



**POLITECNICO**  
MILANO 1863

# Noir

Design, Implementation and Evaluation  
of a Streaming and Batch Processing Framework

Marco Donadoni   Edoardo Morassutto

2021-10-06

Noir

- Big Data: huge amounts of information to process, **in a timely manner**

- Big Data: many things to process, time constraints
- Single computer is not enough, many machines are needed
- Many problems: synchronization, communication, deployment, etc.
- Two kinds of data intensive workloads: Batch processing / Stream processing
- Batch Processing: finite dataset, results as fast as possible
- Stream processing: possibly infinite dataset, flow of tuples that need to be processed as they come (low latency)

- Big Data: huge amounts of information to process, in a timely manner
- Distributed computing

- Big Data: many things to process, time constraints
- Single computer is not enough, many machines are needed
- Many problems: synchronization, communication, deployment, etc.
- Two kinds of data intensive workloads: Batch processing / Stream processing
- Batch Processing: finite dataset, results as fast as possible
- Stream processing: possibly infinite dataset, flow of tuples that need to be processed as they come (low latency)

- Big Data: huge amounts of information to process, in a timely manner
- Distributed computing: synchronization, communication, deployment, etc.

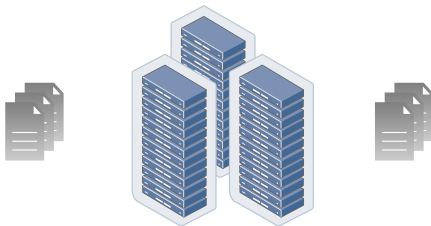
- Big Data: many things to process, time constraints
- Single computer is not enough, many machines are needed
- Many problems: synchronization, communication, deployment, etc.
- Two kinds of data intensive workloads: Batch processing / Stream processing
- Batch Processing: finite dataset, results as fast as possible
- Stream processing: possibly infinite dataset, flow of tuples that need to be processed as they come (low latency)

- Big Data: huge amounts of information to process, in a timely manner
- Distributed computing: synchronization, communication, deployment, etc.
- Two kinds of workloads:

- Big Data: huge amounts of information to process, in a timely manner
- Distributed computing: synchronization, communication, deployment, etc.
- Two kinds of workloads:

- Big Data: many things to process, time constraints
- Single computer is not enough, many machines are needed
- Many problems: synchronization, communication, deployment, etc.
- Two kinds of data intensive workloads: Batch processing / Stream processing
- Batch Processing: finite dataset, results as fast as possible
- Stream processing: possibly infinite dataset, flow of tuples that need to be processed as they come (low latency)

- Big Data: huge amounts of information to process, in a timely manner
- Distributed computing: synchronization, communication, deployment, etc.
- Two kinds of workloads: **Batch processing**

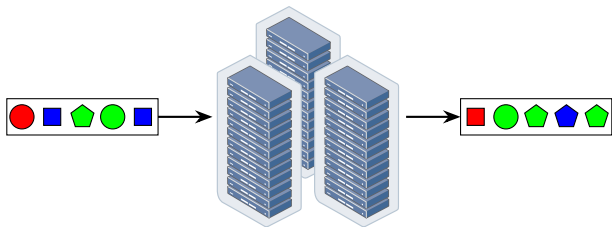


- Big Data: huge amounts of information to process, in a timely manner
- Distributed computing: synchronization, communication, deployment, etc.
- Two kinds of workloads: **Batch processing**



- Big Data: many things to process, time constraints
- Single computer is not enough, many machines are needed
- Many problems: synchronization, communication, deployment, etc.
- Two kinds of data intensive workloads: Batch processing / Stream processing
- Batch Processing: finite dataset, results as fast as possible
- Stream processing: possibly infinite dataset, flow of tuples that need to be processed as they come (low latency)

- Big Data: huge amounts of information to process, in a timely manner
- Distributed computing: synchronization, communication, deployment, etc.
- Two kinds of workloads: Batch processing / **Stream processing**



- Big Data: huge amounts of information to process, in a timely manner
- Distributed computing: synchronization, communication, deployment, etc.
- Two kinds of workloads: Batch processing / **Stream processing**



- Big Data: many things to process, time constraints
- Single computer is not enough, many machines are needed
- Many problems: synchronization, communication, deployment, etc.
- Two kinds of data intensive workloads: Batch processing / Stream processing
- Batch Processing: finite dataset, results as fast as possible
- Stream processing: possibly infinite dataset, flow of tuples that need to be processed as they come (low latency)

Develop ad-hoc solutions for each task using general purpose programming models

- Low-level communication library (e.g. MPI)

2021-10-06

Noir

└ Introduction

└ First Solution

Develop ad-hoc solutions for each task using general purpose programming models

- Low-level communication library (e.g. MPI)

- Custom ad-hoc solutions for each task
- MPI is the de facto standard for HPC
- Advantage: best performance
- Drawback: many aspects need to be manually managed
- Debugging and performance tuning is difficult



Develop ad-hoc solutions for each task using general purpose programming models

- Low-level communication library (e.g. MPI)
- Best performance possible

- Low-level communication library (e.g. MPI)
- Best performance possible

- Custom ad-hoc solutions for each task
- MPI is the de facto standard for HPC
- Advantage: best performance
- Drawback: many aspects need to be manually managed
- Debugging and performance tuning is difficult

Develop ad-hoc solutions for each task using general purpose programming models

- Low-level communication library (e.g. MPI)
- Best performance possible
  - ▶ Data is mutable in-place

- Low-level communication library (e.g. MPI)
- Best performance possible
  - ▶ Data is mutable in-place

- Custom ad-hoc solutions for each task
- MPI is the de facto standard for HPC
- Advantage: best performance
- Drawback: many aspects need to be manually managed
- Debugging and performance tuning is difficult

Develop ad-hoc solutions for each task using general purpose programming models

- Low-level communication library (e.g. MPI)
- Best performance possible
  - ▶ Data is mutable in-place
  - ▶ Optimizations tailored to the task to solve

- Custom ad-hoc solutions for each task
- MPI is the de facto standard for HPC
- Advantage: best performance
- Drawback: many aspects need to be manually managed
- Debugging and performance tuning is difficult

Develop ad-hoc solutions for each task using general purpose programming models

- Low-level communication library (e.g. MPI)
- Best performance possible
  - ▶ Data is mutable in-place
  - ▶ Optimizations tailored to the task to solve
- Manual management of many aspects of the computation

- Custom ad-hoc solutions for each task
- MPI is the de facto standard for HPC
- Advantage: best performance
- Drawback: many aspects need to be manually managed
- Debugging and performance tuning is difficult

Develop ad-hoc solutions for each task using general purpose programming models

- Low-level communication library (e.g. MPI)
- Best performance possible
  - ▶ Data is mutable in-place
  - ▶ Optimizations tailored to the task to solve
- Manual management of many aspects of the computation
  - ▶ parallelization, synchronization, communication, ...

