ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

LUCAS MACHADO ALF

FAULT TOLERANCE FOR HIGH-LEVEL PARALLEL AND DISTRIBUTED STREAM PROCESSING IN C++

Porto Alegre
2025

PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
SCHOOL OF TECHNOLOGY
COMPUTER SCIENCE GRADUATE PROGRAM

FAULT TOLERANCE FOR HIGH-LEVEL PARALLEL AND DISTRIBUTED STREAM PROCESSING IN C++

LUCAS MACHADO ALF

Master Thesis submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Prof. Dr. Dalvan Jair Griebler

Porto Alegre
2025

Ficha Catalográfica

A384f Alf, Lucas Machado

Fault tolerance for high-level parallel and distributed stream processing in C++ / Lucas Machado Alf. – 2025.

108.

Dissertação (Mestrado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. Dalvan Jair Griebler.

1. Parallel programming. 2. Domain-specific language. 3. Resilience. 4. Linear pipeline. 5. Stream parallelism. I. Griebler, Dalvan Jair. II. Título.

LUCAS MACHADO ALF

FAULT TOLERANCE FOR HIGH-LEVEL PARALLEL AND DISTRIBUTED STREAM PROCESSING IN C++

This Master Thesis has been submitted in partial fulfillment of the requirements for the degree of Master in Computer Science of the Computer Science Graduate Program, School of Technology of the Pontifical Catholic University of Rio Grande do Sul

Sanctioned on March 24th, 2025.

COMMITTEE MEMBERS:

Prof. Dr. Mauricio Aronne Pillon (PPGCA/UDESC)

Prof. Dr. Fernando Luís Dotti (PPGCC/PUCRS)

Prof. Dr. Dalvan Jair Griebler (PPGCC/PUCRS - Advisor)

ACKNOWLEDGMENTS

I want to thank my advisor for all the support and guidance throughout the development of this Master's thesis. I also extend my thanks to my friends and research group colleagues, whose contributions, exchanges of ideas, and discussions were invaluable to the advancement of this research. I also want to acknowledge the financial support from MCTIC/CNPq process № 131062/2023-9 and SAP Enterprise.

TOLERÂNCIA A FALHAS PARA PROCESSAMENTO STREAM PARALELO E DISTRIBUÍDO DE ALTO NÍVEL EM C++

RESUMO

O processamento de *stream* é um paradigma computacional voltado para a coleta, o processamento e a análise de fluxos contínuos de dados heterogêneos em grande volume, com o objetivo de extrair informações valiosas em tempo real. Considerando que esses sistemas precisam ser executados por longos períodos, às vezes indefinidamente, realizar o reprocessamento por completo dos dados em caso de falha pode ser extremamente custoso ou até inviável. Sendo assim, é fundamental que um sistema de processamento de *stream* não apenas se recupere de falhas, mas também garanta a integridade dos resultados. A SPar consiste em uma linguagem de domínio específico para C++ baseada em anotações, projetada para simplificar o desenvolvimento de aplicações de processamento de *stream* para múltiplas arquiteturas paralelas. Em arquiteturas distribuídas, a SPar gera código utilizando a DSParLib, que não fornece mecanismos de tolerância a falhas nem garantias de entrega de mensagens. Outras alternativas para programação paralela existentes na literatura, como o MPI, são APIs de baixo nível com suporte restrito para a implementação de aplicações paralelas resilientes. Grande parte das ferramentas no atual estado da arte em relação a programação paralela e distribuída para aplicações de *stream* com suporte à resiliência está disponível em Java. Diante desses fatores, o objetivo principal desta dissertação de mestrado consiste em investigar mecanismos de tolerância a falhas e semântica *exactly-once* para sistemas de processamento de *stream*, de forma que possam ser suportados no ecossistema de *software* SPar sem a necessidade de reescrever o código-fonte do usuário. Para esse fim, criamos a ResiPipe, uma biblioteca em C++ para o processamento de stream distribuído e tolerante a falhas, que se tornou parte do ecossistema SPar para executar código paralelo. Os resultados obtidos demonstram que a ResiPipe apresenta desempenho comparável, em alguns casos superior, a outras bibliotecas de processamento de stream analisadas, além de apresentar um menor número de linhas de código fonte (SLOC) por aplicação e uma estimativa de tempo de desenvolvimento reduzida, conforme o método de Halstead.

Palavras-Chave: Programação Paralela, Linguagem de Domínio Específico, Garantias de entrega de mensagem, Pipeline Linear, Paralelismo de Stream.

FAULT TOLERANCE FOR HIGH-LEVEL PARALLEL AND DISTRIBUTED STREAM PROCESSING IN C++

ABSTRACT

Stream processing is a computing paradigm that addresses the gathering, processing, and analysis of a high-volume heterogeneous continuous data stream, aiming to extract valuable information in real-time. Considering the need for these systems to run for long periods, possibly indefinitely, reprocessing all data in case of failure can be highly costly or even unfeasible. Thus, it is essential for a stream processing system not only to recover after a failure but also to ensure that the generated results are correct. SPar is an annotation-based C++ domain-specific language designed to simplify the development of stream processing applications for multiple parallel architectures. In distributed architectures, SPar generates code using DSParLib, which does not provide fault tolerance mechanisms or strong message delivery guarantees. Other parallel programming alternatives in the literature, such as MPI, are low-level APIs with restrictive support for implementing resilient parallel applications. Most state-of-the-art distributed parallel programming tools for streaming applications with resilience support are available in Java. Given these factors, the main objective of this master's thesis is to investigate fault tolerance and exactly-once semantics for streaming systems so that it is possible to support it on the SPar software ecosystem without rewriting the user source code. To this end, we created ResiPipe, a C++ library for fault-tolerant distributed stream processing that became a runtime system from the SPar ecosystem to execute parallel code. The results demonstrate that ResiPipe delivers performance comparable to, and in some cases superior to, other analyzed stream processing libraries, in addition to requiring fewer source lines of code (SLOC) per application and presenting a reduced estimated development time, according to Halstead's method.

Keywords: Parallel Programming, Domain-Specific Language, Resilience, Message delivery guarantees, Linear Pipeline, Stream Parallelism.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

ABS – Asynchronous Barrier Snapshotting

AST – Abstract Syntax Tree

CINCLE – Compiler Infrastructure for new C/C++ Languages Extensions

CUDA – Compute Unified Device Architecture

DSL – Domain-Specific Language

DSPARLIB – Distributed Stream Parallelism Library

EXT4 – Fourth extended file system

GCC – GNU Compiler Collection

GMAP – Parallel Applications Modeling Group

GPU – Graphics Processing Unit

GSPARLIB – GPU Stream Parallelism Library

HDFS – Hadoop distributed file system

MPI – Message Passing Interface

MPR – Message Passing Runtime

NFS – Network file system

NTFS – New technology file system

PUCRS – Pontifical Catholic University of Rio Grande do Sul

SLOC – Source lines of code

SPAR – Stream Parallelism

WAL – Write-Ahead Log

CONTENTS

1. INTRODUCTION

This chapter introduces the context, motivation, goals, and scientific contributions achieved in this Master's Thesis. Finally, the chapter concludes with a summary of the contents presented in this document.

1.1 Context and motivation

Stream processing can be defined as a computing paradigm that involves the gathering, processing, and analysis of a high-volume heterogeneous continuous stream of data, aiming to extract real-time insights or valuable information. It results from the last 50 years of constant evolution in technologies used to store, organize, and analyze the increasing amount of data organizations generate [9], and can be applied over a diverse variety of applications such as fraud detection, machine learning, and intelligent vehicles.

Before the popularization of stream processing, the batch processing model was already widely deployed for handling high volumes of data. In the batch processing model, the records are grouped into batches before processing, varying from a fixed number of records to a value based on a time window, such as a minute, an hour, or a whole day of accumulated records. Due to this accumulation process, the batch processing model is traditionally associated with higher latency, ranging from minutes to several hours, directly related to the batch size and the arrival of the last record[41].

In contrast, the stream processing model aims to process the data as soon as it arrives, without the artificial delay generated by grouping records into batches. However, according to Akidau, Chernyak, and Lax [3], stream processing systems are historically associated with lower latency at the expense of inaccurate or speculative results. This situation led to the emergence of the Lambda Architecture, created by Nathan Marz (the creator of Apache Storm), which involves the simultaneous execution of a stream and batch processing systems, both performing the same operations. While the stream processing system provides low latency imprecise results (due to the use of approximation algorithms or because the system itself does not provide correctness), a batch processing system simultaneously runs overnight, eventually providing correct results. However, this architecture comes with a series of drawbacks, as it requires building, provisioning, and maintaining two independent versions of the same pipeline and somehow combining the results of the two applications.

Given the drawbacks of the Lambda Architecture, new data processing systems such as MillWheel [1] and Google Dataflow [2] were developed with the idea of unifying stream and batch processing under a single model with strong consistency guarantees. Meanwhile, other systems like Apache Spark adopted hybrid models such as micro-batching, aiming for a balance between the lower latency of the stream processing and the high throughput of the batch processing.

1.2 Research Problem

Nowadays, a great part of the stream processing systems found at the state-of-the-art such as Apache Flink [13], Apache Spark Streaming [10] and Structured Streaming [11], Apache Storm [68], ChronoStream [70], Google DataFlow [2], Naiad [54], Streamscope [45] and TimeStream [58] are developed over high-level programming languages such as Go, Java and C#. However, for stream processing applications with strict latency and throughput requirements, it becomes appealing to write the applications using a programming language traditionally associated with the field of HPC (High-Performance Computing), such as C, C++, and Fortran.

Due to the challenges associated with writing high-performance parallel streaming applications, such as the trade-off between productivity and performance, and the lack of high-level frameworks facilitating the creation of these applications, Griebler et al. [32] developed SPar, a C++ domain-specific language, to simplify the programming of stream processing applications for different parallel architectures. Initially, SPar focused on code generation for multi-core systems with the FastFlow runtime. However, as depicted in Figure 1.1, later works expanded SPar to other runtime systems such as Intel TBB [37] and OpenMP [38], and other computer architectures, such as GSParLib [60, 62], which allows code generation for GPUs and DSParLib [49, 56] which enables code generation for distributed architectures.

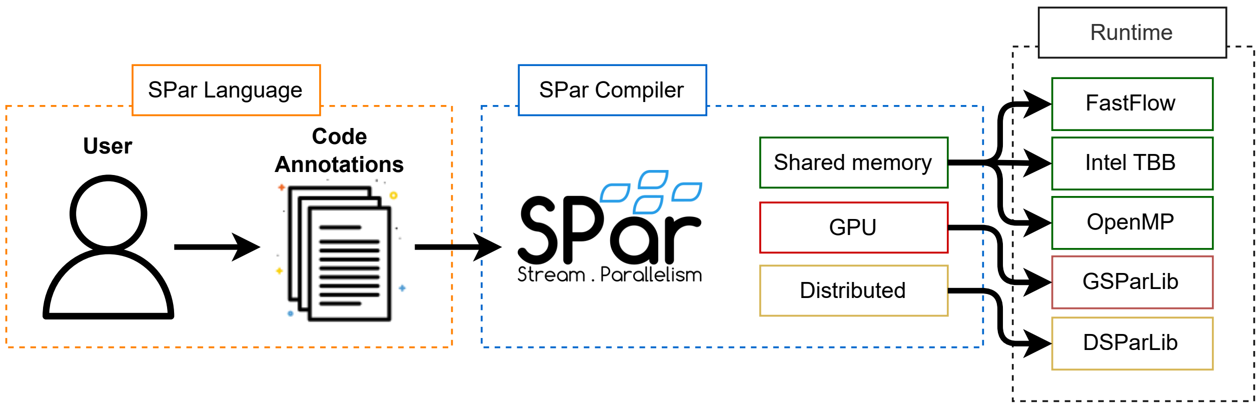


Figure 1.1: The SPar software ecosystem. Adapted from [39].

Other works of the research group, such as MPR [50], also approach the field of distributed stream processing. However, MPR focuses on exploring the dynamic process management features introduced in the second revision of the MPI standard (MPI-2) for dynamic scalability, not providing any fault tolerance mechanism or support for exactly-once semantics. Currently, SPar does not support MPR as a runtime for code generation. However, future works could introduce MPR into the SPar software ecosystem.

None of the currently supported SPar runtimes provide fault tolerance capabilities, meaning that any ongoing data and the entire computational progress are lost in case of failure. Andrade, Gedik, and Turaga [9] state that in many cases, sporadic data loss is acceptable in specific applications as long as the amount of data lost is limited and the final accuracy of the results is properly evaluated. At the same time, there are applications where data losses cannot be tolerated, as the data lost may contain critical information for the application's state, which must survive even catastrophic failures.

Alongside fault tolerance, a stream processing system must also provide message delivery guarantees. Akidau, Chernyak, and Lax [3] assert that "consistency guarantee" in stream processing systems can generally be grouped into three categories: at-most-once processing, at-least-once processing, and exactly-once processing. It is emphasized that these terms refer to the output generated by the application, not the number of times the application processes (or attempts to process) a particular record. Thus, some applications may trade a higher consistency guarantee for lower latency, while for others, it is essential to ensure that the pipeline has an exactly-once consistency guarantee. Motivated by these factors, the primary **goal** of this master's thesis is to investigate fault tolerance and exactly-once semantics for streaming systems so that it is possible to support it on the SPar software ecosystem without rewriting the user source code.

To achieve this goal, this work introduces ResiPipe (described in Chapter 4), a C++ library for distributed stream processing that provides fault tolerance and exactly-once semantics. ResiPipe is integrated into the SPar software ecosystem as an alternative to code generation for parallel and distributed architectures. For distributed execution, the library relies upon MPI (Message Passing Interface) for process communication and implements the Asynchronous Barrier Snapshotting (ABS) protocol [20] to create periodic snapshots of the application state. To achieve exactly-once semantics, ResiPipe requires re-playable data sources and idempotent sinks. A two-phase commit mechanism can be employed for actions over external systems through a callback feature that allows the execution of user-defined functions after certain events of the application life cycle.

