir | Flink | MPI |
|---|---|---|---|---|---|---|
| | 1 | 8 | 9.69 s (± 0.01 s) | 8.26 s (± 0.01 s) | 297.77 s (± 0.97 s) | 6.35 s (± 0.02 s) |
| * | 1 | 8 | 6.26 s (± 0.04 s) | 4.68 s (± 0.03 s) | 145.90 s (± 1.02 s) | 3.58 s (± 0.00 s) |
| | 2 | 16 | 5.44 s (± 0.03 s) | 4.35 s (± 0.04 s) | 165.50 s (± 1.04 s) | 3.34 s (± 0.01 s) |
| * | 2 | 16 | 3.66 s (± 0.02 s) | 2.52 s (± 0.03 s) | 87.25 s (± 1.07 s) | 1.95 s (± 0.01 s) |
| | 3 | 24 | 4.29 s (± 0.03 s) | 3.37 s (± 0.07 s) | 123.50 s (± 0.97 s) | 2.36 s (± 0.00 s) |
| * | 3 | 24 | 2.89 s (± 0.03 s) | 1.89 s (± 0.12 s) | 67.34 s (± 1.11 s) | 1.40 s (± 0.01 s) |
| | 4 | 32 | 3.78 s (± 0.05 s) | 2.83 s (± 0.24 s) | 100.13 s (± 0.90 s) | 1.86 s (± 0.00 s) |
| * | 4 | 32 | 2.65 s (± 0.04 s) | 1.79 s (± 0.02 s) | 53.41 s (± 0.98 s) | 1.14 s (± 0.01 s) |

TABLE 6.12: Results of the $k$-means benchmark with $k = 30$, $h = 30$ (increased from $h = 15$) and $n = 10\,002\,000$. The lines marked with an asterisk are the same as the ones in Table 6.10 (the baseline).

Doubling the number of iterations (see Table 6.12) causes Flink's runtime to nearly double, while the execution times of the other frameworks usually increase by a smaller factor. Under these conditions, the implementation in MPI scales better than the others, and Noir gains a bit of margin from RStream. The gap between Flink and the others is even larger than before, with a factor of at least 35× with respect to Noir.

| | Hosts | Cores | RStream | Noir | Flink | MPI |
|---|---|---|---|---|---|---|
| | 1 | 8 | 11.87 s (± 0.09 s) | 9.30 s (± 0.01 s) | 282.21 s (± 3.96 s) | 6.85 s (± 0.00 s) |
| * | 1 | 8 | 6.26 s (± 0.04 s) | 4.68 s (± 0.03 s) | 145.90 s (± 1.02 s) | 3.58 s (± 0.00 s) |
| | 2 | 16 | 6.56 s (± 0.05 s) | 4.76 s (± 0.06 s) | 156.01 s (± 0.96 s) | 3.59 s (± 0.00 s) |
| * | 2 | 16 | 3.66 s (± 0.02 s) | 2.52 s (± 0.03 s) | 87.25 s (± 1.07 s) | 1.95 s (± 0.01 s) |
| | 3 | 24 | 5.06 s (± 0.03 s) | 3.49 s (± 0.10 s) | 114.62 s (± 0.08 s) | 2.50 s (± 0.01 s) |
| * | 3 | 24 | 2.89 s (± 0.03 s) | 1.89 s (± 0.12 s) | 67.34 s (± 1.11 s) | 1.40 s (± 0.01 s) |
| | 4 | 32 | 4.34 s (± 0.04 s) | 2.88 s (± 0.19 s) | 89.44 s (± 1.15 s) | 1.96 s (± 0.00 s) |
| * | 4 | 32 | 2.65 s (± 0.04 s) | 1.79 s (± 0.02 s) | 53.41 s (± 0.98 s) | 1.14 s (± 0.01 s) |

TABLE 6.13: Results of the $k$-means benchmark with $k = 30$, $h = 15$ and $n = 20\,004\,000$ (increased from $n = 10\,002\,000$). The lines marked with an asterisk are the same as the ones in Table 6.10 (the baseline).

The last version of this benchmark (see Table 6.13) uses a dataset that is double in size. Again, the running time should be doubled, and in this case the execution times are about twice as much as the baseline, as expected.

### 6.3.6 Enum Triangles

This is the first benchmark that deals with graph processing. In *Enum Triangles* the input is an undirected graph, and the output is the list of all the *triangles* in it. A triangle is a triplet of vertices that are directly connected together. In Figure 6.12 you can find an example.

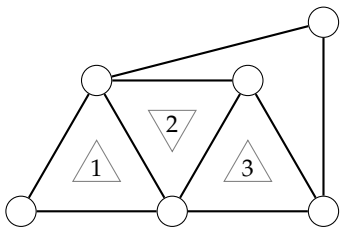



FIGURE 6.12: An example of the *Enum Triangles* benchmark. There are 6 nodes and 7 edges, forming a total of 3 triangles.

This test shows the performance of the `join` operator. The algorithm works as follows:

- For each undirected edge from $a$ to $b$, the algorithm considers only the directed edge $a \to b$, with $a < b$;
- The edges are grouped by their source vertex;
- For each vertex, the list of its neighbors is computed;
- From the list of neighbors, the list of *potential* triangles is generated, that is the list of pair of edges sharing the same source vertex;
- This list is filtered (using a `join`) keeping only the *closed* triangles;
- Finally, the number of triangles is counted.

The dataset we used is a randomly generated Erdős–Rényi graph[1], with 1500 nodes and 900 000 edges. In total there are 288 000 876 triangles in the graph.

| Hosts | Cores | Noir | Flink | MPI |
|---|---|---|---|---|
| 1 | 8 | 26.22 s (± 0.27 s) | 269.30 s (± 0.95 s) | 7.44 s (± 0.05 s) |
| 2 | 16 | 17.27 s (± 0.33 s) | 144.46 s (± 4.12 s) | 4.39 s (± 0.05 s) |
| 3 | 24 | 13.36 s (± 0.16 s) | 96.41 s (± 0.11 s) | 3.03 s (± 0.09 s) |
| 4 | 32 | 10.98 s (± 0.17 s) | 70.38 s (± 0.16 s) | 2.68 s (± 0.08 s) |

TABLE 6.14: Results of the *Enum Triangles* benchmark.