- Custom ad-hoc solutions for each task
- MPI is the de facto standard for HPC
- Advantage: best performance
- Drawback: many aspects need to be manually managed
- Debugging and performance tuning is difficult

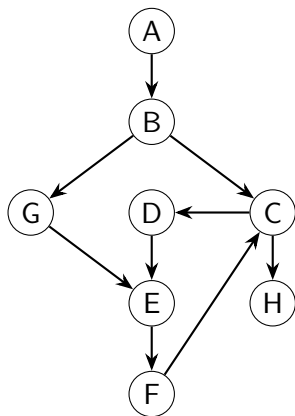
Develop ad-hoc solutions for each task using general purpose programming models

- Low-level communication library (e.g. MPI)
- Best performance possible
  - ▶ Data is mutable in-place
  - ▶ Optimizations tailored to the task to solve
- Manual management of many aspects of the computation
  - ▶ parallelization, synchronization, communication, ...
- Code is complex

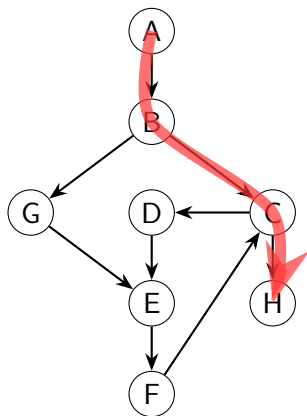
- Custom ad-hoc solutions for each task
- MPI is the de facto standard for HPC
- Advantage: best performance
- Drawback: many aspects need to be manually managed
- Debugging and performance tuning is difficult

Develop ad-hoc solutions for each task using general purpose programming models

- Low-level communication library (e.g. MPI)
- Best performance possible
  - ▶ Data is mutable in-place
  - ▶ Optimizations tailored to the task to solve
- Manual management of many aspects of the computation
  - ▶ parallelization, synchronization, communication, ...
- Code is complex

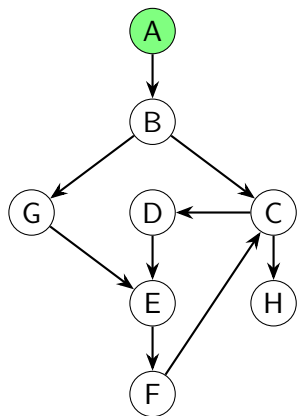


- Dataflow focuses on how data is exchanged between operators
- Each operator consumes one or more input streams and transforms them into output streams
- Dataset is not mutated in-place
- Dataflow is extended to support loops in order to implement iterative workloads
- Parallelism can be achieved by running each operator in parallel



- Dataflow focuses on how data is exchanged between operators
- Each operator consumes one or more input streams and transforms them into output streams
- Dataset is not mutated in-place
- Dataflow is extended to support loops in order to implement iterative workloads
- Parallelism can be achieved by running each operator in parallel

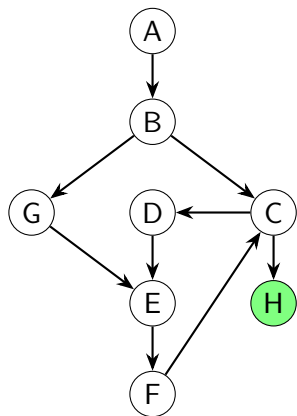




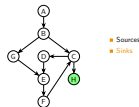
■ Sources



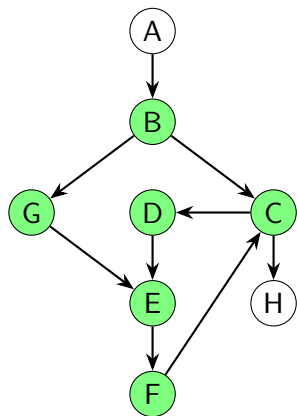
- Dataflow focuses on how data is exchanged between operators
- Each operator consumes one or more input streams and transforms them into output streams
- Dataset is not mutated in-place
- Dataflow is extended to support loops in order to implement iterative workloads
- Parallelism can be achieved by running each operator in parallel



- Sources
- Sinks



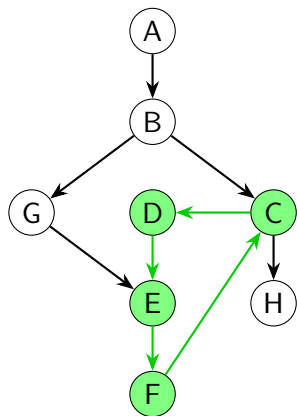
- Dataflow focuses on how data is exchanged between operators
- Each operator consumes one or more input streams and transforms them into output streams
- Dataset is not mutated in-place
- Dataflow is extended to support loops in order to implement iterative workloads
- Parallelism can be achieved by running each operator in parallel



- Sources
- Sinks
- Operators



- Dataflow focuses on how data is exchanged between operators
- Each operator consumes one or more input streams and transforms them into output streams
- Dataset is not mutated in-place
- Dataflow is extended to support loops in order to implement iterative workloads
- Parallelism can be achieved by running each operator in parallel



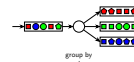
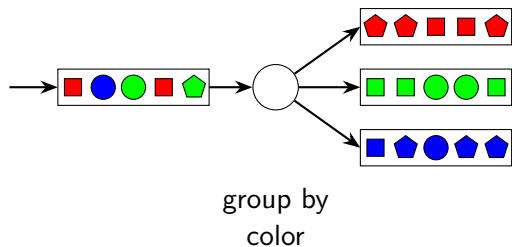
- Sources
- Sinks
- Operators
- **Loops**



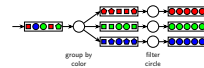
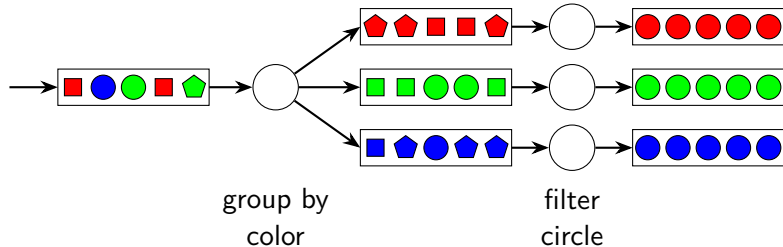
- Dataflow focuses on how data is exchanged between operators
- Each operator consumes one or more input streams and transforms them into output streams
- Dataset is not mutated in-place
- Dataflow is extended to support loops in order to implement iterative workloads
- Parallelism can be achieved by running each operator in parallel