1.3 Research Contributions

This work advances the field of parallel and distributed stream processing by achieving the following scientific contributions:

- A new C++ library that eases the development of fault-tolerant distributed stream processing applications on top of OpenMPI by abstracting the parallel pattern, processes communication, and data serialization.
- A new code generation algorithm for the SPar compiler targeting ResiPipe runtime that allows fault-tolerance in parallel and distributed stream processing applications.
- A literature review that summarizes the fault tolerance mechanisms, progress tracking mechanisms, and message delivery guarantees employed by the current state-of-the-art stream processing systems.
- A quantitative analysis composed of six performance experiments and two programmability evaluations that highlight the performance and programmability aspects of ResiPipe compared to other existing C++ distributed stream processing libraries, and four failure recovery experiments that evaluate the overhead introduced by the ResiPipe fault tolerance mechanism.

1.4 Outline and contents

The remaining chapters of this master's thesis are organized as follows:

- *Chapter 2 - Background*: Presents the background necessary to understand this work, including definitions regarding stream processing, fault tolerance, message delivery guarantees, and the SPar software ecosystem.
- *Chapter 3 - Related Work*: Presents the related work, including an overview of how the current state-of-the-art stream processing systems implement fault tolerance and message delivery guarantees.
- *Chapter 4 - The ResiPipe library*: Introduces the ResiPipe library, describing its implementation and usage, state management, progress tracking, fault tolerance, and load balancing.
- *Chapter 5 - ResiPipe Evaluation*: Presents the ResiPipe performance and failure recovery evaluation, including a comparative analysis regarding the performance of applications implemented using ResiPipe, DSParLib, MPR, and OpenMPI.
- *Chapter 6 - Resilient SPar code generation*: Describes the implementation of the SPar resilient code generation using ResiPipe as runtime and presents an evaluation of the overhead introduced by SPar-generated applications in comparison to regular ResiPipe applications.
- *Chapter 7 - Conclusion*: Presents the conclusion, final remarks, and future works.

2. BACKGROUND

In this chapter, we present the background seen as necessary to understand this Master's thesis. In section 2.1, we introduce the concept of stream processing, the differences between batch processing and stream processing, the concept of message delivery guarantee, and an introduction to fault tolerance. Section 2.2 covers the concepts related to the SPar language and the DSParLib, a library built to provide distributed capabilities for SPar code.

2.1 Stream Processing

According to Garofalakis, Rastogi, and Rajeev [26], traditionally data-management systems are built on the concept of persistent data sets, which are stored in reliable storage and queried/updated multiple times. In this way, the data collected during the day is sent for overnight processing, resulting in a significant delay between collecting data and obtaining results. However, for various applications like fraud detection in bank transactions or network intrusion detection, it is necessary to process data continuously (24/7) as it is collected. Figure 2.1 illustrates some examples of such applications.

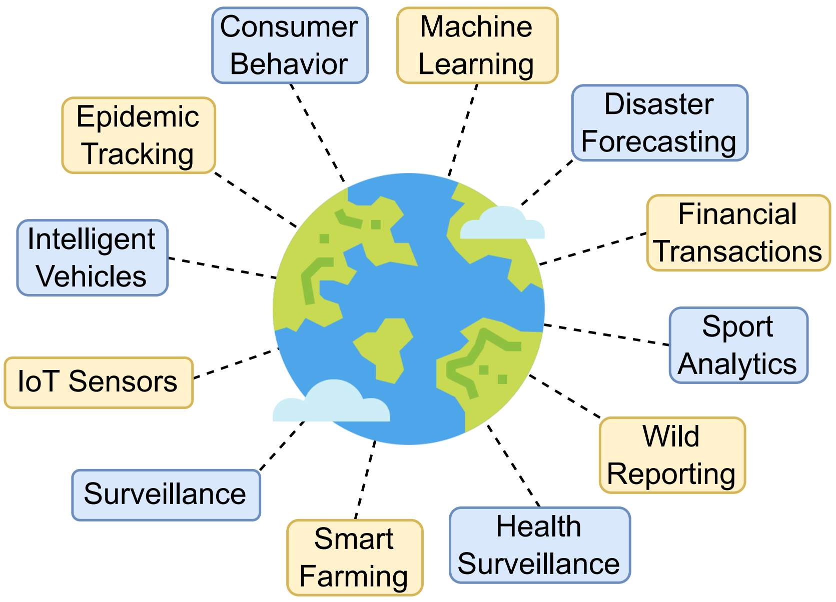


Figure 2.1: Stream processing applications. Adapted from [48].

According to Akidau, Chernyak, and Lax [3], "streaming" asserts to a data processing engine designed with an infinite number of datasets in mind, but has become a generic term for a wide range of applications requiring low latency or approximate/speculative results, leading to misunderstandings about the real meaning of this term. On the other hand, Andrade, Gedik, and Turaga [9] state that stream processing is a computing paradigm that addresses gathering, processing, and analysis of high-volume, heterogeneous continuous data stream, aiming to extract insights or valuable results.

Typically in stream processing systems, the applications can be represented by a series of operator processes that perform actions over the received data. According to Andrade, Gedik, and Turaga [9], an operator is the basic functional unit in a stream application, containing input and output data ports and being responsible for executing an arbitrary action over the received data, such as data conversion, aggregation, splitting, merging, or logical and mathematical operations, and sending the result to the next operator in the sequence. However, some operators do not have input or output data ports, as in the case of the "source" operator, responsible for receiving data from external sources, and the "sink" operator, responsible for sending data to external systems.

2.1.1 Stream and Batch Processing

Before the rise of the stream processing described in Section 2.1, the batch processing model was already widely used for handling high volumes of data. Figure 2.2 depicts the batch processing model according to Hueske and Kalavri [41], in which the data is initially grouped into batches before being processed. The batch size can vary from a fixed number of records to a value based on time, such as a minute, an hour, or a whole day of accumulated records. Due to this accumulation process, the batch processing model is traditionally associated with higher latencies, ranging from minutes to several hours, being directly related to the batch size and the arrival time of the last record.

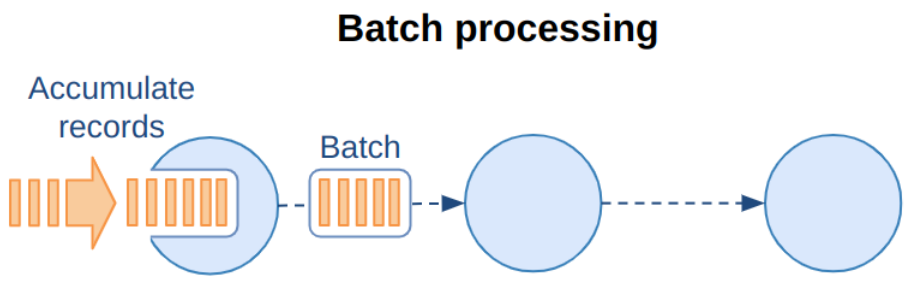


Figure 2.2: Batch processing model

In contrast, in the stream processing model depicted in Figure 2.3, the data is processed as soon as it arrives, without the artificial delay generated by grouping records into batches. However, according to Akidau, Chernyak, and Lax [3], stream processing systems are historically associated with applications that provide lower latencies at the expense of inaccurate or speculative results. This situation led to the emergence of the Lambda Architecture.

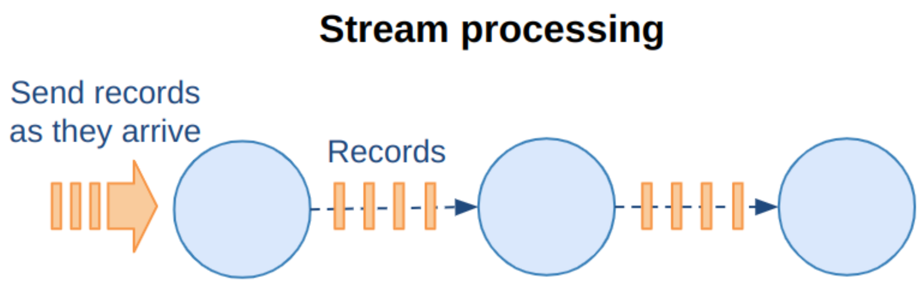


Figure 2.3: Stream processing model

According to the same author, the Lambda Architecture was created by Nathan Marz (the creator of Apache Storm) and involves the simultaneous execution of a stream and batch processing system, both performing the same operations. While the stream processing system provides low latency imprecise results (due to the use of approximation algorithms or because the stream processing system itself does not provide correctness), a batch processing system simultaneously runs overnight, eventually providing correct results. However, this architecture comes with a series of drawbacks, as it requires building, provisioning, and maintaining two independent versions of the same pipeline and somehow combining the results of the two applications.

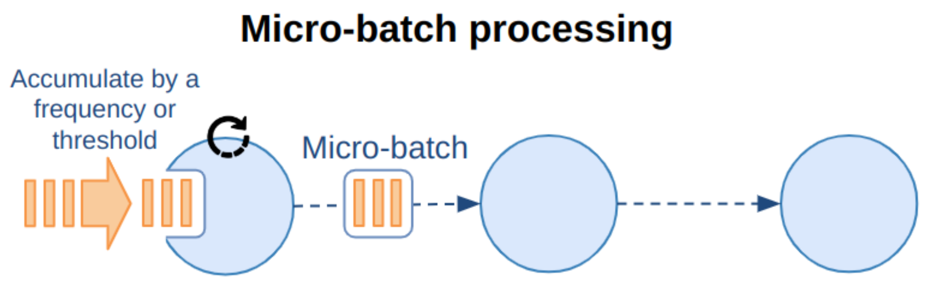


Figure 2.4: Micro-batch processing model

Given the drawbacks of the Lambda Architecture, new data processing systems, such as MillWheel [1] and Google Dataflow [2], were developed with the idea of unifying stream processing and batch processing under a single model with strong consistency guarantees. Meanwhile, other systems like Apache Spark adopted hybrid models such as the micro-batch model depicted in Figure 2.4, where conceptually the data is grouped into small batches of finite records before processing, aiming a balance between the lower latencies of the stream processing and the high throughput of the batch processing.

2.1.2 Fault-tolerance in Stream Processing Systems

Due to the need for stream processing systems to run for long periods of time, even indefinitely, reprocessing all data after a failure becomes highly costly or even unfeasible. Therefore, it is essential that a stream processing system not only recovers after a failure but also ensures that the operator's internal state is correct and the results generated by the application are accurate. Hueske and Kalavri [41] state that when receiving a record, an operator in a stream processing system can perform the following actions: (1) receive an event and store it in a local buffer, (2) make changes to the operator's internal state, or (3) produce an output result. However, a failure can occur during any of these actions, and the stream processing system must have explicit guidelines on how to behave in each of these failure scenarios.

According to Andrade, Gedik, and Turaga [9], a failure can occur due to an error in the user's application logic, a software bug, a failure in the system runtime, or a failure in the computational infrastructure, such as an unavailable machine, a network failure, or a storage failure. Therefore, fault tolerance mechanisms in stream systems generally focus on recovering from infrastructure-level failures, assuming that such failures are temporary and that a failed machine can be quickly replaced. In contrast, failures at the user application logic or at the system runtime are more challenging to fix, as they may require changes to the source code.

According to the same author, the strategies for recovering from failures in stream processing systems are generally based on three basic mechanisms: cold restart, checkpoint, and replication. In the cold restart mechanism, the affected segment is restarted from scratch when a failure is detected. This strategy is only feasible for applications with transient states or those able to reconstruct the state from new incoming data.

The checkpoint mechanism involves writing the application state over reliable storage. This checkpoint can include a complete or incremental copy of the application state and can be initiated either periodically or on-demand. In case of a system restart, the application state can be resumed from the latest checkpoint. However, the checkpoint mechanism alone does not guarantee that no data will be lost in the event of a failure, as new data continues to arrive continuously while the application is being restarted.

The replication mechanism involves maintaining copies of the same application running simultaneously, usually over multiple machines. In this mechanism, the replica that is actually generating results is called the active replica, while the others are called backup replicas. In this model, if the current active replica fails, a backup replica must be able to take over, maintaining data processing intact.

In addition, Elnozahy et al. [24] classify the rollback recovery protocols into checkpoint-based and log-based categories. Checkpoint-based protocols rely on creating checkpoints of the system state over reliable storage for further restoration. They can be sub-classified into coordinated checkpointing, where the processes coordinate their checkpoints in order to save a system-wide consistent state, and communication-induced checkpointing, where the processes are forced to make checkpoints based on information built-in on the messages received from other processes.

Log-based protocols rely on both checkpointing and event logging for fault tolerance and can be sub-classified into pessimistic logging, where the processes need to block their execution until the event is stored over reliable storage; optimistic logging, where the processes are not blocked and the events are persisted over stable storage asynchronously; and causal logging, which aims for a balance between the pessimistic and optimistic approaches.

2.1.3 Asynchronous Barrier Snapshotting (ABS)

Carbone et al. [20] states that in order to provide consistent results, a distributed stream processing system needs to be resilient to failures, and one of the ways of providing this resilience consists of periodically capturing snapshots, like pictures of the global state of the system, including all necessary information to restart the computation from that specific point after a failure.

The Asynchronous Barrier Snapshotting (ABS) algorithm introduced by the same authors consists of a lightweight snapshotting algorithm specifically designed for distributed stateful dataflow systems, aiming for a low impact on performance while also providing a low storage cost. The main idea behind the algorithm consists of slicing the data stream into stages by periodically injecting special snapshot markers (a.k.a barriers) into the stream without disrupting the system's regular execution.

Algorithm 2.1 presents a simplified representation of the Asynchronous Barrier Snapshotting mechanism for acyclic graphs. Over regular execution, a central coordinator periodically injects snapshot markers into all the source operators. When a source operator receives a snapshot marker, it takes a snapshot of its current state and then broadcasts it to all its output channels. When a non-source operator receives a snapshot marker from one of its input channels, it blocks that channel until it receives the same marker from all its remaining input channels. Then, the operator takes a snapshot of its current state, broadcasts the marker to its output channels, and unblocks its input channels. The global snapshot is complete once all the operators have taken a snapshot for that given marker.