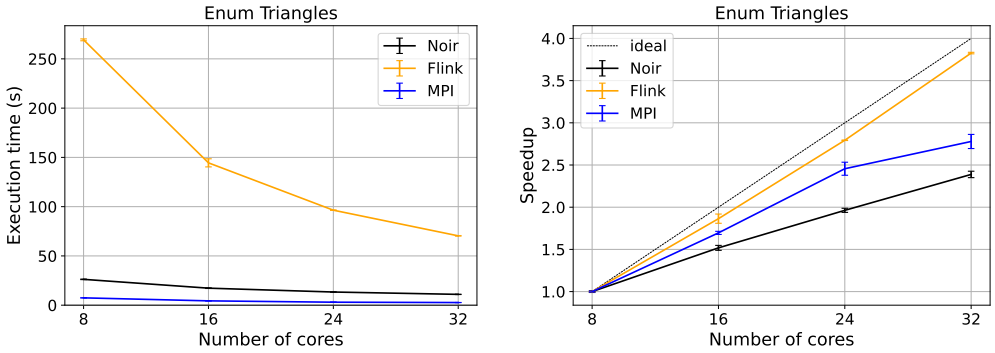



FIGURE 6.13: Execution time and speedup of the *Enum Triangles* benchmark.

From Table 6.14 and Figure 6.13 we can see how the execution time of Noir is much lower than the Flink's one, up to a factor of 10×. Even though Flink is much slower than Noir, its scalability is close to be ideal. The implementation in MPI is a quite a bit faster than the other ones.

### 6.3.7 Connected Components

As the name suggests, *Connected Components* counts the number of connected components in a graph. That is, given an undirected graph, the number of maximal subgraphs whose nodes are all reachable from each other. In Figure 6.14 you can find a very simple example.

This is an iterative and exact algorithm, that also includes the `join` operator inside the loop body. A *component ID* is assigned to each node, initially equal to the *node ID*. This identifier corresponds to the smallest node ID in the component. The algorithm repeats the following steps until the component ID of each node reaches convergence:

- Process all the edges in the graph;
- If there is an edge $a \to b$ and $\text{comp}_a \neq \text{comp}_b$, and assuming without loss of generality that $\text{comp}_a < \text{comp}_b$, then $\text{comp}_b$ is set to $\text{comp}_a$.

The number of iterations of this algorithm is, in the worst case, equal to the diameter of the largest component.

[1] In an Erdős–Rényi network each pair of nodes has the same probability of being connected by an edge.

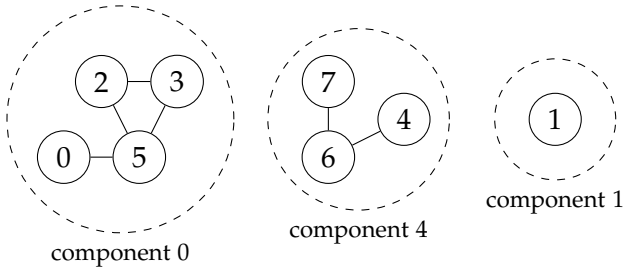



FIGURE 6.14: An example of the *Connected Components* benchmark. There are 8 nodes and 6 edges, forming a total of 3 connected components.

The dataset used is a random graph with 100 components, 200 000 nodes and 5 000 000 edges. The number of iterations required to reach convergence is 6.

| Hosts | Cores | Noir | Flink | MPI |
|---|---|---|---|---|
| 1 | 8 | 4.21 s (± 0.04 s) | 17.36 s (± 1.22 s) | 1.13 s (± 0.03 s) |
| 2 | 16 | 2.52 s (± 0.02 s) | 12.73 s (± 0.14 s) | 1.10 s (± 0.01 s) |
| 3 | 24 | 2.04 s (± 0.07 s) | 10.63 s (± 0.11 s) | 1.06 s (± 0.00 s) |
| 4 | 32 | 1.55 s (± 0.04 s) | 10.33 s (± 0.41 s) | 1.04 s (± 0.00 s) |

TABLE 6.15: Results of the *Connected Components* benchmark.

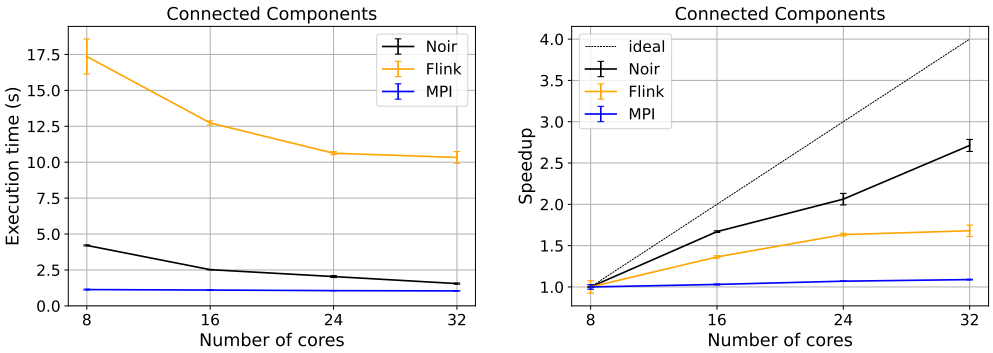



FIGURE 6.15: Execution time and speedup of the *Connected Components* benchmark.

The results can be found in Table 6.15 and Figure 6.15.

This benchmark is quite simple, and it allows some sophisticated primitives to be used by MPI. The implementation in MPI uses `MPI_Allgatherv` to synchronize the array with the component IDs, essentially placing all the communication at the end of each iteration. This results in a very low overhead of communication, and therefore in very short execution times.

On the other hand, Noir is from 4× up to 6× faster than Flink. The implementation in Flink uses *delta iterations*, while the one in Noir keeps the components in the iteration's state.

All the implementations do not scale particularly well. MPI seems not to scale at all, but its execution time is very low from the start, so probably most of it is due to the

communication overhead. In any case, Noir scales better than both Flink and MPI, achieving more than 2.5× performance with four hosts with respect to a single one.

### 6.3.8 PageRank

As the name implies, the *PageRank* benchmark is used to evaluate the performance of the various systems when executing the *PageRank* algorithm. This algorithm can be applied to directed graphs, and it is used to rate the importance of each node in the network. Each node has a weight associated to it, called *rank*. The bigger the rank, the more important the node is. This benchmark exploits many features of Noir, including iterations, iterations' state, side inputs and joins.