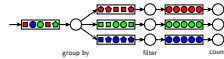
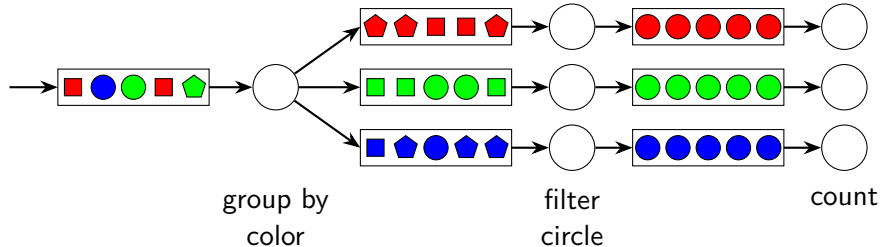
- Sometimes we need to divide the elements of a stream into groups
- The same computation is performed independently on each group
- Example: group by color
- Parallelism can be achieved by processing each substream in parallel



- Sometimes we need to divide the elements of a stream into groups
- The same computation is performed independently on each group
- Example: group by color
- Parallelism can be achieved by processing each substream in parallel



- Sometimes we need to divide the elements of a stream into groups
- The same computation is performed independently on each group
- Example: group by color
- Parallelism can be achieved by processing each substream in parallel



- Sometimes we need to divide the elements of a stream into groups
- The same computation is performed independently on each group
- Example: group by color
- Parallelism can be achieved by processing each substream in parallel



- Apache Flink and Apache Spark
- Timely Dataflow
- RStream

- Apache Flink/Spark: written in Java, high level API, widely used
- Timely Dataflow: written in Rust, does not provide many operators
- RStream: proof-of-concept written in Rust, fast but not expressive enough
- Spark is not considered because benchmarks show it performs similar to Flink
- Timely Dataflow is not considered because many operators are missing, so implementing benchmarks is difficult

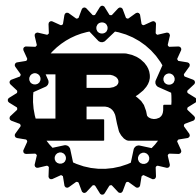
- Apache Flink and Apache Spark
- Timely Dataflow
- RStream
- Noir

- Apache Flink/Spark: written in Java, high level API, widely used
- Timely Dataflow: written in Rust, does not provide many operators
- RStream: proof-of-concept written in Rust, fast but not expressive enough
- Spark is not considered because benchmarks show it performs similar to Flink
- Timely Dataflow is not considered because many operators are missing, so implementing benchmarks is difficult

- Apache Flink and Apache Spark
- Timely Dataflow
- RStream
- Noir

- Apache Flink/Spark: written in Java, high level API, widely used
- Timely Dataflow: written in Rust, does not provide many operators
- RStream: proof-of-concept written in Rust, fast but not expressive enough
- Spark is not considered because benchmarks show it performs similar to Flink
- Timely Dataflow is not considered because many operators are missing, so implementing benchmarks is difficult

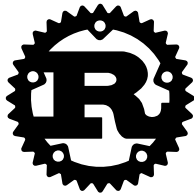
Reliable type safety, borrow checker



- Reliable: *if it compiles, it works*
- Performant: compiled language
- Productive: helpful error messages, tools to manage projects
- Transparent: *you don't pay for what you don't use*

**Reliable** type safety, borrow checker

**Performant** similar performance to C/C++



2021-10-06

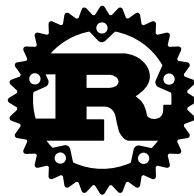
Noir  
└─ Introduction  
    └─ Rust

Reliable type safety, borrow checker  
Performant similar performance to C/C++



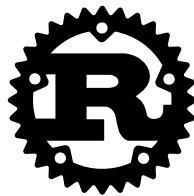
- Reliable: *if it compiles, it works*
- Performant: compiled language
- Productive: helpful error messages, tools to manage projects
- Transparent: *you don't pay for what you don't use*

**Reliable** type safety, borrow checker  
**Performant** similar performance to C/C++  
**Productive** many libraries, good tooling



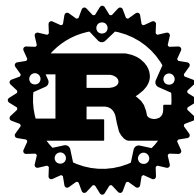
- Reliable: *if it compiles, it works*
- Performant: compiled language
- Productive: helpful error messages, tools to manage projects
- Transparent: *you don't pay for what you don't use*

**Reliable** type safety, borrow checker  
**Performant** similar performance to C/C++  
**Productive** many libraries, good tooling  
**Transparent** zero-cost abstractions



- Reliable: *if it compiles, it works*
- Performant: compiled language
- Productive: helpful error messages, tools to manage projects
- Transparent: *you don't pay for what you don't use*

- Reliable type safety, borrow checker
- Performant similar performance to C/C++
- Productive many libraries, good tooling
- Transparent zero-cost abstractions
- Versatile exposes low-level facilities



- Reliable: *if it compiles, it works*
- Performant: compiled language
- Productive: helpful error messages, tools to manage projects
- Transparent: *you don't pay for what you don't use*



- Based on the Dataflow model
- Many supported operators
- Written in Rust

2021-10-06

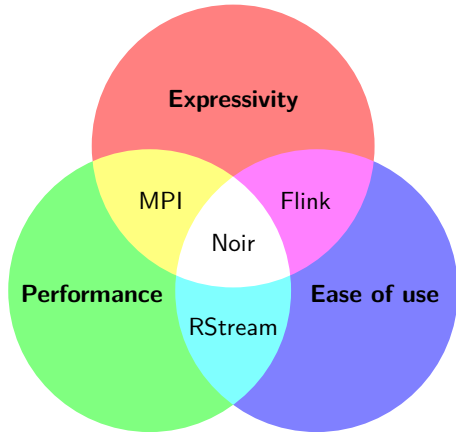
Noir

└─ Noir

└─ Noir

- Based on the Dataflow model
- Many supported operators
- Written in Rust

- Based on the Dataflow model
- Many supported operators
- Written in Rust



2021-10-06

Noir

Noir

Noir



```
let source = FileSource::new("/path/to/dataset.txt");
env.stream(source)
  .flat_map(|line| Tokenizer::tokenize(line))
  .group_by(|word| word.clone())
  .fold(0, |count, _word| *count += 1)
  .collect_vec();
```



```
let source = FileSource::new("/path/to/dataset.txt");
env.stream(source)
  .flat_map(|line| Tokenizer::tokenize(line))
  .group_by(|word| word.clone())
  .fold(0, |count, _word| *count += 1)
  .collect_vec();
```