The algorithm assumes that the network channels are reliable and can handle process crashes or messages losses, respect a FIFO delivery order, and can be blocked and unblocked. It also assumes that the operators can trigger operations such as blocking and unblocking channels, sending messages, and broadcasting messages to all its output channels, and also that messages injected into source operators (e.g. snapshot markers) can be resolved into a "Nil" input channel.

Algorithm 2.1 Asynchronous Barrier Snapshotting for Acyclic Graphs [20]

```
upon event ⟨Init | input_channels, output_channels, fun, init_state⟩ do
    state := init_state; blocked_inputs := ∅; inputs := input_channels; outputs := output_channels;

upon event ⟨receive | input, ⟨barrier⟩⟩ do
    if input ≠ Nil then
        blocked_inputs := blocked_inputs ∪ {input};
        trigger ⟨block | input⟩
    if blocked_inputs = inputs then
        blocked_inputs := ∅;
        broadcast ⟨send | outputs, ⟨barrier⟩⟩
        trigger ⟨snapshot | state⟩
        for each input ∈ inputs
            trigger ⟨unblock | input⟩

upon event ⟨receive | input, msg⟩ do
    state′, out_records = fun(msg, state);
    state := state′;
    for each {output, out_record} ∈ out_records
        trigger ⟨send | output, out_record⟩
```

Carbone et al. [20] also explain that several failure recovery schemes work with this kind of consistent snapshots. However, one of the simplest forms is to merely restart the whole system execution from the last global snapshot complete. In such a way that at startup, every operator retrieves its associated snapshot from the persistent storage, sets its initial state, and starts to ingest records from its input channels. The Asynchronous Barrier Snapshotting mechanism does not provide exactly-once semantics by itself. It only guarantees that in case of failure, the system can restart its execution from a previous consistent state. In order to achieve exactly-once semantics, the system should be able to replay records and ignore possible duplications.

2.1.4 Message Delivery Guarantees

Unlike in a batch processing system, where failures can be recovered simply by reprocessing the failed batch from scratch, in a stream processing system, the failure recovery process brings a series of challenges typically categorized into consistency guarantees. According to Akidau, Chernyak, and Lax [3], consistency guarantees (or message delivery guarantees) can generally be grouped into three main categories: at-most-once processing, at-least-once processing, and exactly-once processing. It's important to note that these semantics refer to the number of times a record is observed in the final result, not the number of times the pipeline processes (or attempts to process) a record.

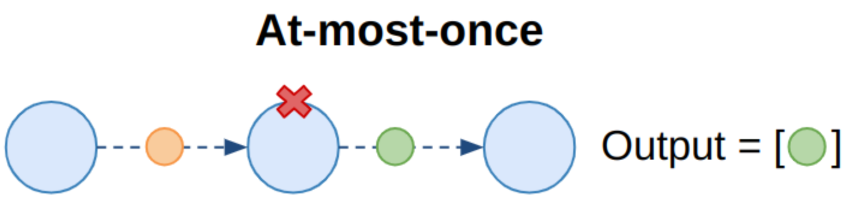


Figure 2.5: At-most-once delivery guarantee

For Hueske and Kalavri [41], the at-most-once delivery guarantee, depicted in Figure 2.5, is also known as "no guarantee". In this scenario, when a failure occurs, no action is taken to recover the application state or the records lost during the failure. This guarantee level is the most straightforward, and as the name suggests, it aims to process each record at most once. Therefore, the records are lost in a failure, and no effort is made to ensure correct results. On the other hand, by not introducing complex failure recovery mechanisms, this guarantee provides the lowest possible latencies.

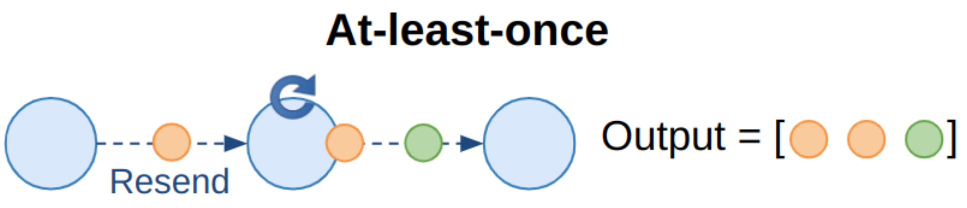


Figure 2.6: At-least-once delivery guarantee

The at-least-once delivery guarantee depicted in Figure 2.6, ensures that all data is processed at least once, minimizing the risk of data loss during a failure. However, some data may be processed more than once, resulting in duplicate outcomes. This guarantee level is appropriate for applications where the final result does not depend on the number of times the records are processed but rather on their successful processing. For instance, if an application is designed to determine whether an event has occurred or not, regardless of how many times it has occurred, this level of guarantee is acceptable.

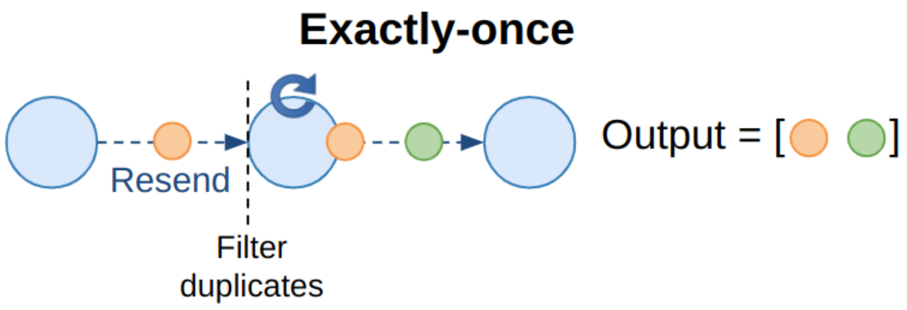


Figure 2.7: Exactly-once delivery guarantee

Finally, Figure 2.7 presents the exactly-once delivery guarantee, which ensures that the processed records will appear just one time at the final output. This level of conformity ensures that no data is lost in the event of a failure and guarantees the absence of duplicate data in the final result. This is the most challenging guarantee level to achieve and essentially means that the application will always produce consistent results, regardless of whether a failure occurs during processing.

2.1.5 The output commit problem

The output commit problem introduced by Strom and Yemini [64] and further described by Elnozahy et al. [24] and Fragkoulis et al. [25], dictates that a system cannot rely on the outside world for roll black recovery. The problem specifies that since a result cannot be retracted from the outside world (external system) once it is sent, the results should not be visible to the outside world until it is guaranteed that, in case of failure, the system can recover itself from a position after the result have been published. For example, a printer cannot undo the effects of printing a character over paper, and an automatic teller machine cannot recover the money after that is dispensed to the customer.

This issue is particularly pertinent to stream processing systems, as most rollback recovery mechanisms attempt to reprocess records after a failure, potentially resulting in duplicated records being sent to external systems. By definition, stream processing systems that solve the output commit problem provide exactly-once output. Fragkoulis et al. [25] classify the strategies to solve the output commit problem into three categories: transaction-based, progress-based, and lineage-based.

Transaction-based strategies such as those employed by Millwheel [1] and Trident [12] rely on the assignment of unique identifiers that can be used to filter out duplicated entries. Millwheel assigns a unique identifier to each record on the data stream and always persists the output over reliable storage before sending it to downstream operators. When an operator receives input records, it acknowledges the received records and discards the records whose identifiers have already been seen by the operator. Trident adopts a different strategy, where the records on the data stream are grouped into transactions with unique identifiers. Trident uses this transaction identifier to process the batches in order and discard the batches whose transaction identifier was already processed.

Progress-based strategies such as those employed by Naid [54] rely on a timestamp comparison to deliver exactly-once output based on the order of the timestamps. On the other hand, Lineage-based strategies such as those employed by TimeStream [58] and Streamscope [45] track the operator input and output dependencies by maintaining a history of which records compose any given result.

2.2 SPar

The SPar, an acronym for Stream Parallelism, is a domain-specific language for C++. Griebler et al. [31, 30, 32] created SPar to address the challenges associated with developing high-performance stream parallel applications, as well as the trade-off between code productivity and performance for different parallel architectures. SPar simplifies the development of stream parallel code without significant refactoring of the serial code by introducing a set of special annotations. These annotations define the basic components of a stream application, such as the data source and processing stages. Recent studies have demonstrated the programmability benefits with beginners [6] and using coding metrics [5, 7].

The code annotations introduced by SPar consist of standard C++11 attributes, which are compiled (source-to-source) into intermediate code that makes calls to a high-performance library. Initially, Griebler's work [32] focused on code generation for multi-core architectures using the FastFlow runtime. After, other works expanded SPar language to support service level objectives [34], self-adaptive techniques [69], other runtime systems for multi-core architectures such as Intel TBB [37] and OpenMP [38], the integration of data parallelism in stream processing [48, 47], and other computer architectures, such as GSParLib [61, 62], which allows code generation for GPUs using CUDA and OpenCL. For parallel distributed architectures, preliminary studies were carried out in [33] and later on with DSParLib [56, 49] using MPI.

```
[[spar::ToStream]] while(1){
  std::string stream_element;
  read_in(stream_element);
  if(stream_in.eof()) break;
  [[spar::Stage,spar::Input(stream_element),spar::Output(
    stream_element)]]
  { compute(stream_element); }
  [[spar::Stage,spar::Input(stream_element)]]
  { write_out(stream_element); }
}
```

Figure 2.8: Simple stream application using SPar. Extracted from [32].

Figure 2.8 illustrates an example of code using SPar annotations. The code represents a simple application where a loop continuously reads data from a stream and executes a function called "compute". After this stage, the resulting data is passed to the "write_out" function. If the received data represents the end of the stream, the loop is interrupted, concluding the application. This example uses the SPar annotations named ToStream and Stage. The ToStream annotation represents a region in the code where the stream processing occurs, while the Stage annotation marks different stages of the application, receiving attributes such as Input (representing the input data of the stage) and Output (representing the output data of the stage). The Stage annotation can also receive an attribute called Replicate, representing the number of parallel processes used in processing.

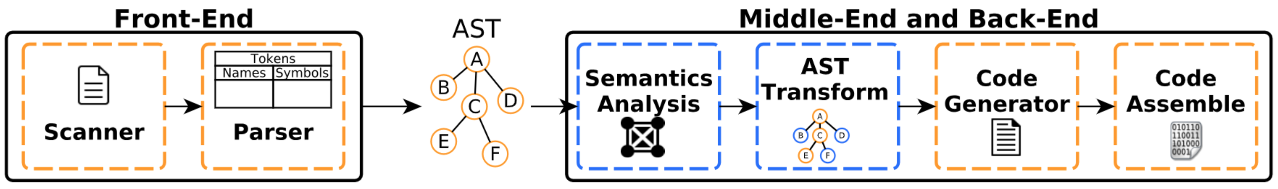


Figure 2.9: SPar compiler pipeline. Extracted from [32].

SPar has its own compiler based on CINCLE (Compiler Infrastructure for New C/C++ Language Extensions) to handle these special annotations. Figure 2.9 illustrates the compilation of a program using the SPar compiler. The compilation process starts with a call to the GNU C++ compiler, just before invoking the scanner that performs the syntactic and semantic analysis of the C++ code. The scanner produces tokens from the source code, which are used to create an Abstract Syntax Tree (AST), which serves as input for both the middle-end and back-end of the compiler. Finally, the compiler generates parallel code based on the AST and then calls the GCC compiler to produce the binary.

2.2.1 DSParLib

As mentioned in Section 2.2, the DSParLib (an acronym for Distributed Stream Parallelism Library) was developed in Pieper's master's thesis [56, 49], aiming to provide SPar code generation for distributed architectures. One of the primary goals of DSParLib is ease of use, and to achieve this, DSParLib is inspired by the structured parallel programming paradigm. The library uses the concept of building blocks to encapsulate user code. These blocks can be connected sequentially, forming the application's flow. Additionally, DSParLib performs compile-time code checks to ensure that the connected blocks are valid, avoiding potential runtime errors.

To achieve distributed stream processing, DSParLib utilizes OpenMPI, an open-source implementation of the Message Passing Interface (MPI), a widely used library for developing distributed applications. The building blocks provided by DSParLib abstract the communication between processes and consist of three basic components illustrated in Figure 2.10: the sequential wrapper (white part of the block), which wraps the user's sequential code, and the Input and Output Serializers (yellow and blue parts of the block), which handle sending, receiving, and data serialization.

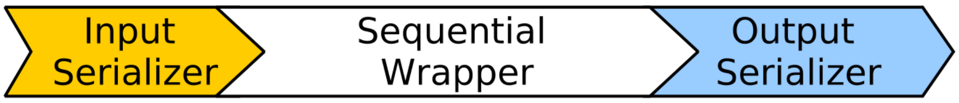


Figure 2.10: DSParLib block composition. Extracted from [56].

DSParLib provides two parallel processing patterns for distributed stream processing: Pipeline and Farm. In this context, the Pipeline model can be implemented through a connected sequence of sequential stages. Unfortunately, DSParLib does not support the replication of sequential stages for parallel execution. On the other hand, the Farm model can be implemented by connecting three blocks: the emitter, the worker, and the collector. The worker block can be replicated in this model, allowing better performance in high computational demand situations.

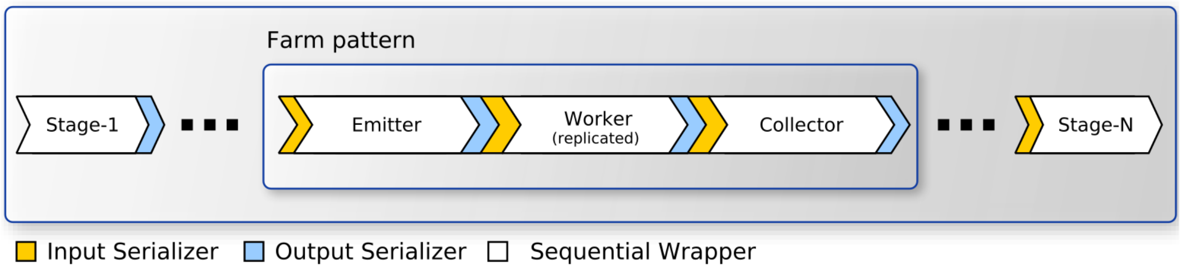


Figure 2.11: DSParLib pattern composition. Extracted from [56].