In particular, the algorithm works as follows. Initially, to each node is assigned the same rank, and the sum of all the ranks must be equal to one. Then, each node evenly distributes its own rank between all the nodes that it can reach using one of its outbound edges. Finally, the ranks are modified using a *damping factor* (equal to 0.85 in our case). These steps are repeated until convergence, that is until the rank of each node does not change anymore, or when the number of iterations reaches a chosen limit.

The dataset used is a network derived from the Twitter social network, provided by the *Stanford Network Analysis Project* [23]. It is a 26 MB CSV file containing the edges of the graph. This graph is composed of 81 306 nodes and 2 420 766 directed edges. The computation is performed for 100 iterations.

| Hosts | Cores | Noir | Flink | MPI |
|---|---|---|---|---|
| 1 | 8 | 8.60 s (± 0.08 s) | 136.88 s (± 1.95 s) | 3.70 s (± 0.09 s) |
| 2 | 16 | 6.19 s (± 0.16 s) | 130.03 s (± 1.10 s) | 3.19 s (± 0.06 s) |
| 3 | 24 | 5.50 s (± 0.24 s) | 125.44 s (± 3.73 s) | 2.52 s (± 0.03 s) |
| 4 | 32 | 5.27 s (± 0.09 s) | 125.69 s (± 0.95 s) | 2.39 s (± 0.02 s) |

Table 6.16: Results of the PageRank benchmark.

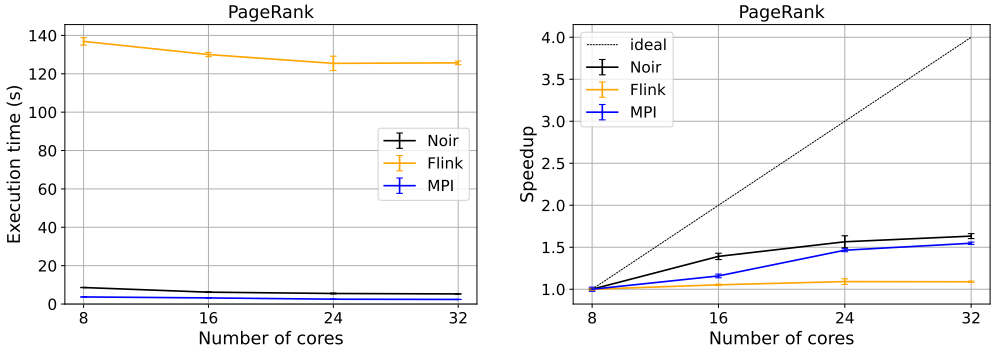



Figure 6.16: Execution time and speedup of the PageRank benchmark.

The results of this benchmark can be found in Table 6.16 and Figure 6.16.

As in Section 6.3.7, this benchmark allows for some sophisticated primitives to be used by MPI. The rank of the nodes is shared via `MPI_Allgatherv` only at the end of each iteration, achieving a very low synchronization overhead.

Just like the previous benchmarks, Noir's performance is much better than Flink's, being at least 15× faster in all the scenarios considered. In any case, the scalability of all the systems is quite far from being ideal, the best being Noir that is able to achieve only about 1.5× the performance of a single host, when using four of them.

### 6.3.9 Transitive Closure

The last batch processing benchmark is *Transitive Closure*. This algorithm computes the transitive closure of a directed graph: every time there are the edges $a \to b$ and $b \to c$, it adds, if missing, the edge $a \to c$, until convergence (see Figure 6.17).

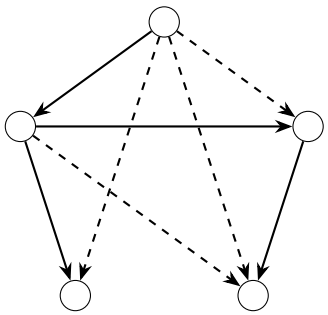


FIGURE 6.17: An example of the *Transitive Closure* benchmark. The original graph has 5 nodes and 4 edges. Its transitive closure adds 4 new edges (the dashed ones).

The peculiarity of this algorithm is that it is iterative, and the size of the data inside the loop grows over time. Indeed, the number of edges at the end of the algorithm can be quadratic with respect to the number of nodes, and the number of elements present in the stream during the computation can be even cubic.

The dataset used for this benchmark consists in a randomly generated network with 2000 nodes and 3000 edges. At the end of the execution, there are 1 448 937 edges present in the transitive closure of the graph.

| Hosts | Cores | Noir | Flink |
|---|---|---|---|
| 1 | 8 | 15.47 s (± 0.21 s) | 68.56 s (± 0.62 s) |
| 2 | 16 | 8.49 s (± 0.10 s) | 47.83 s (± 3.71 s) |
| 3 | 24 | 6.30 s (± 0.06 s) | 39.08 s (± 0.16 s) |
| 4 | 32 | 5.26 s (± 0.13 s) | 36.70 s (± 0.53 s) |

TABLE 6.17: Results of the *Transitive Closure* benchmark.

From Table 6.17 and Figure 6.18, we can see that, again, Noir is much better than Flink, both in terms of execution time and scalability. This is probably due to the memory management of Flink, which might struggle in this benchmark as there are many objects to allocate.

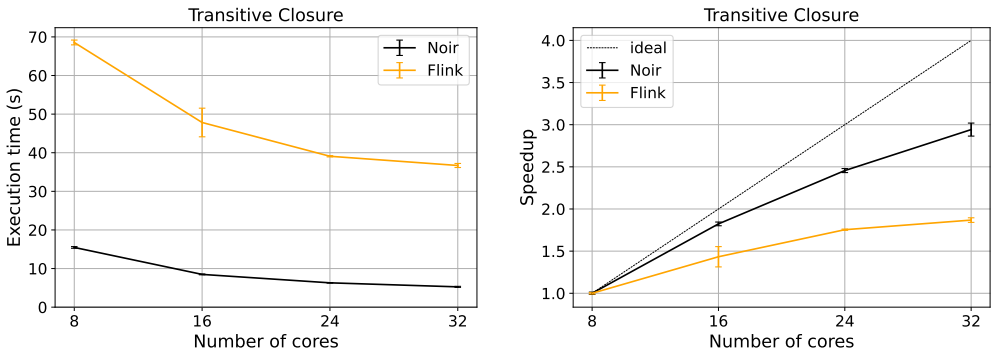



FIGURE 6.18: Execution time and speedup of the Transitive Closure benchmark.