- First example: given a text file, count how many times each word appears in it
- Source that reads the file in parallel
- Flat map that splits each line into words
- Partition the stream for each word
- Count the number of occurrences of each word
- Collect the results in an array
- The graph in the bottom is called Job Graph

```
let source = FileSource::new("/path/to/dataset.txt");
env.stream(source)
  .flat_map(|line| Tokenizer::tokenize(line))
  .group_by(|word| word.clone())
  .fold(0, |count, _word| *count += 1)
  .collect_vec();
```



```
let source = FileSource::new("/path/to/dataset.txt");
env.stream(source)
  .flat_map(|line| Tokenizer::tokenize(line))
  .group_by(|word| word.clone())
  .fold(0, |count, _word| *count += 1)
  .collect_vec();
```



- First example: given a text file, count how many times each word appears in it
- Source that reads the file in parallel
- Flat map that splits each line into words
- Partition the stream for each word
- Count the number of occurrences of each word
- Collect the results in an array
- The graph in the bottom is called Job Graph

```
let source = FileSource::new("/path/to/dataset.txt");
env.stream(source)
.flat_map(|line| Tokenizer::tokenize(line))
.group_by(|word| word.clone())
.fold(0, |count, _word| *count += 1)
.collect_vec();
```

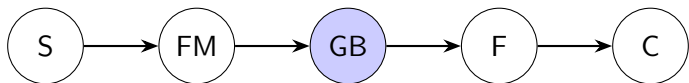


```
let source = FileSource::new("/path/to/dataset.txt");
env.stream(source)
.flat_map(|line| Tokenizer::tokenize(line))
.group_by(|word| word.clone())
.fold(0, |count, _word| *count += 1)
.collect_vec();
```



- First example: given a text file, count how many times each word appears in it
- Source that reads the file in parallel
- Flat map that splits each line into words
- Partition the stream for each word
- Count the number of occurrences of each word
- Collect the results in an array
- The graph in the bottom is called Job Graph

```
let source = FileSource::new("/path/to/dataset.txt");
env.stream(source)
  .flat_map(|line| Tokenizer::tokenize(line))
  .group_by(|word| word.clone())
  .fold(0, |count, _word| *count += 1)
  .collect_vec();
```



```
let source = FileSource::new("/path/to/dataset.txt");
env.stream(source)
  .flat_map(|line| Tokenizer::tokenize(line))
  .group_by(|word| word.clone())
  .fold(0, |count, _word| *count += 1)
  .collect_vec();
```



- First example: given a text file, count how many times each word appears in it
- Source that reads the file in parallel
- Flat map that splits each line into words
- Partition the stream for each word
- Count the number of occurrences of each word
- Collect the results in an array
- The graph in the bottom is called Job Graph

```
let source = FileSource::new("/path/to/dataset.txt");
env.stream(source)
  .flat_map(|line| Tokenizer::tokenize(line))
  .group_by(|word| word.clone())
  .fold(0, |count, _word| *count += 1)
  .collect_vec();
```

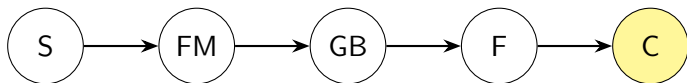


```
let source = FileSource::new("/path/to/dataset.txt");
env.stream(source)
  .flat_map(|line| Tokenizer::tokenize(line))
  .group_by(|word| word.clone())
  .fold(0, |count, _word| *count += 1)
  .collect_vec();
```



- First example: given a text file, count how many times each word appears in it
- Source that reads the file in parallel
- Flat map that splits each line into words
- Partition the stream for each word
- Count the number of occurrences of each word
- Collect the results in an array
- The graph in the bottom is called Job Graph

```
let source = FileSource::new("/path/to/dataset.txt");
env.stream(source)
  .flat_map(|line| Tokenizer::tokenize(line))
  .group_by(|word| word.clone())
  .fold(0, |count, _word| *count += 1)
  .collect_vec();
```



```
let source = FileSource::new("/path/to/dataset.txt");
env.stream(source)
  .flat_map(|line| Tokenizer::tokenize(line))
  .group_by(|word| word.clone())
  .fold(0, |count, _word| *count += 1)
  .collect_vec();
```



- First example: given a text file, count how many times each word appears in it
- Source that reads the file in parallel
- Flat map that splits each line into words
- Partition the stream for each word
- Count the number of occurrences of each word
- Collect the results in an array
- The graph in the bottom is called Job Graph



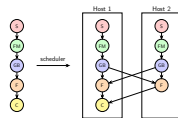
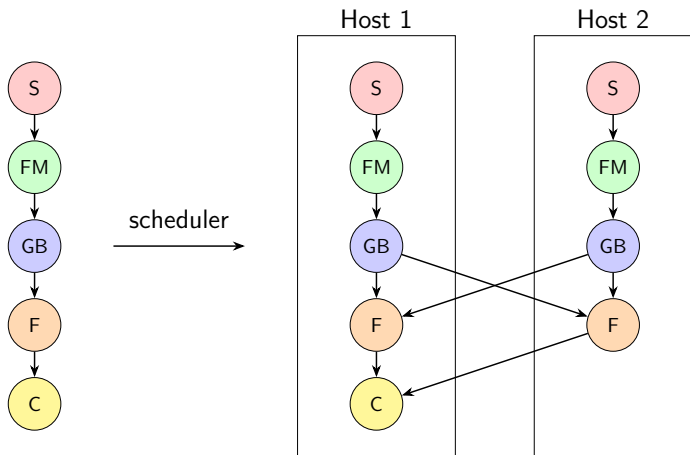
```
let source = FileSource::new("/path/to/dataset.txt");
env.stream(source)
  .flat_map(|line| Tokenizer::tokenize(line))
  .group_by(|word| word.clone())
  .fold(0, |count, _word| *count += 1)
  .collect_vec();
```



```
let source = FileSource::new("/path/to/dataset.txt");
env.stream(source)
  .flat_map(|line| Tokenizer::tokenize(line))
  .group_by(|word| word.clone())
  .fold(0, |count, _word| *count += 1)
  .collect_vec();
```

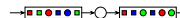
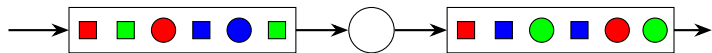


- First example: given a text file, count how many times each word appears in it
- Source that reads the file in parallel
- Flat map that splits each line into words
- Partition the stream for each word
- Count the number of occurrences of each word
- Collect the results in an array
- The graph in the bottom is called Job Graph



- On the left: previous job graph
- Scheduler's job is to build the execution graph
- Duplicating and allocating the operators in the hosts
- Sources read in parallel, two independent streams
- Group by has to move data between hosts so that same word goes to same operator

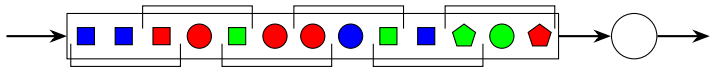
Basic map, filter, fold, reduce, group\_by, ...