As depicted in Figure 2.11, DSParLib also allows the creation of semi-arbitrary compositions of the parallel patterns, enabling the grouping of multiple Farm stages, which provides stage replication within Pipeline stages that can only be executed sequentially.

2.2.2 MPR

The work of Löff et al. [46] introduces MPR (Message Passing Runtime) as a framework built on top of MPI, designed to simplify the development of self-adaptive distributed stream processing applications in C++. The MPR framework allows the applications to reconfigure the number of parallel processes during execution time (horizontal scaling) in response to workload variations without disrupting the application's regular execution.

The MPR framework relies upon the MPI dynamic process management feature introduced into the MPI-2 specification to create and remove processes over execution time. Figure 2.12 presents an overview of the MPR architecture, which is divided into three layers. The first layer consists of the MPR high-level API, which includes all the functions, structs, and classes available to the developer. The second layer is the Processing Engine, which includes configuration files and processes responsible for the user application execution and dynamic scalability. Finally, the last layer involves communication aspects, such as MPI calls.

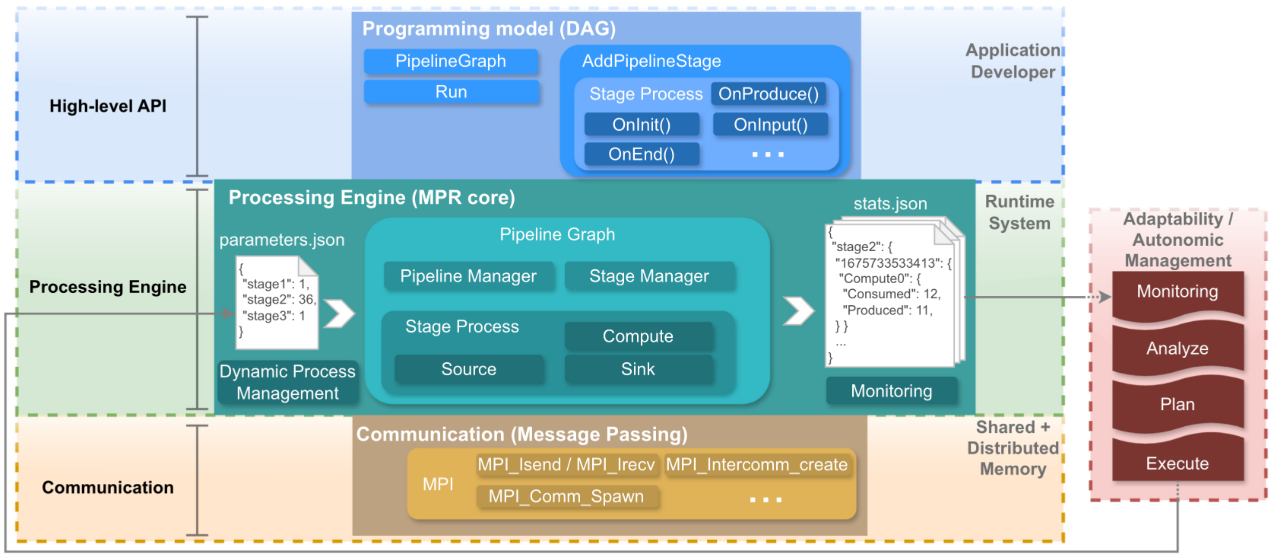


Figure 2.12: MPR System Architecture. Extracted from [46].

The MPR components are into three categories: Pipeline Manager (a process responsible for decisions such as choosing processes that must be removed from the current execution, broadcasting critical events, and maintaining a consistent state of the pipeline), Stage Managers (intermediary processes that mediate the communication between the Pipeline Manager and the remaining processes), and Stage processes (The user application itself, such as data sources, computational stages, and sink operators).

As a limitation, the current implementation of MPR does not provide a fault tolerance mechanism or strong message delivery guarantees, assuming that the underlying environment will be equipped with a reliable high-speed and low-latency network. The MPR also supports only 3-stage pipelines, considerably reducing the gamut of supported applications.

2.2.3 MPI - Message Passing Interface

The Message Passing Interface Forum [52] defines MPI, an acronym for Message-Passing Interface, as a specification for implementing library interfaces that address the message-passing parallel programming model, in which data is moved from the address space of one process to another using cooperative operations. The MPI specification defines a base set of operations for process communication expressed as functions, subroutines, or methods according to the appropriate language bindings for C and Fortran.

The MPI movement began in 1992, with formal standardization efforts started in early 1993 as an attempt to create a standard among multiple mutually incompatible and yet functionally equivalent message-passing interfaces emerging in the HPC (High-Performance Computing) field due to the popularization of the so-called massive parallel computing [65]. The creation of the MPI standard involved approximately 40 organizations from the United States and Europe, including major vendors of concurrent computers, researchers from universities, government laboratories, and industry. [42]

One of the main reasons for the wide adoption of MPI has been the ability to deliver portable applications with acceptable performance and scalability for multiple platforms such as distributed-memory massively parallel processing (MPP) platforms, symmetric multiprocessing (SMP) machines with shared memory, and hybrid systems with coupled SMP nodes [36].

Since the announcement of the first release of the MPI specification on May 5, 1994, the MPI Forum has released several revisions of the standard, including new features such as dynamic process management, one-sided communication, and non-blocking communication. Currently, the latest revision of the MPI standard is MPI-4.1, released on November 2, 2023, which includes mostly corrections and clarifications to the previous MPI-4.0 release [52].

As a specification, MPI is not a proper implementation or language. Multiple implementations of the MPI specification, such as OpenMPI[1], MPICH[2], and MS-MPI[3] can be found in the literature, each one implementing its own set of optimizations and support to high-performance interconnect technologies, such as Infiniband and Intel Omni-Path. The main advantage of establishing a message-passing standard is to improve the ease of use and portability across distributed environments, in which high-level routines and abstractions are built on top of lower-level message-passing routines. Creating a standard provides vendors with a clearly defined base set of routines that can be efficiently implemented or, in some cases, provided with hardware support, enhancing the scalability and efficiency of the systems.

2.2.4 LLVM and Clang

According to Murashko, Ivan [53], the LLVM project was originally designed to be a next-generation compiler infrastructure for building optimized compilers for many programming languages. The LLVM project was started in 2000 by Chris Lattern and Vikram Adave as a project at the University of Illinois at Urbana-Champaign. Since the project started, it has evolved into a full-featured platform for building various tools, such as

[1]https://www.open-mpi.org/
[2]https://www.mpich.org/
[3]https://learn.microsoft.com/en-us/message-passing-interface/microsoft-mpi

debuggers, profilers, and static analysis tools. It also has been widely adopted by the software industry and academic research community.

Murashko, Ivan [53] states that a typical programming compiler workflow is generally divided into three stages: front-end, middle-end, and back-end. The front-end is responsible for parsing the source code and performing the lexical analysis, generally including a syntax analysis step that verifies if the source code is well-organized and follows the grammar rules of the programming language and a semantic analysis step that validates if the source code is meaningful and rejects invalid programs with wrong types or definitions. The middle-end performs optimizations over an intermediate representation of the source code. The back-end transforms the optimized intermediate representation of the source code into machine code or assembly code.

Initially, the LLVM project used the GCC (GNU Compile Collection) as the default C/C++ compiler front-end. However, GCC is licensed over GLP (General Public License), which prevents its usage as a front-end compiler in some proprietary projects. It also had limited support for Objective-C at the time, which was a significant drawback since Apple had made the LLVM an integral part of its development tools (Xcode development environment). To address this issues, Chris Lattner started the Clang project in 2006.

Hsu, Min-Yih [40] defines Clang as the LLVM's official front-end for C-family programming languages such as C, C++, and Objective-C, that is responsible for performing the parsing, type checking, and semantic reasoning (among others) steps over the source code. The Clang front-end parses the input source code into an Abstract Syntax Tree (AST) that can be manipulated and modified as required, allowing complex code generation. As a result, Clang generates an equivalent intermediate representation (LLVM IR), which is processed and optimized by the following stages of the compiler infrastructure, eventually becoming machine code.

2.2.5 Halstead's Metrics

Bundschuh, M. and Dekkers, C. [18] define Halstead's Metrics as a set of measures introduced in 1997 by Maurice H. Halstead [35] to evaluate software source code complexity. It evaluates the complexity of a program based on the total number of operators, such as comparisons, arithmetic operators, alternatives, loops, reads, and writes (N_1), the total number of operands, such as variables, constants, marks, records, and unions (N_2). The total number of distinct operators (n_1), and the total number of distinct operands (n_1). As depicted in Figure 2.1, the metrics estimate the program vocabulary, length, volume (in bits), programming difficulty, and programming effort.

Table 2.1: Halstead’s Metrics (extracted from [43])

Symbol	**Value**	**Metric**
n	$n_1 + n_2$	Program vocabulary
N	$N_1 + N_2$	Program length
V	$N \times log_2\ n$	Volume
D	$(n_1 / 2) \times (N_2 / n_2)$	Difficulty
E	$D \times V$	Effort

Legaux et al. [43] state that Halstead’s Metrics are not tied to a specific programming language since they capture the complexity of writing based on the number of operands and operators. However, there is no standard definition of what exactly is considered an operator or an operator. The author states that a reasonable definition for the C++ language could be presented as follows: operands are composed of type names, constants, and user-defined identifiers. While operators are composed of storage class specifiers (static, virtual, inline, ...), type qualifiers (const, friend, ...), reserved instructions (for, if, struct, namespace, typenames, ...), all the arithmetic and logical operators (+, ==, &&, ...), the ";" delimiter and the parenthesis pairs.

According to Andrade [8] based on these metrics, it is possible to stipulate the programming time (T) required to convert an algorithm to a specific programming language by using the equation $T = E/S$, in which S stands for the rate in seconds the brain makes elementary mental discriminations. Gordon et al. [27] states that the speed the brain takes to make elementary discriminations in software science can be obtained from psychology as $5 \leq S \leq 20$ discriminations per second. However, this time may vary according to the programmer’s concentration level and fluency in the given language.

3. RELATED WORK

This chapter presents the related work exploring the literature centering scientific documents within the area of streaming processing systems. Section 3.1 presents a comprehensive literature analysis of the fault tolerance and message delivery guarantee mechanisms employed by different systems within this domain. Section 3.3 presents an overview of existing high-level parallel programming interfaces for stream processing.

3.1 Related Research for Fault Tolerance in Stream Processing Systems

In this section, we present a detailed review of the fault tolerance mechanisms and message delivery guarantees present in the related research. The addressed systems include Apache Flink, Apache Spark Streaming, Apache Spark Structured Streaming, Apache Storm, ChronoStream, Google DataFlow, MillWheel and Naiad. Table 3.1 presents the search string used in this research. This search string has been executed over the Scopus database approaching the terms "stream processing", "stream data processing", "stream computation", "streaming API", "data processing system", and "dataflow." The documents are limited to conference papers and articles related to the field of computer science published between 2011 and 2024.

Table 3.1: Search string composition

Search String
TITLE-ABS-KEY ("stream processing" OR "stream data processing" OR "stream computation" OR "streaming API" OR "data processing system" OR "dataflow") AND PUBYEAR >2011 AND PUBYEAR <2024 AND (LIMIT-TO (DOCTYPE , "ar") OR LIMIT-TO (DOCTYPE, "cp")) AND (LIMIT-TO (SUBJAREA, "COMP"))

The Scopus database was chosen because it indexes many relevant sources in the researched area and is one of the primary databases used by the research group. The results of the search have been sorted in descending order based on the number of citations. To include or exclude documents from the search, a set of inclusion and exclusion criteria has been created and summarized in Table 3.2. In addition, the Google Dataflow paper did not appear in the results of the search string and was manually included in the list due to its relevance.

Table 3.2: Inclusion and exclusion criteria

Inclusion criteria
1. The publication must provide complete access to the material;
2. The document must be published in journals or conferences;
3. The publication must be related to the topic of stream processing systems.
Exclusion criteria
1. Publications written in languages other than English or Portuguese;
2. Publications in editorials, prefaces, abstracts, interviews, news, and reviews;
3. Incomplete or inconsistent publications;
4. Duplicated publications in different databases.

3.1.1 Apache Flink

The work of Carbone et al. [19] presents the state management in Apache Flink [13], which consists of a framework and distributed stream processing engine that enables stateful computations on data streams. Apache Flink has incorporated fault tolerance through a distributed snapshot mechanism similar to the classical Chandy-Lamport protocol [21], which allows the system to revert to a previous consistent snapshot but does not provide exactly-once guarantees on its own. In version 1.4.0, the two-phase commit feature described by Carbone et al. [19] was introduced, enabling end-to-end exactly-once applications for data sources and sinks that support transactions.

Carbone et al. [19] state that during Apache Flink regular execution, snapshot markers are injected on the data stream to divide the records into logical slices called *epochs*. An alignment phase is executed for operators with multiple input sources to synchronize all the input sources into the same epoch before proceeding with a snapshot. During this alignment phase, when an operator receives a new snapshot marker, the operator blocks the input channel that received the marker and waits for the same marker to arrive from all the input channels. Once the snapshot marker is received from all the input channels, the operator makes a snapshot of its current state, broadcasts the received marker to all its output channels, and unblocks the input channels. The distributed snapshot is complete once all the operators have performed a snapshot for the given epoch.

On recovery, the system restores the most recent complete snapshot and reprocesses the records from the incomplete epoch that was active during the failure. Regarding state management, Apache Flink classifies states into two types: local state and external state. The local state includes all the in-memory states or within an embedded key-value database like RocksDB. The external state encompasses all states stored over external databases. For local states, when a snapshot is started, a copy of the current state is written to reliable storage, such as a distributed filesystem. For external states, the approach varies based on the capabilities of the external database.