### 6.3.10 Latency

This last benchmark tries to measure the latency of the system. The setup is pretty simple:

- A number of instances insert items at a given rate into the system;
- All the items are sent to the instance $X_0$, which tags each of them with the local system time;
- The items are sent to the instance $X_1$, then to the instance $X_2$, then $X_3$, then $X_4$;
- Finally, the items are sent back to $X_0$, and the latency is computed using the difference between the local system time and the time of the item.

In total there are 5 network shuffles included in the measurement; during the computation the messages simply pass through the operators, without changing in number. This computation is accurate because the system time is measured on the same node, so there is no need to synchronize the clocks between the hosts.

We analyzed the latency under three scenarios:

- Light load (1000 items per second), with batch mode fixed to 1000;
- Light load (1000 items per second), with batch mode adaptive (1000 items, 50 ms of maximum delay);
- Heavy load (as many items as possible per second), with batch mode fixed to 1000.

In Figure 6.19 we can see the measurements of the latency under light load, with batch mode fixed to 1000. Since the throughput is 1000 items per second, a batch is filled in about one second. Therefore, when items enter the first batch, the first one of them has to wait for 1 s, while the last one does not wait any time at all, triggering the flush of the batch. This means that, after 1000 tuples, the batch is full and is sent over the network. Note that items only have to wait at the beginning of the pipeline at $X_0$, given that when they reach the second host at $X_1$ the batch is already full, so it is immediately sent downstream.

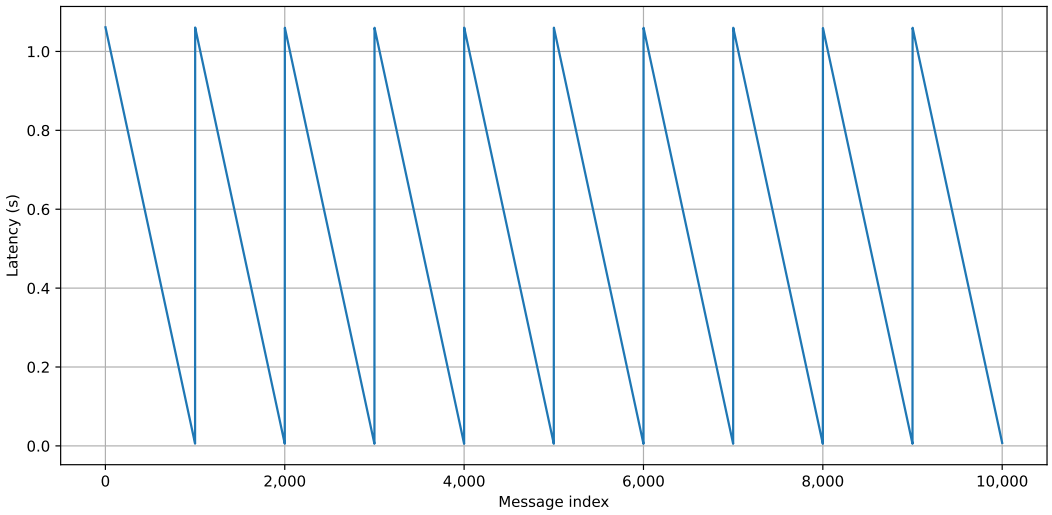



FIGURE 6.19: Latency with fixed size batching under light load.

We can see that, under light load, the latency can be very high. If the throughput is constant, and no message is added or removed from the stream, we can estimate the maximum latency as the batch size divided by the throughput.

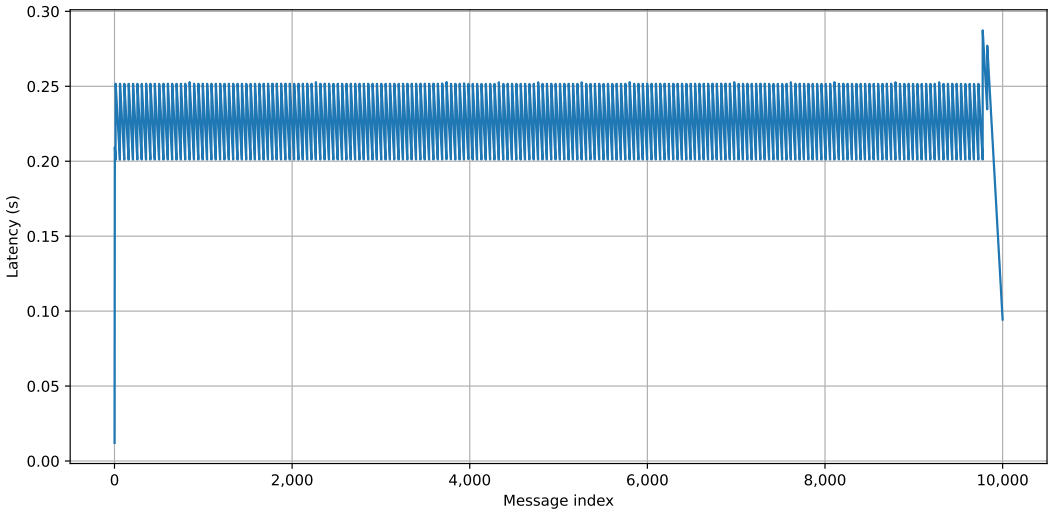



FIGURE 6.20: Latency with adaptive batching under light load.

Figure 6.20 shows the latency obtained by using the adaptive batching mode, instead of the fixed one. We can see that the average latency is much lower, and it is very consistent throughout the experiment. Each network shuffle adds around 50 ms of delay at most, as configured, so the overall latency is around 250 ms given that there are five steps. Just like with fixed batching, the first tuple entering the batch has to wait the full batch time (50 ms), while the last one does not; therefore the overall latency oscillates between 200 ms and 250 ms. For these reasons, the adaptive batching mode is the one used by default one when no batch mode is specified.

The last variant of this benchmark is the one under heavy load. All the available instances generate data as fast as possible, until the back-pressure slows them down.