- Basic operators transform one stream into another
- Windows make possible to execute operations on unbounded streams by slicing them
- Join merge two streams into one
- Iterations make data recirculate in a loop
- Point being that expressivity is one of our goals

Basic map, filter, fold, reduce, group\_by, ...

Windows event time, processing time, count, sliding, tumbling, session, ...

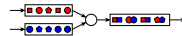
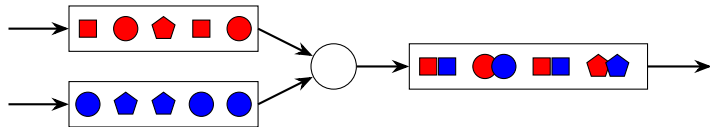


- Basic operators transform one stream into another
- Windows make possible to execute operations on unbounded streams by slicing them
- Join merge two streams into one
- Iterations make data recirculate in a loop
- Point being that expressivity is one of our goals

**Basic** map, filter, fold, reduce, group\_by, ...

**Windows** event time, processing time, count, sliding, tumbling, session, ...

**Joins** inner, outer, ship strategies, local strategies, ...



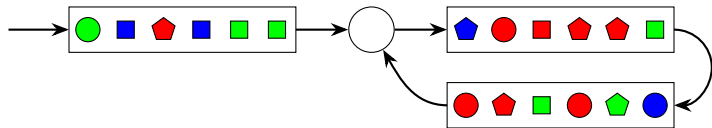
- Basic operators transform one stream into another
- Windows make possible to execute operations on unbounded streams by slicing them
- Join merge two streams into one
- Iterations make data recirculate in a loop
- Point being that expressivity is one of our goals

**Basic** map, filter, fold, reduce, group\_by, ...

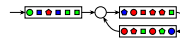
**Windows** event time, processing time, count, sliding, tumbling, session, ...

**Joins** inner, outer, ship strategies, local strategies, ...

**Iterations** side-input, iteration state, nested iterations, ...

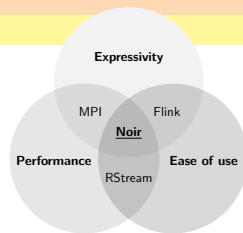


Basic map, filter, fold, reduce, group\_by, ...  
Windows event time, processing time, count, sliding, tumbling, session, ...  
Joins inner, outer, ship strategies, local strategies, ...  
Iterations side-input, iteration state, nested iterations, ...



- Basic operators transform one stream into another
- Windows make possible to execute operations on unbounded streams by slicing them
- Join merge two streams into one
- Iterations make data recirculate in a loop
- Point being that expressivity is one of our goals

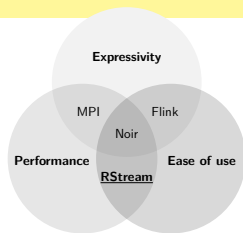
```
fn main() {
  let (config, args) = EnvironmentConfig::from_args();
  let mut env = StreamEnvironment::new(config);
  env.spawn_remote_workers();
  let path = args.nth(1).expect("Missing dataset path");
  let result = env
    .stream(FileSource::new(path))
    .flat_map(|line| Tokenizer::tokenize(line))
    .group_by(|word| word.clone())
    .fold(0, |count, _word| *count += 1)
    .collect_vec();
  env.execute();
  if let Some(res) = result.get() {
    eprintln!("Output: {:?}", res);
  }
}
```



```
fn main() {
  let (config, args) = EnvironmentConfig::from_args();
  let mut env = StreamEnvironment::new(config);
  env.spawn_remote_workers();
  let path = args.nth(1).expect("Missing dataset path");
  let result = env
    .stream(FileSource::new(path))
    .flat_map(|line| Tokenizer::tokenize(line))
    .group_by(|word| word.clone())
    .fold(0, |count, _word| *count += 1)
    .collect_vec();
  env.execute();
  if let Some(res) = result.get() {
    eprintln!("Output: {:?}", res);
  }
}
```

- Let's compare the implementation of wordcount in the various frameworks
- Noir: some boilerplate before and after the application logic, but the code is coherent
- RStream: the same, little boilerplate, code very compact
- Flink: again
- MPI: around 200 LoC, less readable code, logic is mixed with communication

```
fn main() {
    let path: String = env::args()
        .nth(1)
        .expect("Missing dataset path");
    let word_count = Stream::from_readlines(&path)
        .flat_map(|line| Tokenizer::tokenize(line))
        .group_by(|(word, _count)| word.clone())
        .reduce(|(word, c1), (_word, c2)| (word, c1 + c2))
        .collect_vec();
    finalize();
    println!("{:?}", word_count);
    Ok(())
}
```



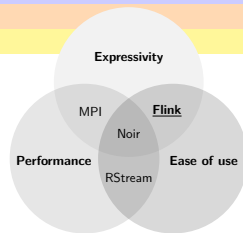
```
fn main() {
    let path: String = env::args()
        .nth(1)
        .expect("Missing dataset path");
    let word_count = Stream::from_readlines(&path)
        .flat_map(|line| Tokenizer::tokenize(line))
        .group_by(|(word, _count)| word.clone())
        .reduce(|(word, c1), (_word, c2)| (word, c1 + c2))
        .collect_vec();
    finalize();
    println!("{:?}", word_count);
    Ok(())
}
```