For databases with MVCC (Multi-Version Concurrency Control) support, the state changes associated with each snapshot can be stored across multiple database versions. Once the global snapshot is complete, the database version is incremented, and if the snapshot fails, it is decremented. For Non-MVCC databases, each snapshot follows a two-phase commit protocol: when an operator makes a snapshot, the state changes are logged and pre-committed. Once the global snapshot is complete, all pre-committed changes are fully committed by the JobManager in an atomic transaction.

3.1.2 Apache Spark Streaming

Spark Streaming [10] [72] is an extension of the Apache Spark ecosystem, developed for high-throughput and fault-tolerant data processing. On Apache Spark Streaming, the continuous data stream is represented by an abstraction called "Discretized Stream" or DStream. The DStream aggregates the records in batches (e.g., the 1-second interval of records) represented as an RDD (Resilient Distributed Dataset).

Zaharia et al. [71] introduce the Resilient Distributed Datasets (RDDs) as a distributed memory abstraction that allows fault-tolerant in-memory computations over large clusters. RDDs are read-only, partitioned data collections exclusively formed through deterministic operations on data stored in stable storage or other RDDs. One of the main characteristics of RDD is that it keeps information about how it was created (its lineage) in a way where a program cannot reference an RDD that it cannot reconstruct after a failure.

During the normal execution of Spark Streaming, the system will start running the "driver-program", which contains the "main" method of the user application. The driver will request the cluster manager (e.g., Mesos or YARN) for resources to launch the "executors" responsible for running the tasks associated with the application. Once the executors are initiated, they connect with the driver to receive work. The driver determines the number of tasks that need to be created and generates a logical and physical execution plan. Upon completion, the executors send back the results to the driver. When all the tasks are completed, Spark requests the cluster manager to release all the resources associated with the application.

Spark Streaming provides two kinds of data source operators to achieve exactly-once delivery guarantees. The first one is called a "reliable" data source and requires that the external system supports a confirmation that Spark has received the data, and also supports rewinding and resending data in case of a failure. The second kind of data source is called "unreliable". This data source is simple to implement but doesn’t provide fault-tolerant guarantees. Spark Streaming also offers guarantees for sink operators through two methods: "Idempotent updates", which ensure that multiple attempts to write data always result in the same outcome, and "Transactional updates", which execute all updates in atomic transactions, providing exactly-once guarantees.

To prevent data loss in the event of a node failure, Spark 1.2 implemented the write-ahead log feature. This ensures all received data is written in a persistent storage before processing. Additionally, for cases where it is too costly to recreate RDDs from their lineage, Spark supports periodic progress checkpoints. In the occurrence of a failure on the driver node, the entire topology stops and needs to be restarted. Upon initialization, the driver program utilizes the latest completed checkpoint to reconstruct the context and restart the executor nodes.

3.1.3 Apache Spark Structured Streaming

Apache Spark Structured Streaming [11][14] consists of a fault-tolerant stream processing engine that, similar to Apache Spark, allows the user to express the application logic using high-level abstractions. The main difference between these two streaming engines is that Structured Streaming is built on top of the Spark SQL engine and allows batch and continuous stream processing.

Structured Streaming utilizes two abstractions to represent a continuous data stream: the Dataset and DataFrame. The Dataset is a distributed collection of data that offers the same advantages as RDDs but can operate within the Spark SQL engine. The DataFrame abstraction, on the other hand, is a Dataset that incorporates named columns, similar in concept to a table in a relational database. Also, Structured Streaming offers two distinct operating modes: micro-batching and continuous processing (introduced in Spark 2.3). By default, micro-batching is employed, which aggregates records into small data batches, similar to Spark Streaming's discretized streams execution model. This approach delivers latencies up to 100 milliseconds, and offers exactly-once delivery guarantees. Alternatively, continuous processing mode reaches even lower latencies, up to one millisecond, but only provides at-least-once delivery guarantees.

To provide exactly-once semantics, Structured Streaming assumes that every data source is replayable and uses start and end offset positions to track progress. The engine saves these offsets on checkpoint and write-ahead log. Structured Streaming also provides sink operators designed to be idempotent for handling reprocessing.

On Spark Structured Streaming, if a failure occurs, the entire execution pipeline is stopped, and upon restart, the system uses the latest completed checkpoint and the information on the write-ahead logs to continue the processing from where it has stopped.

3.1.4 Apache Storm

Trident [12] can be described as an effort to provide exactly-once delivery guarantees for Apache Storm [68]. For archive exactly-once guarantees, Storm follows two primitives. The first is that each batch of data has a unique identifier called the "transaction_id". The second one is that state updates are ordered among batches, so batch "3" updates won't be applied before batch "2" updates have succeeded.

With these primitives, Storm stores both the value and the unique identifier in an atomic transaction over an external database. If a failure happens and some batch is replayed, Storm verifies if the identifier already exists on the database; if it exists, the batch can be ignored. This verification can be turned off if the user wants to avoid paying the cost of storing the unique identifier and verifying his existence on the database; in this scenario, the user still has at-least-once delivery guarantee.

Storm provides a checkpoint mechanism for fault-tolerance where a message flows through a separate channel across the topology. In this checkpoint mechanism, a new transaction is started at a given time interval, and a special checkpoint source emits a checkpoint message. When a stateful operator receives a checkpoint message, it saves the current state and prepares the transaction, then notifies the checkpoint source that the message has been received and forwards it to the next operator. The checkpoint is completed, and the transaction is committed once the checkpoint source receives acknowledgments from all the operators.

The recovery phase is initiated when the topology is launched for the first time or in case of any detected failure. During this phase, if the previous transaction has not been prepared, the checkpoint source will send a rollback message, causing the operators to abort their current transaction and the data source to resend the data. However, if the previous transaction was successfully prepared but not yet committed, the checkpoint source will send a commit message, enabling any prepared transaction to be committed.

3.1.5 ChronoStream: elastic stateful stream computation in the cloud

The work of Y. Wu and K. -L Tan [70] describe ChronoStream as a system designed to run distributed stateful stream computation in the cloud, with dynamic scaling and failure recovery. In ChronoStream, each operator is encapsulated in a container that periodically reports its current status, such as heartbeat and computation progress, to a job manager. The job manager issues instructions to the containers when dynamic scaling or failure recovery is required. Each container periodically makes a checkpoint of its active slices of records to remote peer containers, and the job manager also records any update on the container configuration.

To track the computation’s progress, ChronoStream labels each record in the stream with a unique identifier and maintains a vector containing the number of consumed records for each input stream. When a checkpoint is triggered, the progress vector and a snapshot of the records are recorded.

For fault tolerance, ChronoStream implements Chained Backups and Asynchronous delta checkpointing. On the Chained Backup, the system periodically makes a checkpoint of the active slices of records to remote nodes for supporting elasticity and high availability; under this checkpoint, each slice of records is saved to its peer containers with a locality-sensitive data placement scheme in a way where the backups are placed in an interconnected chain.

The Asynchronous Delta Checkpoint divides the computation’s lifespan into three stages: normal, checkpointing, and merging. During the normal phase, all state updates are logged on an in-memory key-value store, with the updated entries being marked with a "dirty" bit. During the checkpoint phase, the system scans the key-value store and saves the updated entries to remote storage. This phase is nonblocking; any incoming updates are buffered to a temporary data structure. Finally, all buffered updates are integrated into the key-value store in the merging phase. And once a failure is detected, the job manager requests neighbor nodes for slice reconstruction. The neighbor nodes use the data from the Chained Backups to reconstruct the slice of records of the failed node, and the normal execution flow continues.

3.1.6 Google Dataflow

The work of Akidau et al. [2] introduces Google Dataflow as a data-processing service designed on top of years of experience from Google with FlumeJava and MillWhell. Among the primary motivations for developing Google Dataflow was the internal need for a model that unifies batch and streaming processing models. Other motivations came from previous experiences with the Lambda Architecture, where the consumers ran their streaming pipelines in a weak consistency mode, with a nightly MapReduce to verify the reliability of the results, resulting in consumers losing trust over time in the results generated by the streaming pipelines with weak consistency, and reimplementing their systems with strong consistency solutions.

Google Dataflow works in a software-as-a-service model, where all resources needed to run the service are provided by the Google Cloud Platform (GPC). Once a Dataflow task is initiated, the Dataflow service assigns a set of worker virtual machines to carry out the tasks, dynamically scaling the number of virtual machines up or down as required and disposing of them once the job is finished or terminated. The user is only charged for the resources used during the execution of the tasks.

Dataflow reaches exactly once delivery guarantees using a strategy that involves shuffling the data between the workers using remote procedure calls (RPCs). Each message is tagged with a unique identifier, and the workers try to resend it until they receive a confirmation that it has been received at the destination. Dataflow workers use a Bloom filter over the unique message identifier to overcome duplicated messages. Upon receiving a new message, the user code is executed over the input records, which may generate state changes or output records. These outcomes are then stored in a fault-tolerant storage, in a stage similar to a checkpoint, before being sent to downstream operators.

Dataflow also requires that data sources support rewind and replay records if needed to reach exactly-once delivery guarantee. For deterministic sources (such as Apache Kafka or Google Pub/Sub), Dataflow uses start and end offset positions to track progress. For non-deterministic sources, Dataflow requires that the data source informs a unique record identifier to avoid duplication. Also, to ensure that the records are delivered exactly-once, Dataflow provides built-in idempotent sink operators based on the Apache Beam SDK, designed not to produce duplication.

In a failure, the Dataflow service automatically restarts the failed worker and retries the execution of the user code. This behavior can generate unwanted results if the user code interacts with external systems (e.g., Writing data on an external database). In this case, the user must guarantee that his code can be executed multiple times without generating unwanted side effects.

3.1.7 MillWheel: Fault-Tolerant Stream Processing at Internet Scale

Tyler Akidau et al. [1] describe the programming model, implementation, and fault-tolerance guarantees of MillWheel, a low-latency data-processing framework for data-stream applications widely used at Google. There are two modes in which MillWheel can operate. The first mode, "Weak Productions", comes with lower resource and latency costs but does not guarantee exactly-once delivery. The second mode "Strong Productions" offers exactly-once delivery guarantee but comes with higher resource and latency costs.

On the normal execution of MilWheel in "Strong Production" mode, every time an operator receives a record from an input stream, the record is checked against duplication using a Bloom filter, and the user code is executed for the input record, possibly resulting in state changes or result records that are committed to the backing store (e.g., BigTable), in a process similar to a per record checkpoint. The computation result is delivered to the output stream; the input stream is notified that the record has been processed successfully, and the system is informed that older checkpoints can be garbage collected.

By default on MillWheel, as long the application uses the state and communication abstractions, any failures and retries are hidden from the user. The stream abstraction retries to send the records until they receive an acknowledgment that they have been received, obtaining the at-least-once requirement, a pre-requisite for the exactly-once.

In the occurrence of a failure, only the failed operator is restarted, and the input stream abstraction will replay the records from the latest completed checkpoint. The Bloom filter will discard any duplicated record the operator has already processed. When the sink operator receives a record, it also performs a duplication check, executes a prepared action over an external system (e.g., write the output on a database), and notifies the input stream that the record has been processed so the checkpoint of this record can be garbage collected.

3.1.8 Naiad: A Timely Dataflow System

Murray et al. [54] describe Naiad as a high throughput distributed system for data processing and execution of cyclic dataflow programs that has the ability to run both interactive and incremental computations. Naiad is implemented as a library for the C# programming language and works upon a specific computational model named "timely dataflow". In the timely dataflow model, the records are marked with a logical timestamp representing progress points in the computation, which are used to keep an asynchronous coordination mechanism.

In the distributed scenario, Naiad uses the concept of local frontiers to keep a local approximation of progress. In this scenario, each worker node keeps a local approximation of the global progress, and every time a worker node processes a record, it includes this record to his local approximation and broadcasts the information of this event to the other workers, allowing them to include this event to their own local approximations.

As fault-tolerance mechanism, Naiad incorporates a checkpoint and restore feature in every stateful operator. The system runs this feature as appropriate to ensure that all workers have consistent checkpoints. During the scheduled checkpoints, all processes and message queues are paused, and the system flushes the message queues and initiates a checkpoint on each operator. Once the checkpoint is complete, the worker and message delivery threads are resumed. In case of a process failure, all live processes revert to the last completed checkpoint while the remaining processes take over the data of the failed process.

3.1.9 Streamscope: continuous reliable distributed processing of big data streams

The work of Lin, W et al. [45] introduces StreamScope (or “StreamS") as a stream processing extension for the SCOPE batch-processing system [73]. On StreamScope the user application is compiled into a DAG (Directed Acyclic Graph), which is converted into a logical execution plan. The StreamScope optimizer evaluates the logical execution plan and chooses the configuration that will result in the lowest estimated cost based on the current available resources, data statistics (such as incoming rate), and an internal cost model. Then, the optimized execution plan is mapped into an appropriate number of physical nodes for parallel execution and scaling.

On StreamScope, the user application is created using two base abstractions (rStream and rVertex) that respectively represent the communication channels between the operators and the operator itself. The rStream abstraction maintains supports concurrent readers and writers, has the ability to rewind records, and assigns a continuously increasing number to each record for progress tracking. The rVertex abstraction is responsible for the event processing, it keeps the operator state, and periodically saves its current progress using checkpoints.

StreamScope provides three different recovery approaches for fault tolerance and allows each operator to recover from failures independently. The first approach involves a checkpoint-based recovery mechanism where each operator conducts periodic checkpoints on reliable storage. If a failure occurs, the operator is restarted, and execution resumes from the most recent checkpoint.

The second strategy is replay-based recovery, where entire event windows (e.g., the last 5 minutes) are reprocessed instead of using checkpoints. In this method, the event window that was active at the time of failure is reprocessed, potentially resulting in a large number of records being reprocessed. However, this approach eliminates the overhead associated with checkpointing.