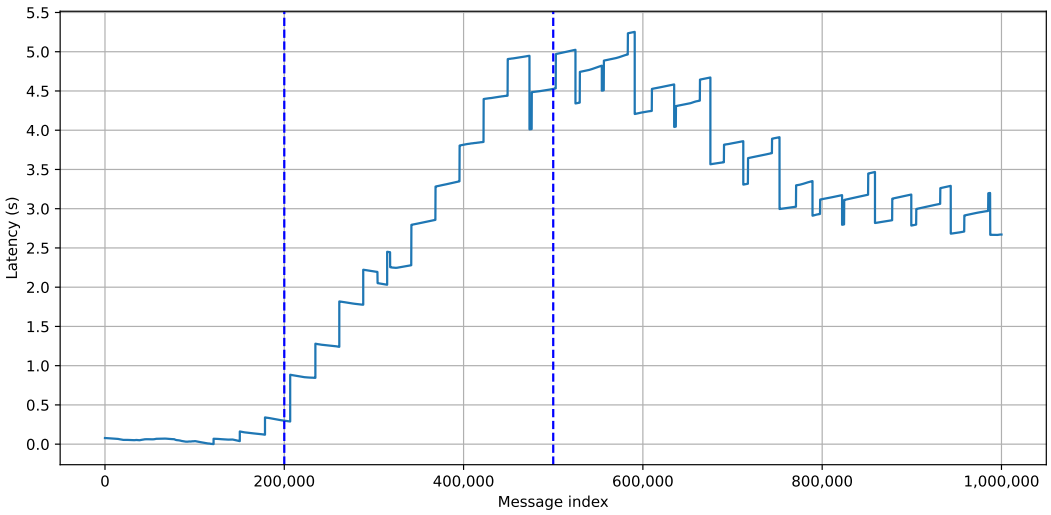



FIGURE 6.21: Latency with adaptive batching under heavy load.

The batch mode has a marginal role in this case, since the batches are filled almost immediately.

The behavior can be divided into three parts:

- At the very start, when the system is empty, the latency is very low since the batches are filled quickly, and they can reach the sinks of the stream without slowdowns.
- Then the system starts to fill up. The latency is still low, but it starts growing over time since the operators are not able to keep up with the throughput.
- Finally, the back-pressure slows down the whole system, including the sources, stabilizing the latency at a near-constant value.

We can see those phases in Figure 6.21: the first one lasts for about the first 200 000 items, the second one for about the next 300 000, and the third one until the end of the test.

## 6.4 Network Usage

By using the profiler (see Appendix B), we are able to study the network usage of the two variants of the *Wordcount* benchmark. The Job Graph of both variants consists in three blocks; this means that we can measure the network usage between the first two blocks, and between the last two blocks. The results are shown in Table 6.18, and they refer to the Gutenberg dataset. We can see that the associative variant uses a lot less the network than the non-associative one, and this is due to the local reduction done in the first block.

By using the `iptables` kernel module we are also able to precisely measure the network usage of all the frameworks, between two hosts. In Table 6.19 we can see the results of both the associative and non-associative variants. While Noir and RStream work pretty much in the exact same way, Noir is able to use send much

| Connection | Non-associative | Associative |
|---|---|---|
| 1–2 | 3.3 GiB | 25.1 MiB |
| 2–3 | 866 KiB | 849 KiB |

TABLE 6.18: Network usage of Noir when executing the two variants of the *Wordcount* benchmark with the Gutenberg dataset, using four hosts.

fewer bytes of data over the network, thanks to the variable-size integer encoding provided by `bincode` [26].

| | Noir | RStream | Flink | MPI |
|---|---|---|---|---|
| Associative | 2.1 MiB | 7.7 MiB | 6.1 MiB | 33.1 MiB |
| Non-associative | 236.3 MiB | 1032.1 MiB | – | – |

TABLE 6.19: Network usage from the first to the second host while running *Wordcount* with the Gutenberg dataset, using four hosts.

# 7 Conclusions and Future Work

In this thesis we presented Noir, a stream-processing framework that aims to provide easy to use facilities to construct custom computations, with minimal compromises to scalability and performance.

The benchmarks show that Noir consistently outperforms Apache Flink, while providing a very similar API and feature set. At the same time, Noir is on par with RStream, as they have similar behavior in nearly all benchmarks; however, Noir provides many improvements and new operators, including joins and more flexible iterations. Finally, while low-level implementations in MPI are usually faster than all the other competitors, Noir is able to rival and sometimes beat them in some workloads. All considered, these results show that it is actually possible to have a much better trade-off between performance and ease-of-use than what is achievable today with currently existing distributed processing frameworks.

## 7.1 Future Work

Noir provides very similar features to Apache Flink, with a notable exception: fault tolerance. This is needed when dealing with long-lasting computations, such as streaming pipelines that can be kept running for very long periods of time. One possible approach is to implement a mechanism similar to *coordinated checkpointing* or *distributed snapshot* to save the global state of the system to persistent storage. At the same time, the data stream of each source needs to be persisted as well. In case of failure, the system can be recovered by using one of these snapshots to rewind the state of each operator and then replaying the streams of each source. In particular, Flink uses a variant of the Chandy-Lamport algorithm called *Asynchronous Barrier Snapshotting* [3]. Note that special care must be taken when dealing with iterations, as the resulting Job and Execution Graphs are cyclic.

Another possible improvement is to provide high-level primitives that can be used to define new computations. For example, Flink provides the Table API, a high-level interface in which datasets are modeled as tables that can be modified using relational operators such as selection, filtering and join. Flink also supports defining queries using SQL. These APIs are very useful, as they make the framework more accessible to developers familiar with SQL and RDBMS. Another option would be to provide ready-to-use implementations of commonly used algorithms, for example the ones used for graph analysis, event processing, machine learning and deep learning.

On the other hand, the support for custom operators is also missing. Even though the interface provided by Noir is very complete and expressive, it might be interesting to be able to define new operators that manually handle all the elements flowing into a stream, including watermarks and control elements. This feature makes it possible to implement operators with a very complex logic or that exploit peculiarities of the stream to make optimized computations.

Considering the performance of the system, the benchmarks in Chapter 6 show that Noir is generally very fast. However, there are some scenarios in which Noir is not as performant or scales as well as expected, in particular in the Non-associative Wordcount and Windowed Wordcount benchmarks. In the former, the slowness seems to be related to back pressure, which is due to an uneven partitioning of the load. Instead, windows seem to cause the latter benchmark to not perform as well as possible. Therefore, some performance benefits can probably be gained by optimizing or reworking the implementations of the windows and of the network layer.

Finally, it might be interesting to investigate if it is convenient and feasible to implement hardware acceleration, exploiting the processing power of GPUs or ad-hoc coprocessors. This can be very beneficial for some of the proposed extensions to Noir, such as the training of neural networks used in deep learning applications.