- Let's compare the implementation of wordcount in the various frameworks
- Noir: some boilerplate before and after the application logic, but the code is coherent
- RStream: the same, little boilerplate, code very compact
- Flink: again
- MPI: around 200 LoC, less readable code, logic is mixed with communication



```
public static void main(String[] args) {
    MultipleParameterTool params = MultipleParameterTool.fromArgs(args);
    ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
    env.getConfig().setGlobalJobParameters(params);
```

```
    DataSet<Tuple2<String, Integer>> counts = env
        .readTextFile(params.get("input"));
        .flatMap(new Tokenizer())
        .groupBy(0) // group by word
        .sum(1);    // sum the counts
    counts.count();
}
```



```
public static void main(String[] args) {
    MultipleParameterTool params = MultipleParameterTool.fromArgs(args);
    ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
    env.getConfig().setGlobalJobParameters(params);

    DataSet<Tuple2<String, Integer>> counts = env
        .readTextFile(params.get("input"));
        .flatMap(new Tokenizer())
        .groupBy(0) // group by word
        .sum(1);    // sum the counts
    counts.count();
}
```

- Let's compare the implementation of wordcount in the various frameworks
- Noir: some boilerplate before and after the application logic, but the code is coherent
- RStream: the same, little boilerplate, code very compact
- Flink: again
- MPI: around 200 LoC, less readable code, logic is mixed with communication

# Noir

## API Comparison – MPI

11/19

```
void initBuffer() {
    int blockLength[] = {20, 20, 20};
    MPI_Aint offsets[] = {0, 0, 0};
    MPI_Datatype types[] = {MPI_CHAR, MPI_UNSIGNED_SHORT, MPI_UNSIGNED_SHORT};
    MPI_Type_create_struct(3, blockLength, offsets, types, userDataType);
    MPI_Type_commit(userDataType);
}

template <typename T> std::vector<T> receiveVector(int source, int tag) {
    MPI_Status status;
    MPI_Message msg;
    MPI_Pack(source, tag, MPI_CHAR, userDataType, status);
    int length;
    MPI_Get_count(status, userDataType, length);
    MPI_Get_count(status, userDataType, length);
    // Allocate buffer and receive result
    std::vector<T> result(length);
    MPI_Recv(result.data(), length, userDataType, source, MPI_SOURCE,
            status, MPI_TAG, MPI_COMM_WORLD, status);
    return result;
}

result_t associateData(mapped, size_t start, size_t end, size_t fileIndex) {
    size_t pos = start;
    if (start == 0) {
        char c = mapped[pos];
        while (pos < fileIndex || c == '\n') {
            c = mapped[pos];
        }
    }
    result_t count;
    if (pos == fileIndex || pos == end)
        return count;
    std::string cur;
    char c = mapped[pos];
    while (pos < fileIndex || (c == '\n' || pos == end)) {
        if (c == '\n' || c == '\t') {
            cur = std::to_string(c);
        } else {
            if (cur.empty()) {
                count[pos]++;
            }
            cur += c;
        }
        pos = mapped[pos];
    }
    if (cur.empty()) {
        count[pos]++;
    }
    return count;
}

result_t merge(result_t A, const result_t B) {
    for (auto [k, v] : B)
        A[k] += v;
    return A;
}

result_t associateMap(size_t rank, size_t numProcesses, size_t numThreads,
                    std::string datasetPath) {
    count size_t datasetSize = std::filesystem::file_size(datasetPath);
    count size_t processChunk = (datasetSize + numProcesses - 1) / numProcesses;
    count size_t threadChunk = (processChunk + numThreads - 1) / numThreads;
    result_t result;
    auto fd = open(datasetPath.c_str(), O_RDONLY);
    char *mapped = (char *)mmap(NULL, datasetSize, PROT_READ, MAP_SHARED, fd, 0);
    // Prepare map declare reduction: result_t r; map_out = merge(map_out, map_in);
    // Prepare map parallel for: auto dataStatus; if (reduction == result)
    for (size_t th = 0; th < numThreads; th++) {
        size_t start = processChunk * rank + threadChunk * th;
        size_t end = start + threadChunk;
        printf(stderr, "[PID%04d] has interval [%d - %d] %s", rank, th, start, end);
        result = associate(mapped, start, end, datasetPath);
        printf(stderr, "[PID%04d] has interval [%d - %d] -- done", rank, th, start, end);
    }
    return result;
}

int main(int argc, char **argv) {
    int rank;
    int numProcesses;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &numProcesses);
    initBuffer();
    std::string filePath = "data/getmsg0.txt";
    std::string method = "map";
    count int numThreads = get_opt_max_threads();
    // Intermediate step
    // Send intermediate results to other nodes, based on the hash of the word
    // Prepare the buffers to be sent
    std::vector<std::vector<Word>> flatMap(numProcesses);
    for (auto [k, v] : result) {
        Word word;
        std::string cur;
        word.set(filePath, k, v);
        word.set(RR_SIZE - 1) = '\0';
        word.count = v;
    }
    size_t h = std::hash<std::string>{}(k);
    flatMap[h % numProcesses].push_back(word);
}
```

```
// Send the buffers
std::vector<MPI_Request> requests(numProcesses);
for (int i = 0; i < numProcesses; i++) {
    if (rank == 0) {
        MPI_Isend(flatMap[i].data(), flatMap[i].size(), userDataType, 1, 1,
                MPI_COMM_WORLD, requests[i]);
        printf(stderr, "[PID %04d] Process %d has sent %d records to %d", rank,
                rank, flatMap[i].size(), 1);
    }
}

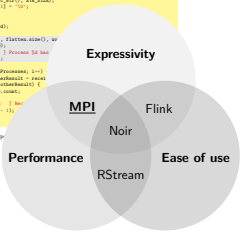
// Receive buffers from others
std::vector<Word> all = std::move(flatMap[rank]);
for (int i = 0; i < numProcesses; i++) {
    if (rank != 0) {
        std::vector<Word> received = receiveVector(rank, 1, 1);
        printf(stderr, "[PID %04d] Received %d records from %d", rank,
                i);
        all.insert(all.begin(), received.begin(), received.end());
    }
}

// Merge the results
result.clear();
for (Word w : all) {
    result[w.word] += w.count;
}

// Wait on all the threads, so that the buffers can be safely deallocated
for (int i = 0; i < numProcesses; i++) {
    if (rank != 0) {
        MPI_Wait(requests[i], MPI_STATUS_IGNORE);
    }
}

if (rank == 0) {
    std::vector<Word> flatMap;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &numProcesses);
    Word word;
    std::string filePath = "data/getmsg0.txt";
    word.set(filePath, k, v);
    word.set(RR_SIZE - 1) = '\0';
    word.count = v;
    flatMap.push_back(word);
    MPI_Isend(flatMap.data(), flatMap.size(), userDataType, 1, 1,
            MPI_COMM_WORLD, requests);
    printf(stderr, "[PID %04d] Process %d has %d records to %d",
            rank, rank, flatMap.size(), 1);
} else {
    for (int i = 1; i < numProcesses; i++) {
        std::vector<Word> otherRank = receiveVector(rank, 1, 1);
        count count = w.count;
        result[w.word] += w.count;
    }
}

MPI_Type_create_struct(1,
    {flatMap[rank % numProcesses].push_back(word);
}
```