The third strategy is replication-based recovery, where multiple instances execute the same operator simultaneously. In this scenario, some instances may perform checkpoints, while others focus solely on regular application execution. Consequently, the additional latency caused by checkpoints does not impact the final result. If a failure occurs, an instance can recover the checkpoint from another instance to speed up the recovery process.

3.1.10 TimeStream: Reliable Stream Computation in the Cloud

The work of Qian et al. [58] introduces TimeStream as a distributed stream processing system designed for low-latency processing of high volumes of data across large clusters of commodity machines. The development of TimeStream was inspired by the StreamInsight [4] programming model, allowing applications written for StreamInsight to run on TimeStream without any modifications. Therefore, both TimeStream and StreamInsight applications are written in LINQ (Language-Integrated Query) [51], using a Microsoft .NET programming language such as C#.

TimeStream uses a "resilient substitution" mechanism to replace or reallocate operators in case of failures or adjust the system to workload changes. This mechanism assigns a unique sequential identifier to each record on the stream and tracks the output dependencies and the state of each operator to minimize the number of records reprocessed during recovery. When a failure occurs, the recovery process involves initiating a recovery task for each failed operator. This task retrieves the output dependencies and the operator state, recursively initializing a new recovery task for any necessary records unavailable in the input stream. Once all the required data is retrieved, the recovery task clones the operator in its initial state and supplies it with all the recovered records. Given that each operator performs deterministic computation, the data generated after the recovery task is the same as that generated by regular execution.

The resilient substitution mechanism retrieves a set of records to recover together rather than recovering each one individually. For operators whose state depends on all previously processed records, the recursive recovery task would imply restoring and reprocessing all the records since the beginning of the application. For this specific scenario, TimeStream also supports periodic state checkpoints.

3.2 Summary of finds for Fault Tolerance in Stream Processing Systems

Table 3.3 presents a summary of fault tolerance mechanisms and message delivery guarantees discussed in the related work, where it is evident that the checkpoint mechanism emerges as the most popular among the analyzed solutions. However, despite its popularity, there is a lack of uniformity in implementing this mechanism, with each application seemingly addressing the checkpoint in a distinct manner.

Table 3.3: Fault tolerance and processing semantics of the related systems

Year	System	Programming Language	Delivery Guarantee	Fault tolerance		Progress tracking		
				Checkpoint	Log	Transaction-based	Progress-based	Lineage-based
2011	Apache Flink [13, 19, 20]	Java, Scala, Python	Exactly-once	X		X		
2011	Apache Storm [12]	Java, Python	Exactly-once	X		X		
2013	MillWheel [1]		Exactly-once		X	X		
2013	Naiad [54]	C#	Exactly-once	X			X	
2013	TimeStream [58]	C#	Exactly-once	X				X
2014	Apache Spark Streaming [10]	Scala, Python, Java	Exactly-once	X				X
2015	ChronoStream [70]	C++	Exactly-once	X		X		
2015	Google Dataflow [2]	Java, Python, Go	Exactly-once		X	X		
2016	Streamscope [45]	C#	Exactly-once	X				X
2018	Apache Spark Structured Streaming [11, 14]	Scala, Python, Java	Exactly-once	X				X
2025	**ResiPipe**	**C++**	**Exactly-once**	X		X		

Regarding message delivery guarantees, all the analyzed systems provide exactly-once semantics in some way, even if it is necessary to comply with a series of prerequisites to reach this level of conformity. For instance, Spark Structured Streaming offers an exactly-once guarantee only in micro-batch mode; MilWheel provides this guarantee only in "Strong Productions" mode; Flink requires sources and sinks that support transactions; DataFlow requires data sources that support rewind and replay records; And Spark Streaming demands data sources which acknowledgment support and the ability to rewind and replay records as needed.

Concerning the programming language of the analyzed stream processing system, only ChronoStream is written in C++. However, since ChronoStream does not provide public access to its source code or executable binaries, it’s not possible to compare the performance and usability aspects between ChronoStream and ResiPipe. In contrast, the other analyzed systems are written in programming languages such as Java, Scala, Python, and Go. The MillWheel work does not provide any information regarding its programming language, and since it is a proprietary software internally used at Google, it was not possible to find additional public information regarding its implementation.

Compared to the related work, this master’s thesis introduces ResiPipe, a fault-tolerant C++ distributed stream processing library built on top of MPI that eases the development of stream processing applications over clusters of commodity machines. The fault-tolerance approach used by ResiPipe draws inspiration from the same mechanisms used by Apache Flink [20]. The system creates periodic incremental checkpoints through the Asynchronous Barrier Snapshotting (ABS) protocol, and the two-step commit mechanism can be applied to applications requiring exactly-once delivery guarantees.

3.3 Related Research for High-Level Programming Abstractions for Stream Processing

This section presents a set of related high-level programming abstractions for the development of stream processing applications, including domain-specific languages and programming models. Only programming abstractions that provide a streaming interface are included.

3.3.1 DirectFlow

DirectFlow [44] is a domain-specific language (DSL), a compiler and a runtime system for the development of Information-Flow System (a.k.a stream processing systems) for the Java programming language. As depicted in Listing 3.1, the DirectFlow language defines the stream processing components using *pipes* as basic building blocks. The *pipes* can be used to filter, prioritize, duplicate, split or decompose the stream items.

Listing 3.1: DirectFlow example. Extracted from [44].

```
pipe Filter {
    inport in;
    outport out;
    process {
        Object packet;
        packet = in ?;
        out ! packet;
    }
}
```

The DirectFlow *pipe* modules contain the definition of the stream processing operator input and output data ports, the data processing blocks, and an optional Java superclass declaration. The DirectFlow compiler verifies if the *pipe* modules are valid and generates one or more Java classes for each specified module. The DirectFlow runtime system allows the developers to integrate these custom components with standard Java components. The DirectFlow also validates if the components are consistent and does not generate unimplementable pipelines. Finally, the composed pipeline is generated and can be executed as a regular Java program.

3.3.2 OmpSs-2

The OmpSs-2 [29] is the second generation of the OmpSs programming model for the development of concurrent applications. The OmpSs-2 model follows the fundamental design principles of OpenMP, which consists of a high-level abstraction that uses pragmas of the C language to express parallelism, and StarSs (Star Superscalar) which consists of a task-based programming model that aims to the enable automatic exploitation of task parallelism while keeping the application unaware of the target execution platform.

Listing 3.2: OmpSs example. Extracted from [15].

```
int main(int argc, char *argv[])
{
    int x = argc;
    #pragma omp task inout(x)
    {
        x++;
    }
    #pragma omp task in(x)
    {
        printf("argc + 1 == %d\n", x);
    }
    #pragma omp taskwait
    return 0;
}
```

As depicted in Listing 3.2, similar to the OpenMP model, the OmpSs also uses C pragmas to express parallel code regions. The data dependencies are expressed in the *task* construct by using the *in*, *out* and *inout* clauses, which specify the input, output, and input/output data from the task respectively. Currently, the OmpSs-2 has two reference implementations, one of which uses the nOS-V [74] and NODES [16] as runtime systems with the Clang compiler. A second implementation that is being deprecated is based on the Nano6 [17] runtime with the Clang compiler.

3.3.3 OpenStream

The OpenStream [57] programming model is a data-flow extension of OpenMP that aims to ease the development of dynamic dependent tasks by abstracting the task communication, and providing support for complex data structures and unbounded fan-in and fan-out communications.

Listing 3.3: OpenStream example.

```
int main (int argc, char **argv)
{
    int i, x __attribute__((stream));
    for (i = 0; i < 10; ++i)
    {
        #pragma omp task firstprivate (i) output (x)
        {
           x = i;
           printf ("Task 1: write %d to stream.\n", i);
        }

        #pragma omp task input (x) firstprivate (i)
        {
           printf (" => Task 2: read from stream %d (%d).\n", x, i);
        }
    }
  return 0;
}
```

As depicted in Listing 3.3, OpenStream uses OpenMP pragmas to define parallel code regions, and allows the specification of the tasks input, outputs, and private data dependencies. The OpenStream model is implemented on top of the GCC compiler OpenMP expansion pass, which was modified to parse the newly introduced *input* and *output* clauses for the OpenMP task constructs, allowing the GCC compiler to generate an intermediate representation while preserving typing information.

3.3.4 Spidle

Spidle [22] is a domain-specific language (DSL) designed explicitly for developing stream processing applications. The language allows the development of applications through the usage of high-level components that perform filtering and mapping operations over the data stream. Spidle also performs verifications to check if the user-defined components express valid programs to enhance the robustness of the applications.

Listing 3.4: Spidle example. Extracted from [22].

```
filter Weighting {
    interface {
        stream in bit[50][16] e;
        stream out bit[40][16] x;
    }
    import {
        func Weighting_filter from "rpe.c";
    }
    run {
        Weighting_filter (e, x);
    }
}
```

A Spidle is defined as a network of *stream tasks*, in which *Flow declarations* (components) specify how the stream items flow through the stream tasks (nodes and edges), as well as the data type of the items. Listing 3.4 provides an example of a Spidle filter component. The input and output ports of the component are defined inside the *interface* attribute. Functions from other files can be imported using the *import* attribute. And the computation that is executed over the stream items is defined inside the *run* attribute.

3.3.5 StreamIt

StreamIt [66, 67] is a domain-specific language (DSL) and a compiler for the development of stream processing applications in the Java programming language. The StreamIt language has the goal of providing a high-level abstraction for improving the programmer productivity and robustness, while the StreamIt compiler has the goal of performing optimizations to achieve high performance.

Listing 3.5: StreamIt example. Extracted from [66].

```
class FIRFilter extends Filter {
    float[] weights;
    int N;
    void init(float[] weights) {
        setInput(Float.TYPE); setOutput(Float.TYPE);
        setPush(N); setPop(1); setPeek(N);
        this.weights = weights;
        this.N = weights.length;
    }
    void work() {
        float sum = 0;
        for (int i=0; i<N; i++)
            sum += input.peek(i)*weights[i];
        input.pop();
        output.push(sum);
    }
}
class Main extends Pipeline {
    void init() {
        add(new DataSource());
        add(new FIRFilter(N));
        add(new Display());
    }
}
```

As presented in Listing 3.5, the most basic unit of computation inside a StreamIt application is the *Filter*. The filter class implements the *work* function which is responsible for consuming and processing items from the input channels and forwarding its results to the output channels, which consist of FIFO queues declared in the Filter base class. StreamIt supports both shared-memory and distributed architecture using the TCP/IP protocol for processes communication.

3.4 Summary of finds of high-level programming interfaces

Table 3.4 summarizes the high-level programming abstractions discussed in the related work. Among the analyzed abstractions, DirectFlow, Spidle, and StreamIt are external domain-specific languages, which means that they implement their own set of syntax and grammar rules instead of being built on top of another programming language, requiring the user to rewrite any existing application using the DSL syntax.

Table 3.4: Related work comparison for high-level programming interfaces

Work	**Interface Type**	**Supported Architecture**	**Programming Language**	**Supported Runtimes**	**Fault Tolerance Mechanism**
DirectFlow [44]	External DSL	Multi-core	Java	Threads	None
OmpSs-2 [29]	C pragmas	Multi-core, Distributed	C/C++	nOS-V [74], NODES [16], Nano6 [17]	None
OpenStream [57]	C pragmas	Multi-core	C/C++	Threads	None
Spidle [22]	External DSL	Multi-core	C	Threads	None
StreamIt [66, 67]	External DSL	Multi-core, Distributed	Java	Custom	None
SPar [31, 30, 32]	C++ annotations	Multi-core, GPU, Distributed	C++	FastFlow [32], OpenMP [38], Intel TBB [37], GSParLib [61, 62], DSParLib [56, 49], ResiPipe	ABS [20] using ResiPipe

OpenSs-2 and OpenStream extend the existing OpenMP capabilities by introducing new specific stream processing clauses to the existing OpenMP pragmas for task parallelism, requiring fewer modifications over the user source code than external DSLs. However, some level of adaptation over the source code is still expected to implement the task parallelism model used by these abstractions. In comparison to the analyzed programming interfaces, SPar is the only abstraction that uses C++ annotations to express parallel regions in the source code, provides multi-core support, distributed, and GPU architectures, and provides fault tolerance through the ResiPipe runtime, which is further described in Chapter 4.

4. THE RESIPIPE LIBRARY

In this section, we introduce the fundamental concepts of ResiPipe, a C++ library designed to simplify the development of fault-tolerant distributed stream processing applications on clusters of commodity machines. ResiPipe is a header-only library that uses OpenMPI to create/communicate processes across multiple machines. It can be utilized as a standalone library or as a target for code generation within the SPar software ecosystem.

In ResiPipe, the only file that needs to be compiled alongside the user application is the monitor agent. This agent is responsible for launching, monitoring, and restarting processes when failures are detected. For process communication, ResiPipe utilizes OpenMPI, which is an open-source implementation of the Message Passing Interface (MPI) library. However, other MPI implementations can also be easily supported.

One of the main goals of the ResiPipe library is ease of use, allowing the user to create distributed stream processing applications without worrying about communication, serialization, or fault tolerance. To achieve this goal, the ResiPipe library exposes a series of well-defined functions to the end user, divided into four namespaces: pipeline, utility, keyStore, and serialization.

The pipeline namespace includes functions for creating applications using the pipeline parallel pattern, such as the definition of the data source, middle stages, and sink operators. Currently, the library only supports linear pipelines, which are translated into a directed acyclic graph. However, other patterns can be further implemented. The utility namespace groups useful functions, such as getting the machine's name where the process is running, the current MPI process identifier, or the total number of processes composing the pipeline. The keyStore namespace provides functions for storing and retrieving data from an embedded in-memory key store. The serialization namespace provides functions for serializing and deserializing data to a binary representation. Under the hood, ResiPipe uses the cereal library [28], which allows fast serialization of almost every type in the C++ standard and supports the serialization of complex data types such as structs. Each function the ResiPipe API provides will be further detailed in the section 4.1.