# Sample operator implementation

The `map` operator is defined by implementing the `Operator` trait on the `Map` struct. This struct holds the operators chain (`prev` field) and the user-provided closure to perform the mapping between input and output elements (`f` field).

```
#[derive(Clone, Derivative)]
#[derivative(Debug)]
pub struct Map<Out: Data, NewOut: Data, F, PreviousOperators>
where
    F: Fn(Out) -> NewOut + Send + Clone,
    PreviousOperators: Operator<Out>,
{
    prev: PreviousOperators,
    #[derivative(Debug = "ignore")]
    f: F,
    _out: PhantomData<Out>,
    _new_out: PhantomData<NewOut>,
}
```

First of all, we define a `new` method so that the `Map` struct can be constructed.

```
impl<Out: Data, NewOut: Data, F, PreviousOperators> Map<Out, NewOut,
↪  F, PreviousOperators>
where
    F: Fn(Out) -> NewOut + Send + Clone,
    PreviousOperators: Operator<Out>,
{
    fn new(prev: PreviousOperators, f: F) -> Self {
        Self {
            prev,
            f,
            _out: Default::default(),
            _new_out: Default::default(),
        }
    }
}
```

Then, we implement the `Operator` trait to describe how the `map` operator behaves.

```rust
impl<Out: Data, NewOut: Data, F, PreviousOperators> Operator<NewOut>
    for Map<Out, NewOut, F, PreviousOperators>
where
    F: Fn(Out) -> NewOut + Send + Clone,
    PreviousOperators: Operator<Out>,
{
    fn setup(&mut self, metadata: ExecutionMetadata) {
        self.prev.setup(metadata);
    }

    fn next(&mut self) -> StreamElement<NewOut> {
        self.prev.next().map(&self.f)
    }

    fn to_string(&self) -> String {
        format!(
            "{} -> Map<{} -> {}>",
            self.prev.to_string(),
            std::any::type_name::<Out>(),
            std::any::type_name::<NewOut>()
        )
    }

    fn structure(&self) -> BlockStructure {
        self.prev
            .structure()
            .add_operator(OperatorStructure::new::<NewOut, _>("Map"))
    }
}
```

Finally, we implement the `map` method on both `Stream` and `KeyedStream`, which adds the operator to the operators chain of the given stream.

```rust
impl<Out: Data, OperatorChain> Stream<Out, OperatorChain>
where
    OperatorChain: Operator<Out> + 'static,
{
    pub fn map<NewOut: Data, F>(self, f: F) -> Stream<NewOut, impl
    ↪ Operator<NewOut>>
    where
        F: Fn(Out) -> NewOut + Send + Clone + 'static,
    {
        self.add_operator(|prev| Map::new(prev, f))
    }
}

impl<Key: DataKey, Out: Data, OperatorChain> KeyedStream<Key, Out,
    ↪ OperatorChain>
```

```
where
    OperatorChain: Operator<KeyValue<Key, Out>> + 'static,
{
    pub fn map<NewOut: Data, F>(
        self,
        f: F,
    ) -> KeyedStream<Key, NewOut, impl Operator<KeyValue<Key,
    ↪ NewOut>>>
    where
        F: Fn(KeyValue<&Key, Out>) -> NewOut + Send + Clone + 'static,
    {
        self.add_operator(|prev| {
            Map::new(prev, move |(k, v)| {
                let mapped_value = f((&k, v));
                (k, mapped_value)
            })
        })
    }
}
```

# Graph and Metrics Visualizer

# B

Noir provides a visualizer for both the Job Graph and the Execution Graph. It also displays useful runtime metrics collected during the execution of the stream.

Those metrics may cause some overhead, so they are disabled by default. To enable them, add the compiler flag `--features=profiler` to the command line. This will compile the support for the collection of the profiling information. Then, you have to enable the profiler in the configuration file by adding the line:

```
1 tracing_dir: ./tracing
```

After a job is executed, the profile data will be written to the directory `./tracing`. The profile data is stored in a `JSON` file, that can be uploaded to the visualizer web app.

In Figure B.1 the Job Graph for the non-associative *Wordcount* benchmark is shown. There are three blocks, one with the source that reads from the dataset file, one that performs the fold for counting the words, and the last one with the `collect_vec` sink. Each operator is shown with its name, and the links connecting them are labelled with the type of the tuples that flow between them. The links that connect the blocks have a thickness proportional to the amount of data passing through them.

In Figure B.2 the Execution Graph for the same benchmark is shown. In this case, the cluster contains two hosts, each with two cores. You can see that the first two blocks are fully replicated in all the hosts, while the last block is present only on the first host.

Other than the graph topologies, the visualizer is also able to display the runtime metrics collected during the execution. In fact, by clicking on some operators in either graphs, the visualizer displays the metrics for that operator. The details of an `EndBlock` of the first block are shown in Figure B.3. These include basic information about the operator, as well as the blocks this operator connects to. Furthermore, metrics about each connection are shown, indicating the amount of data passing through each link. In this list only aggregated metrics are displayed, but by clicking on one of them it is also possible to inspect the runtime behavior of each metric. In Figure B.4 the evolution of each metric is shown.

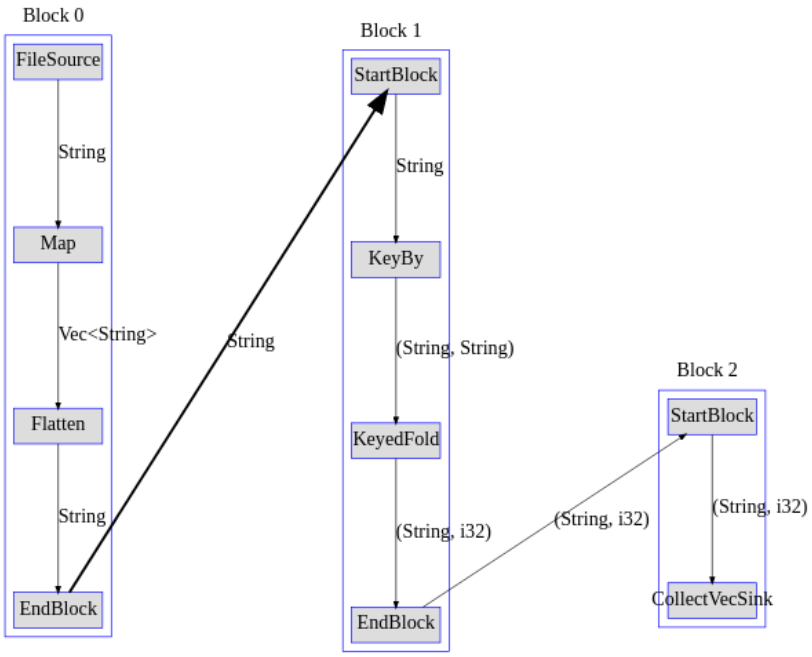



FIGURE B.1: The visualized Job Graph for the non-associative *Word-count* benchmark.