2021-10-06


Noir  
└─ Noir

## └─ API Comparison – MPI



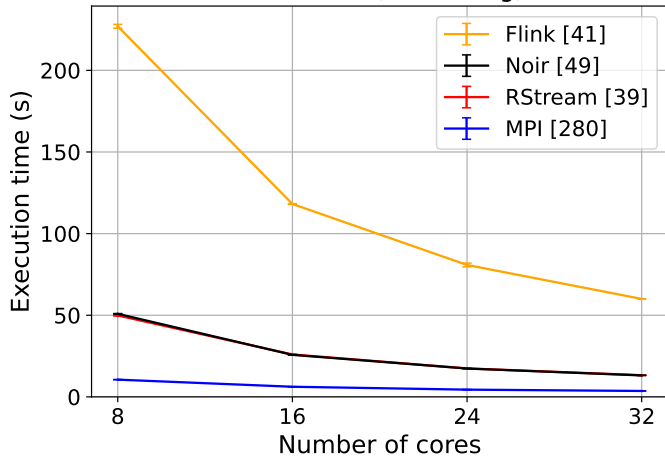
- Let's compare the implementation of wordcount in the various frameworks
- Noir: some boilerplate before and after the application logic, but the code is cohered
- RStream: the same, little boilerplate, code very compact
- Flink: again
- MPI: around 200 LoC, less readable code, logic is mixed with communication

Machine type	4x 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
RAM	16 GiB
Network	5 Gbps
Ping	0.12 ms
Cost	1.36 \$/h (4 VMs)

Machine type	4x 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	
RAM	16 GiB	
Network	5 Gbps	
Ping	0.12 ms	
Cost	1.36 \$/h (4 VMs)	

- Rented 4 VM on AWS
- 8 threads each with a fast network
- This is a very typical infrastructure for data intensive applications
- We tested the system under 11 benchmarks, we only show a subset of them

Wordcount (Gutenberg)

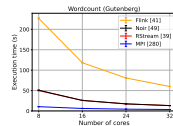


2021-10-06

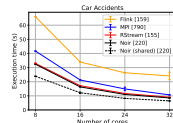
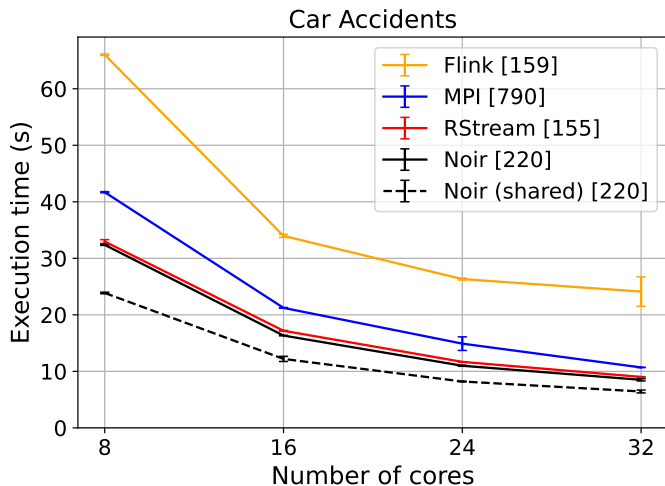
Noir

Performance Evaluation

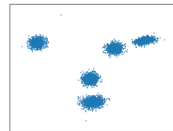
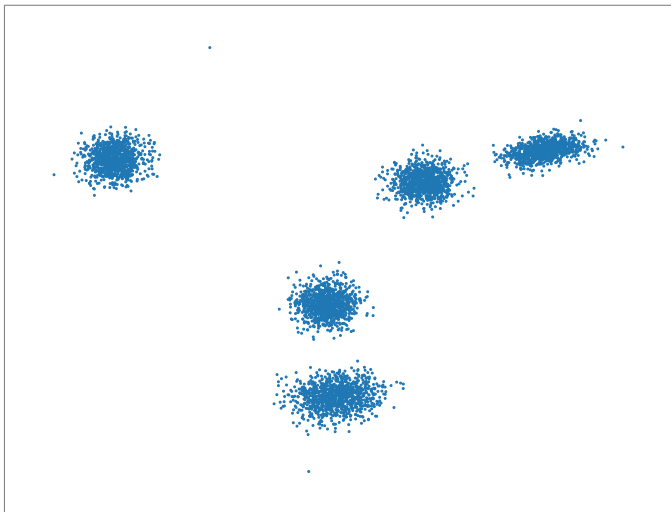
Wordcount



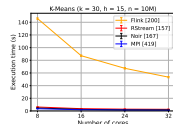
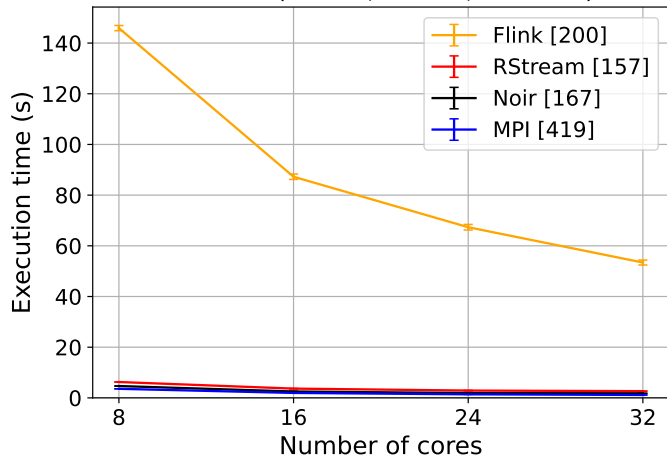
- This is a slight variation of the previous wordcount, the classical first benchmark that many uses
- The dataset is called Gutenberg, text from many books (4GB, 100K distinct words)
- Noir and RStream have the exact same performance
- MPI is faster but Flink is much slower
- The numbers in the legend are the number of lines of code