As depicted in Figure 4.1, one of the main components of the ResiPipe architecture is the monitor agent, which is responsible for launching the MPI processes over the cluster nodes, monitoring the machines using a heartbeat mechanism, and restarting the application in case of failure. At startup, the monitor process receives a series of arguments, such as the application path, the hostname of the machines, and the number of processes that must be executed per machine. A detailed summary of the arguments supported by the monitor agent is presented in Table 4.1.

The heartbeat mechanism used by the monitor agent can be classified as a partial synchronous unreliable failure detection mechanism, which means that it can produce

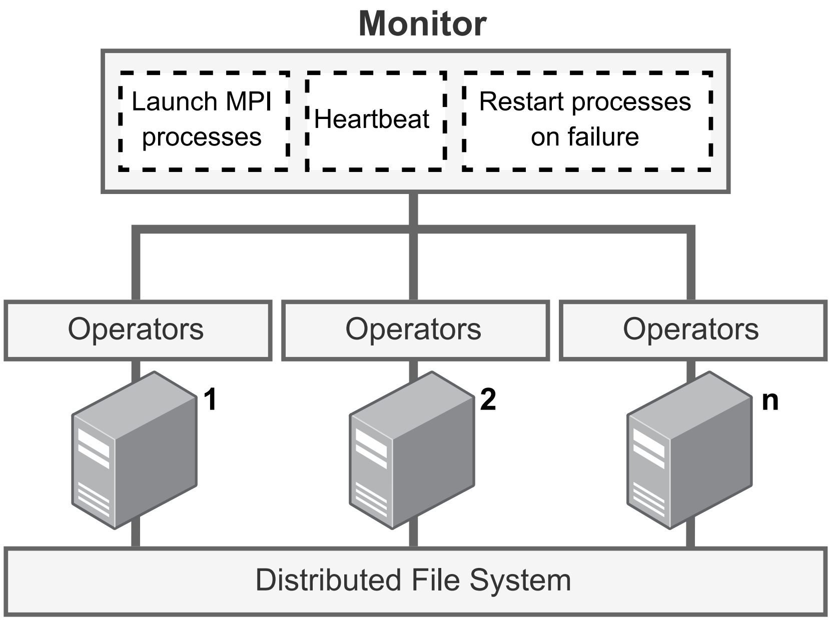


Figure 4.1: ResiPipe Architecture

Table 4.1: Summary of the monitor process arguments

Argument	Description	Type	Default
-np	Number of processes.	int	
-file	Application path.	string	
-host	List host machines. Format: "machine:max-processes".	string	
-timeout	Defines a time limit in seconds for the application execution.	int	
-max-retries	Number of times the application will be restarted in case of failure.	int	3
-clear-logs	Clear the existing snapshots before running the application.	bool	false
-snapshot-each-time	Time between each snapshot in seconds. (-1 for disable)	int	30
-snapshot-each-records	Number of records between each snapshot. (-1 for disable)	int	-1

false positives by misidentifying a live machine as a failed machine due to network instabilities, causing the monitor agent to restart the application without necessity.

When the monitor agent is initiated, it starts by verifying whether all the host machines are available before launching the user application. If one or more machines are unavailable, the processes are redistributed over the remaining machines, possibly exceeding the maximum number of processes per machine defined at the startup. Once the user application is running, the monitor agent starts to send heartbeat requests every 5 seconds to verify if the machines are alive. If a machine previously identified as unavailable becomes available, the monitor automatically redistributes the processes over the newly available machines. Using the same logic, if a machine previously identified as available stops responding to the heartbeat requests, the monitor tags the machine as unavailable and redistributes its processes over the remaining machines.

Regarding the high availability of the monitor agent, it can be executed both over a dedicated machine or over one machine that shares the monitor agent and the user application processes. Ideally, the monitor agent would be deployed across all machines, using a consensus algorithm to elect a leader. This setup would reduce the risk of downtime if the machine running the monitor agent fails. However, this kind of topology is not yet supported and needs further implementation studies since it may handle network partition (or split-brain) situations.

Also, regarding the current topology implementation, it is recommended that all the machines must be connected to a fault-tolerant distributed file system, such as the Hadoop Distributed File System (HDFS), to store snapshots in production environments. This setup ensures that any machine can retrieve the latest complete snapshot of any process during failure recovery, even if the process was running on a different machine. However, for testing environments, ResiPipe can also be deployed over a network file system (NFS) for distributed execution or over a regular file system such as NTFS (New Technology File System) or Ext4 (fourth extended file system) for single-node execution. In this case, it is recommended that the user create periodic backups of the snapshot files over reliable storage to prevent data loss in case of machine or storage failure. The ResiPipe library also can be easily modified to store snapshots in distributed databases with concurrent access support.

4.1 Implementation and usage

With ease of use and fault tolerance as the primary goals, the ResiPipe library provides a high-level interface for the development of stream processing operators, including a fault-tolerant in-memory key store, periodic state checkpointing, support for different message delivery modes between operators, and a callback feature that allows the execution of user-defined functions after certain events of the application life cycle. In this section, we describe the implementation and usage of the main features of the ResiPipe library, which are provided through a series of well-defined functions, also referred to as ResiPipe API, further summarized in Table 4.2.

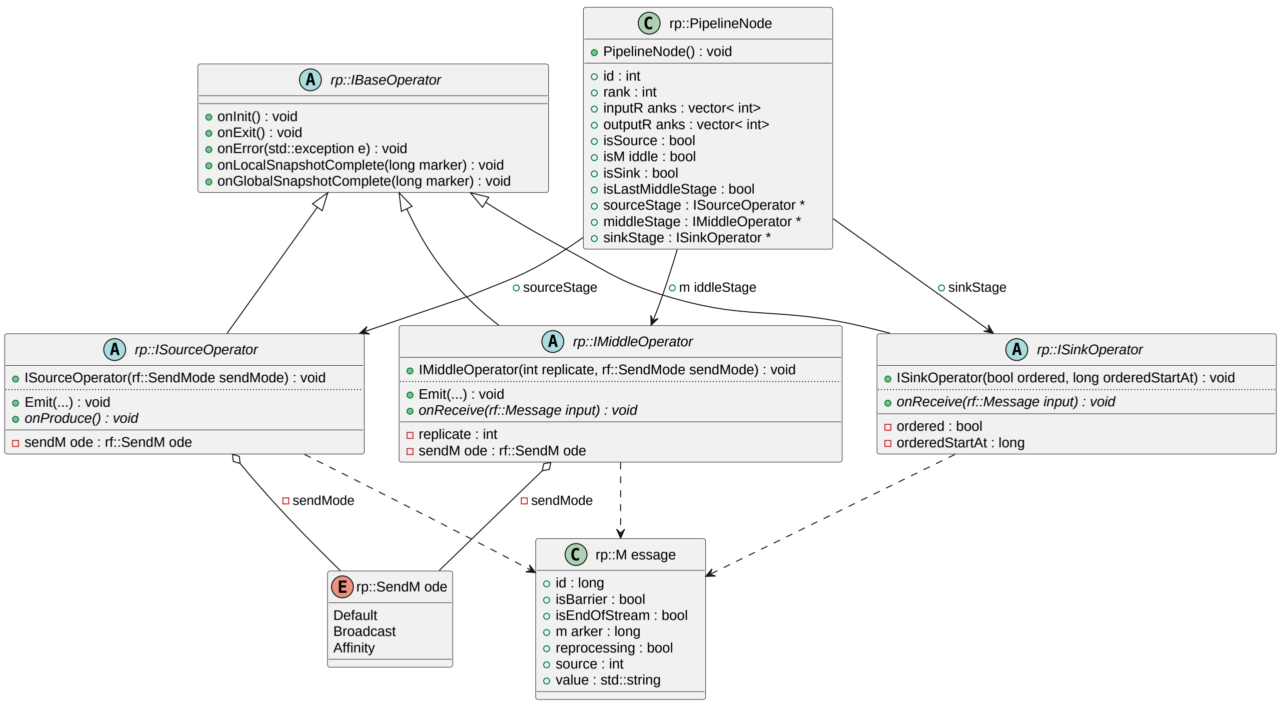
In summary, the ResiPipe applications are expressed by the implementation of three C++ class interfaces, named *ISourceOperator*, *IMiddleOperator*, and *ISinkOperator*, that respectively represent the data source responsible for ingesting data from external systems, the middle operators responsible from processing the stream items, and the sink operator responsible for outputting the results to external systems. All the ResiPipe operator interfaces inherit the *IBaseOperator* interface, which contains the basic functions and methods shared between all the ResiPipe operators. All the operators that send messages inside the application have the *sendMode* attribute, which can vary between the default, broadcast, and affinity mode, which are further detailed in Section 4.5.

In addition, all the operators exchange messages using the *rp::Message* struct, which contains information regarding the message's unique identifier, snapshot marker,

Table 4.2: ResiPipe API functions

Namespace	Function	Returns	Description
rp	*init(argc, argv)*	void	Initializes the ResiPipe environment
rp	*finalize()*	void	Finalizes the ResiPipe environment
rp::serialization	*serialize(value)*	std::string	Serialize the given value to a binary representation.
rp::serialization	*deserialize<type>(value)*	template type	Deserialize a string to the given type.
rp::keyStore	*store(key, value)*	void	Store a value in the key store.
rp::keyStore	*retrieve<type>(key)*	template type	Retrieve a value from the key store.
rp::keyStore	*contains(key)*	bool	Verifies if the given key exists on the key store.
rp::pipeline	*run(...)*	void	Starts the pipeline execution.
rp::utility	*getRank()*	int	Returns the process MPI identifier (rank).
rp::utility	*getCommunicatorSize()*	int	Returns the MPI communicator size.
rp::utility	*getProcessorName()*	std::string	Returns the processor/machine name.
rp::utility	*getExecutionPlan()*	struct	Returns a struct containing information regarding the distribution of the processes among the machines.
rp::utility	*getNode()*	struct	Returns a struct containing information regarding the current node, such as input and output channels.

the identifier of the operator that sent the message, and the message payload serialized into a binary string format. Internally, ResiPipe wraps operators classes implemented by the user inside a *rp::PipelineNode* class, which contains information regarding the node identifier, the MPI processes rank that will execute the operator, and the input and output channels of the operator. Figure 4.2 presents a simplified class diagram demonstrating the relationship between these interfaces within ResiPipe.



rp::PipelineNode
PipelineNode() : void
id : int
rank : int
inputR anks : vector< int>
outputR anks : vector< int>
isSource : bool
isM iddle : bool
isSink : bool
isLastMiddleStage : bool
sourceStage : ISourceOperator *
middleStage : IMiddleOperator *
sinkStage : ISinkOperator *
rp::IBaseOperator
onInit() : void
onExit() : void
onError(std::exception e) : void
onLocalSnapshotComplete(long marker) : void
onGlobalSnapshotComplete(long marker) : void
sourceStage
m iddleStage
sinkStage
rp::ISourceOperator
ISourceOperator(rf::SendMode sendMode) : void
Emit(...) : void
onProduce() : void
sendM ode : rf::SendM ode
rp::IMiddleOperator
IMiddleOperator(int replicate, rf::SendMode sendMode) : void
Emit(...) : void
onReceive(rf::Message input) : void
replicate : int
sendM ode : rf::SendM ode
rp::ISinkOperator
ISinkOperator(bool ordered, long orderedStartAt) : void
onReceive(rf::Message input) : void
ordered : bool
orderedStartAt : long
sendMode
sendMode
rp::SendM ode
Default
Broadcast
Affinity
rp::M essage
id : long
isBarrier : bool
isEndOfStream : bool
m arker : long
reprocessing : bool
source : int
value : std::string

Figure 4.2: Simplified ResiPipe class diagram

Listing 4.1 provides an example of a source operator in ResiPipe. Following the stream processing model concepts, the source operator is responsible for ingesting data from external systems such as databases, messaging systems, or sensors. It is expected that ResiPipe supports any external system that provides a C++ API for integration such as librdkafka[1] for Apache Kafka and rabbitmq-c[2] for RabbitMQ. In ResiPipe, the source operator class must implement the *ISourceOperator* interface, which requires the definition of the *onProduce* method. This method specifies how data is ingested from external systems. Within the *onProduce* method, users can call the *Emit* method to send new records over the pipeline. The *Emit* method takes two arguments: The first is a numeric identifier that must be incremental and unique for each record. If the sink operator is configured to maintain order, this identifier will help ensure the correct sequence of messages. The second argument of the *Emit* method is the record itself.

In ResiPipe, all operators exchange messages using the *string* datatype. Therefore, if the record consists of a struct, class, number, pointer, or any other variable type, it must be serialized to a string format using the provided *serialize* function. When the record is received at the other end, it should be deserialized to its original type using the *deserialize* function. If a programmer needs to use a datatype not supported by the Cereal library, which covers almost all types in the C++ standard, they must implement their own serialization functions.

Listing 4.1: ResiPipe source operator example

```
// Source Operator
class Source : public rp::ISourceOperator
{
   public:
    void onProduce()
    {
        // Continuously consume events from an external system
        while(true)
        {
            Event item = consumeEvent();
            Emit(item.id, rp::serialization::serialize(item));
        }
    }
};
```

Listing 4.2 provides an example of a middle operator, also known as a worker stage. In ResiPipe, all middle operators must implement the *IMiddleOperator* interface. This interface requires the implementation of the *onReceive* method, which defines how the middle operator handles the reception of new records. Within the *onReceive* method, the user receives the input record wrapped inside the *Message* struct. This struct contains information regarding the record identifier, the operator that sent the record, the current snapshot marker, a flag indicating whether the current marker is being reprocessed, and

[1]https://github.com/confluentinc/librdkafka
[2]https://github.com/alanxz/rabbitmq-c

the record data. Inside the *onReceive* method, the user can call the *Emit* method to forward results to the next operator in the pipeline. Currently, it is not possible to choose which operator the message will be sent to since, under the hood, ResiPipe abstracts the load balancing and automatically chooses the output operator based on the application pipeline. Unlike the source operator, the *Emit* method of the middle operator only takes the output record as an argument. The record identifier is automatically assigned as the same identifier as the input record.