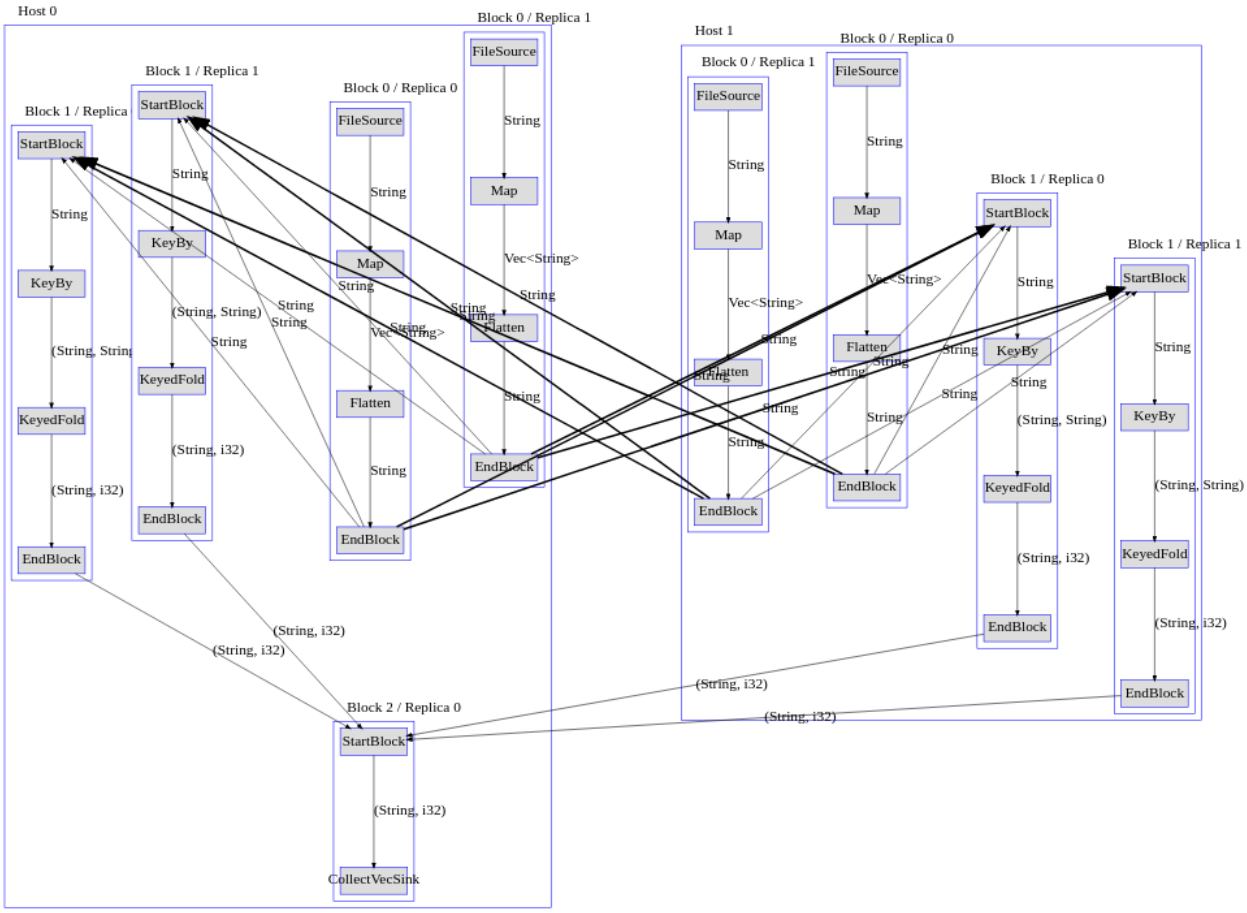



FIGURE B.2: The visualized Execution Graph for the non-associative *Wordcount* benchmark.

## Details

**Operator:** `EndBlock`

**At:** `Host0 Block0 Replica0`

**Produces:** `String`

**Connects to:**

- `Host0 Block1 Replica0` sending `String` with strategy `GroupBy`: 1,081,945 items sent
- `Host0 Block1 Replica1` sending `String` with strategy `GroupBy`: 931,987 items sent
- `Host1 Block1 Replica0` sending `String` with strategy `GroupBy`: 1,093,498 items sent (in 1069 messages, for a total of 6.8MiB)
- `Host1 Block1 Replica1` sending `String` with strategy `GroupBy`: 1,337,050 items sent (in 1307 messages, for a total of 7.6MiB)

FIGURE B.3: The metrics for the `EndBlock` of the first block, in the first instance of the first host.

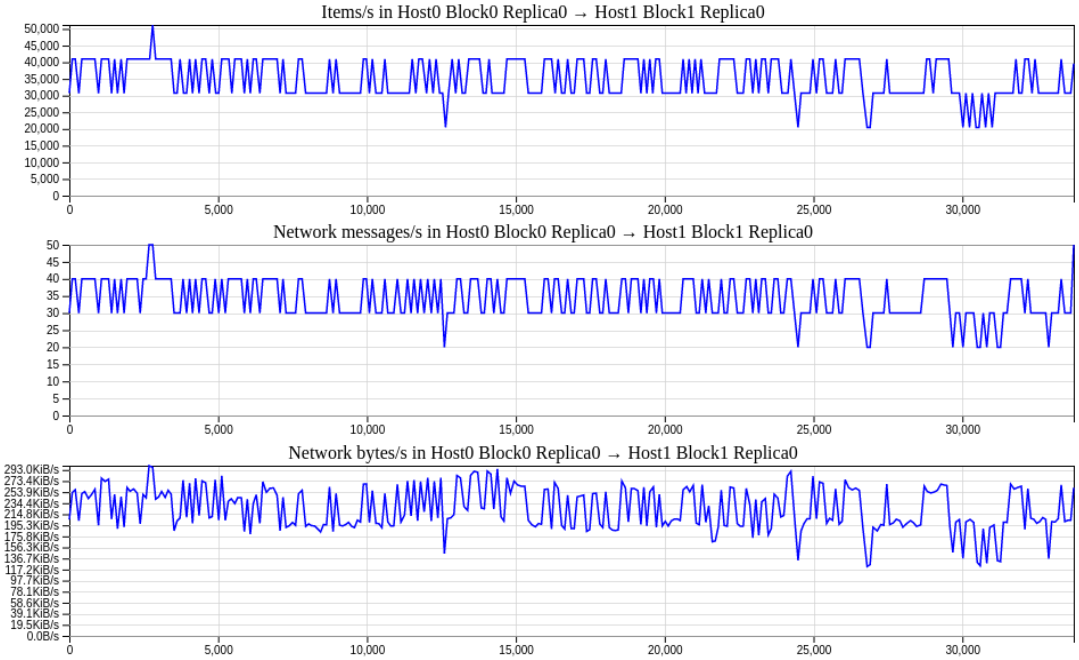



FIGURE B.4: The runtime metrics of one of the links.