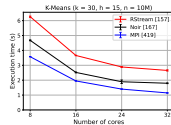
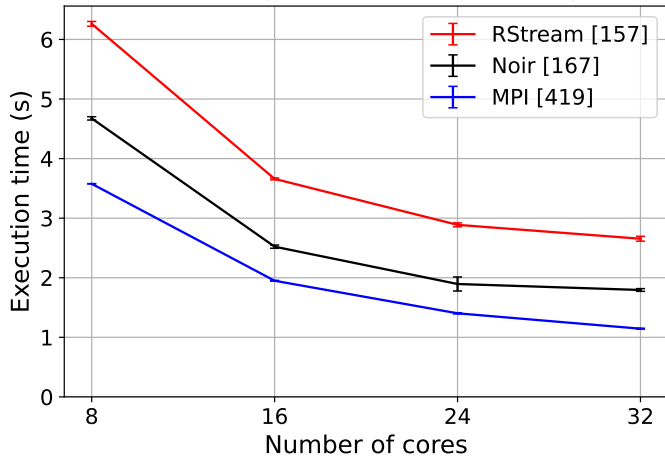
- The next benchmark tries to represent a real world application
- The dataset is a CSV with 24M car accidents in NYC
- 3 queries of different difficulty
- RStream is forced to run them one after the other, reading the dataset 3 times
- When Noir does the same, it is as fast
- Noir does not have this limitation and can run the queries in parallel
- Even though we tried hard (see the number of lines) MPI is a bit slower
- This shows that MPI does not guarantee the performance but optimizations and fine-tuning may be required



- Classical application: find the best clustering of a series of data points
- We chose this benchmark because it's a very popular *iterative* algorithm
- $k = \#$  of clusters,  $h = \#$  of iterations,  $n = \#$  of points
- Flink is much slower than the others because the garbage collector struggles to keep up with so many allocations
- Noir is faster than RStream, meaning that the iterations are more optimized

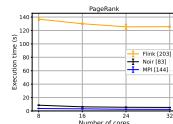
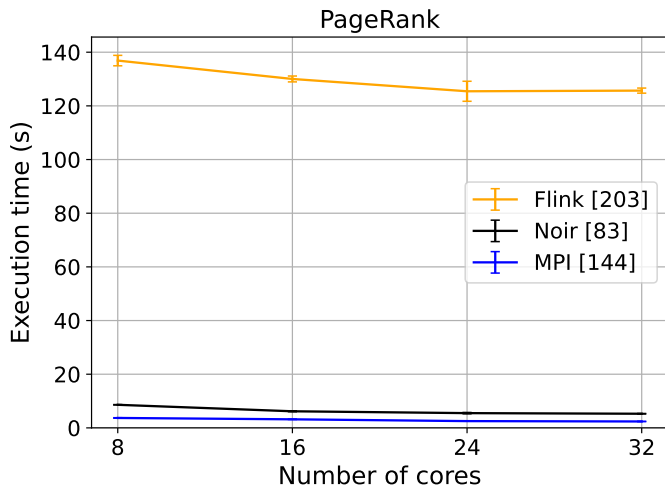
K-Means ( $k = 30$ ,  $h = 15$ ,  $n = 10M$ )

- Classical application: find the best clustering of a series of data points
- We chose this benchmark because it's a very popular *iterative* algorithm
- $k = \#$  of clusters,  $h = \#$  of iterations,  $n = \#$  of points
- Flink is much slower than the others because the garbage collector struggles to keep up with so many allocations
- Noir is faster than RStream, meaning that the iterations are more optimized

K-Means ( $k = 30, h = 15, n = 10M$ )

- Classical application: find the best clustering of a series of data points
- We chose this benchmark because it's a very popular *iterative* algorithm
- $k = \#$  of clusters,  $h = \#$  of iterations,  $n = \#$  of points
- Flink is much slower than the others because the garbage collector struggles to keep up with so many allocations
- Noir is faster than RStream, meaning that the iterations are more optimized





- Classical real world application: find the page rank of the nodes of a graph
- Iterative workload that stresses many aspects: iteration state for the ranks, side input, join in the loop
- RStream lacks many of these features, so it cannot be implemented with it
- Flink is a lot slower than the others, and it does not scale
- Noir is pretty close to MPI in comparison
- Note:** The benchmark we've shown are all batch processing, but in the thesis you can find also streaming workloads and latency analysis

Noir performance is ...

- much better than Flink, up to 30×

2021-10-06

Noir

└─ Conclusions

└─ Conclusions

Noir performance is ...  
■ much better than Flink, up to 30×

Noir performance is ...

- much better than Flink, up to 30×
- very similar to RStream, but Noir has many more features

2021-10-06

Noir  
└─ Conclusions

└─ Conclusions

Noir performance is ...  
■ much better than Flink, up to 30×  
■ very similar to RStream, but Noir has many more features

Noir performance is ...

- much better than Flink, up to 30×
- very similar to RStream, but Noir has many more features
- similar to MPI in some workloads, a bit worse in others, but Noir is much easier to use

2021-10-06

Noir  
└─ Conclusions

└─ Conclusions

Noir performance is ...

- much better than Flink, up to 30×
- very similar to RStream, but Noir has many more features
- similar to MPI in some workloads, a bit worse in others, but Noir is much easier to use

Noir performance is ...

- much better than Flink, up to 30×
- very similar to RStream, but Noir has many more features
- similar to MPI in some workloads, a bit worse in others, but Noir is much easier to use

Noir is able to achieve a better trade off between ease-of-use, expressivity and performance than what is achievable with existing systems

2021-10-06

Noir  
└─ Conclusions

└─ Conclusions

Noir performance is ...

- much better than Flink, up to 30×
- very similar to RStream, but Noir has many more features
- similar to MPI in some workloads, a bit worse in others, but Noir is much easier to use

Noir is able to achieve a better trade off between ease-of-use, expressivity and performance than what is achievable with existing systems

- Fault tolerance
- Extensions with higher level API
- Support for hybrid architectures (e.g. GPUs)

2021-10-06

Noir

└─ Conclusions

└─ Future Work

- Fault tolerance
- Extensions with higher level API
- Support for hybrid architectures (e.g. GPUs)

- Biggest missing feature that Flink has is Fault Tolerance
- Ext: SQL like interface for expressing the queries, pre-written ML algorithms (MLlib, GraphX)
- Try to exploit hybrid architectures, for example trying to add graphic cards for accelerating operators



# Noir

Design, Implementation and Evaluation  
of a Streaming and Batch Processing Framework

Marco Donadoni   Edoardo Morassutto

**POLITECNICO MILANO 1863**

2021-10-06

Noir

└ Thanks for your attention!



Noir  
Design, Implementation and Evaluation  
of a Streaming and Batch Processing Framework  
Marco Donadoni   Edoardo Morassutto  
POLITECNICO MILANO 1863