# Bibliography


[1] Tyler Akidau et al. "The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing". In: *Proceedings of the VLDB Endowment* 8 (2015), pp. 1792–1803.

[2] Johannes Albrecht et al. "A Roadmap for HEP Software and Computing R&D for the 2020s". In: *Computing and Software for Big Science* 3.1 (2019). ISSN: 2510-2044. DOI: 10.1007/s41781-018-0018-8.

[3] Paris Carbone et al. "Apache Flink: Stream and batch processing in a single engine". In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.4 (2015).

[4] L. Dagum and R. Menon. "OpenMP: an industry standard API for shared-memory programming". In: *IEEE Computational Science and Engineering* 5.1 (1998), pp. 46–55. DOI: 10.1109/99.660313.

[5] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA, 2004, pp. 137–150.

[6] Alessio Fino. "Design, implementation and performance analysis of RStream, a library for distributed stream and batch processing". 2019–2020. URL: https://hdl.handle.net/10589/169285.

[7] Jeyhun Karimov et al. "Benchmarking Distributed Stream Data Processing Systems". In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 2018, pp. 1507–1518. DOI: 10.1109/ICDE.2018.00169.

[8] Daan Leijen, Ben Zorn, and Leonardo de Moura. *Mimalloc: Free List Sharding in Action*. Tech. rep. MSR-TR-2019-18. Microsoft, 2019. URL: https://www.microsoft.com/en-us/research/publication/mimalloc-free-list-sharding-in-action/.

[9] Ovidiu-Cristian Marcu et al. "Spark Versus Flink: Understanding Performance in Big Data Analytics Frameworks". In: *2016 IEEE International Conference on Cluster Computing (CLUSTER)*. 2016, pp. 433–442. DOI: 10.1109/CLUSTER.2016.22.

[10] Frank McSherry et al. "Differential Dataflow". In: *Sixth Biennial Conference on Innovative Data Systems Research, CIDR 2013, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. www.cidrdb.org, 2013. URL: http://cidrdb.org/cidr2013/Papers/CIDR13_Paper111.pdf.

[11] Derek G. Murray et al. "Naiad: A Timely Dataflow System". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: Association for Computing Machinery, 2013, 439–455. ISBN: 9781450323888. DOI: 10.1145/2517349.2522738.

[12] CORPORATE The MPI Forum. "MPI: A Message Passing Interface". In: *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*. Supercomputing '93.

Portland, Oregon, USA: Association for Computing Machinery, 1993, 878–883. ISBN: 0818643404. DOI: 10.1145/169627.169855.

[13] Matei Zaharia et al. "Spark: Cluster Computing with Working Sets". In: *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*. HotCloud'10. Boston, MA: USENIX Association, 2010, p. 10.

[14] Steffen Zeuch et al. "Analyzing Efficient Stream Processing on Modern Hardware". In: *Proc. VLDB Endow.* 12.5 (Jan. 2019), 516–530. ISSN: 2150-8097. DOI: 10.14778/3303753.3303758.

# Online resources


[15] *Amazon Web Services, Elastic Cloud Compute*. https://aws.amazon.com/ec2/.

[16] *crates.io: Rust Package Registry*. https://crates.io/.

[17] *fast-cpp-csv-parser*. https://github.com/ben-strasser/fast-cpp-csv-parser.

[18] *Flink: Java Lambda Expressions*. https://ci.apache.org/projects/flink/flink-docs-master/docs/dev/datastream/java_lambdas/.

[19] *Flink: Task Slots and Resources*. https://ci.apache.org/projects/flink/flink-docs-release-1.13/docs/concepts/flink-architecture/#task-slots-and-resources.

[20] *Motor Vehicle Collisions - Crashes | NYC Open Data*. https://data.cityofnewyork.us/Public-Safety/Motor-Vehicle-Collisions-Crashes/h9gi-nx95.

[21] *Project Gutenberg library of Public Domain books*. https://www.gutenberg.org/.

[22] *Rust Iterator trait*. https://doc.rust-lang.org/std/iter/index.html.

[23] *SNAP: Network datasets: Social circles*. https://snap.stanford.edu/data/ego-Twitter.html.

[24] *affinity*. https://docs.rs/affinity.

[25] *bincode: encoding and decoding using a tiny binary serialization strategy*. http://docs.rs/bincode.

[26] *bincode's variable-size integer encoding*. https://docs.rs/bincode/1.3.3/bincode/config/struct.VarintEncoding.html.

[27] *coarsetime: a crate to make time measurements that focuses on speed*. https://docs.rs/coarsetime/.

[28] *crossbeam: tools for concurrent programming*. https://docs.rs/crossbeam.

[29] *flume: a blazingly fast multi-producer, multi-consumer channel*. https://docs.rs/flume/.

[30] *perf: Linux profiling with performance counters*. https://perf.wiki.kernel.org/index.php/Main_Page.

[31] *sched_setaffinity system call*. https://man7.org/linux/man-pages/man2/sched_setaffinity.2.html.

[32] *The Rust Programming Language*. https://www.rust-lang.org/.

[33] *Time Stamp Counter instruction*. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-2b-manual.pdf. pp. 545–546.

[34] *Timely Dataflow GitHub repository*. https://github.com/TimelyDataflow/timely-dataflow.

[35] *Tokei: Count your code, quickly*. https://github.com/XAMPPRocky/tokei.

[36] *YAML: YAML Ain't Markup Language*. https://yaml.org/.

[37] *Yielding in crossbeam-channel · Issue #366 · crossbeam-rs/crossbeam*. https://github.com/crossbeam-rs/crossbeam/issues/366.