Listing 4.2: ResiPipe middle operator example

```
// Middle Operator
class Middle : public rp::IMiddleOperator
{
   public:
    void onReceive(const rp::Message &input)
    {
        // Deserialize the input
        Event item = rp::serialization::deserialize<Event>(input.value);

        // Run some computation
        int output = compute(value);

        // Forward the result
        Emit(rp::serialization::serialize(output));
    }
};
```

Listing 4.3 presents an example of a Sink operator. In ResiPipe, the sink operator must implement the *ISinkOperator* interface, which, similar to the middle operator interface, also requires the specification of the *onReceive* method and also receives the input data wrapped over the *Message* struct. Conceptually, the *ISinkOperator* interface is almost identical to the middle *IMiddleOperator* interface. However, it does not provide a *Emit* method since the sink operator is always the last operator of the pipeline.

Listing 4.3: ResiPipe sink operator example

```
// Sink Operator
class Sink : public rp::ISinkOperator
{
   public:
    void onReceive(const rp::Message &input)
    {
        // Deserialize the input
        int value = rp::serialization::deserialize<int>(input.value);
        // Print the result
        printf("Event Id: %lu, Value: %d\n", input.id, value);
    }
};
```

Finally, the listing 4.4 presents an example of how the ResiPipe API is called inside the *main* function of the application. All the features of ResiPipe, including the serialization functions and operator interfaces, are available using a single include statement. At the beginning of the application, the user must call the *init* function to start the ResiPipe environment. The *init* function receives the *argc* and *argv* values of the application and is responsible for initializing the underlying MPI environment, starting the communication, snapshot processes, and recovering previous snapshots at startup.

After initializing the ResiPipe environment, the user can declare the pipeline operators and set configurations, such as the number of replicas of each operator, and define whether or not the sink operator must be ordered. Once the ResiPipe environment is started and the operators have been properly declared and configured, the user must call the *pipeline::run* function to start the pipeline execution. If the application requires multiple middle-stage operators, the user can give a vector of operators to the *pipeline::run* function. At the end of the application, the user must call the *finalize* function to properly stop the ResiPipe processes and finalize the underlying MPI environment.

Listing 4.4: ResiPipe main example

```
#include "resipipe/resipipe.hpp"

int main(int argc, char **argv)
{
    // Initialize the environment
    rp::init(argc, argv);

    // Create the operators
    Source source;
    Middle middle;
    Sink sink;

    // Set the number of replicas
    middle.setReplicate(2);

    // Set the sink as ordered
    sink.setOrdered(true);
    // Start the pipeline
    rp::pipeline::run(&source, &middle, &sink);

    // For multiple middle operators:
    // rp::pipeline::run(
    //     &source,
    //     std::vector<rp::IMiddleOperator*>{
    //         &middle1,
    //         &middle2,
    //         ...
    //     },
    //     &sink
    // );

    // Finishes the environment
    rp::finalize();
}
```

The ResiPipe library provides a callback feature that allows the execution of user-defined functions after specific events of the application life cycle through the implementation of the methods *onInit*, *onExit*, *onError*, *onLocalSnapshotComplete*, and *onGlobalSnapshotComplete* on the operator class. This feature was developed to allow users to implement a two-phase commit mechanism for applications that interact with external systems, such as databases or messaging brokers.

1. **onInit**: This method is executed at the operator startup before it starts to ingest messages, allowing the user to allocate resources or open connections to external systems.

2. **onExit**: This method is executed just before the operator exits, allowing the user to release resources before the application stops.

3. **onError**: This method is triggered when a failure is detected, enabling users to handle exceptions before the application exits or restarts. The onExit method is always triggered after the execution of the onError method.

4. **onLocalSnapshotComplete**: This method is executed after the operator completes a local snapshot.

5. **onGlobalSnapshotComplete**: This method is executed after a global snapshot completes, allowing the user to output data to external systems with the guarantee that any data before the global snapshot will not be reprocessed in case of failure.

4.2 State Management

ResiPipe implements a built-in fault-tolerant key store for state management where stateful operators can store variables of almost every type of the C++ 11 standard, including complex types such as structs. The values inside the key store are limited to the scope of the operator process, so one process cannot access the values from the key store of another process. As exemplified on the Listing 4.5, the user can call the functions *store*, *retrieve*, and *contains* of the *keyStore* namespace to manipulate values inside the embedded key store.

Listing 4.5: Key Store usage example

```
// Stores a value in the key store
rp::keyStore::store("key", 123);

// Retrieves the value from the key store, assuming zero as the default value.
int value = rp::keyStore::retrieve<int>("key").value_or(0);

// Verifies if the key is defined
bool exists = rp::keyStore::contains("key");
```

The function *store* receives the key that identifies the value as its first argument and the value itself as the second argument. When the *store* function is called, it serializes the given value to binary representation, then places the value inside an in-memory data structure, similar to an *std::map*. The *retrieve* function receives the key as its first argument and the type that the value must be parsed as a template. The *retrieve* function returns an *std::optional* for the given type, allowing the user to verify if the key contains a value using the *empty()* function or define a default value using the *value_or()* function. Finally, the *contains* function verifies if a given key exists inside the key store without parsing its value; it receives the key as the first argument and returns a boolean.

4.3 Progress tracking

In ResiPipe, each message emitted by the source operator is associated with a unique numeric identifier. This identifier is provided by the user as the first argument of the *emit* function and represents a unit of progress within the application, such as the current index of a loop, the offset position from an external messaging system, or the line number in a text file. ResiPipe automatically stores the identifier of the last emitted message in the key store, which is accessible by the key "*RESIPIPE_SOURCE_POSITION*". In the event of a failure or system restart, this stored identifier enables the application to resume from the latest complete checkpoint, thereby avoiding the need to reprocess all records from the beginning. Also, for applications that require ordered sink operators, the message identifier can be used to maintain a processing order among the messages.

It is important to note that the position retrieved from the latest checkpoint may not correspond to the last emitted message, as checkpoints are typically created at regular time intervals rather than after each emitted record. Consequently, both middle and sink operators must handle potential duplicate messages. To ease the management of this situation, all the messages received by middle and sink operators are encapsulated in a struct that includes the message identifier, the identifier of the processes that have sent the message, the current checkpoint marker, the message value, and a flag indicating whether the checkpoint marker is being reprocessed or not. If the user application interacts with external systems, this information can be used to implement additional validations for records flagged as being reprocessed.

4.4 Fault tolerance

Andrade, Gedik, and Turuga [9] state that a failure in a stream processing system can happen due to a software bug, such as a mistake in the application logic, a failure at the system runtime, or a failure in the computational infrastructure, such as an unavailable machine, a network failure or storage failure. Since a failure in the user logic may require changes in the application source code, ResiPipe focuses on providing fault tolerance for fail-stop failures, crash failures, and infrastructure-level failures. Omission failures are handled by the TPC protocol used by MPI, which tries to resend messages in case of network malfunction. By default, if the MPI tries to send a message to an inaccessible machine, it generates a system crash, and ResiPipe restarts the application. ResiPipe also assumes that the software stack and the network inside the cluster are closed and controlled by the cluster infrastructure team, so ResiPipe does not handle Byzantine failures such as the intervention of malicious agents.

We choose to implement the fault tolerance mechanism at the runtime level instead of using fault-tolerant implementations of MPI, such as LAM/MPI, FT-MPI, or the ULFM (User-Level Failure Mitigation) implementation of OpenMPI, because the mechanisms employed by these MPI implementations are not specifically designed for stream processing systems and can be discontinued at any time. OpenMPI deprecated the checkpoint/restart feature in version 1.7 due to its limited adoption and lack of maintenance. Implementing the fault tolerance mechanism at the runtime level also allows for greater flexibility to implement exactly-once mechanisms, such as the two-step commit protocol.

Similarly to Apache Flink, the ResiPipe library implements a distributed snapshot mechanism for fault tolerance named Asynchronous Barrier Snapshotting (ABS) [20], which is conceptually similar to the classical Chandy-Lamport algorithm [21]. Under regular execution, the ResiPipe source operator periodically injects a snapshot message over the data stream. The snapshot message only carries a snapshot marker (aka. barrier), which consists of a unique sequential number that identifies the current snapshot. After the emission of the marker, all the messages emitted by the source operator are tagged with the same marker, allowing the user to identify which snapshot marker is currently being processed. By default, the ResiPipe library emits a new snapshot marker every 30 seconds. However, the user can set a custom snapshot interval at the application startup.

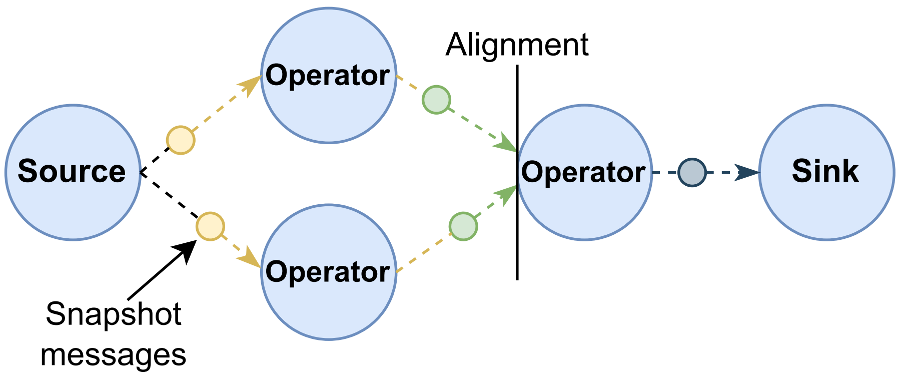


Figure 4.3: Representation of the snapshot alignment

As depicted in Figure 4.3, when a middle or sink operator receives a new snapshot marker, it blocks the input channel from which the snapshot marker has been received in a step called *alignment* and waits until the same snapshot marker arrives on all its input channels. Once the operator receives the same snapshot marker from all its input channels, it stores the current state (the values at the embedded key store) over a fault-tolerant distributed file system, broadcasts the new snapshot marker to all its output channels, and unblocks all its input channels allowing the application to processed with the regular execution. The global snapshot is considered complete once all the operators have received the snapshot marker and persisted their states over reliable storage.

In the event of a failure or system restart, at startup, all the processes restore the latest complete snapshot. It is essential to highlight that the events being processed during the failure/restart are lost, and after the startup recovery, the system will continue processing the events from the position of the last snapshot. Therefore, for data sources that cannot rewind and replay records, all received records must be persisted in reliable storage before processing, allowing these records to be recovered in the event of failure/restart. However, the current implementation of the ResiPipe library does not provide functions that ease this process, requiring the user to implement a logging mechanism for this situation. Also, restarting the application from its last snapshot may result in record reprocessing, which generates duplicated interactions over external systems. For applications that require exactly-once semantics, a sink operator with idempotent capabilities is required, or a two-phase commit mechanism, which can be employed using the callback feature.

4.5 Load balancing

Regarding load balancing, in ResiPipe, all the operators exchange messages using the on-demand model, in which the operators send their results to the first channel that asks for work. This strategy aims for a better load balance between the operators since it prioritizes sending work to operators who are not busy at the expense of exchanging extra work request messages. In the current implementation, each operator starts by sending work requests to all its input channels. These work request messages are sent asynchronously using MPI non-blocking communication. After sending the work request messages, the operator waits until a work message arrives from one of its input channels. If the next stage of the pipeline is the sink operator, the output is directly sent to the sink since the current implementation only supports one sink per pipeline. Otherwise, it sends the output result to the first output operator that has requested work.

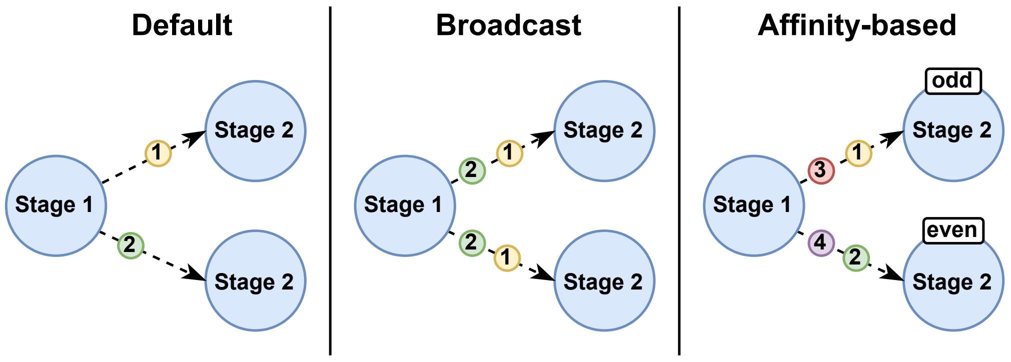


Figure 4.4: Message delivery modes

Also, in the ResiPipe library, each stage within the application pipeline can send messages distinctly based on the application's requirements. Figure 4.4 illustrates the currently supported message delivery modes: the default mode, the broadcast mode, and the affinity-based mode. In the default mode, each message is sent just once to the first operator that has requested work. This mode is the most straightforward and is suitable for most applications. In the broadcast mode, a copy of the same message is sent to all the output channels. This mode is useful for applications where each replicated stage performs a different action over the same record. In the affinity-based mode, all records with a common user-defined property are grouped into the same output channel. For example, in Figure 4.4, all messages with odd numbers are always sent to the first output channel, while messages with even numbers are sent to the second output channel.

4.6 Final Remarks

This chapter introduced the basic concepts, implementation, and usage examples of ResiPipe, a C++ library that eases the development of fault-tolerant distributed stream processing applications on clusters of commodity machines. The ResiPipe library abstracts the implementation of the pipeline parallel pattern, message passing between processes, data serialization, fault tolerance, and exactly-once semantics.

In summary, ResiPipe applications are created by implementing a set of C++ class interfaces representing different processing stages (also named operators) within a stream processing application. To achieve fault tolerance, ResiPipe implements the Asynchronous Barrier Snapshotting (ABS) protocol to create periodic consistent snapshots of the application state and relies upon a monitor agent that uses a heartbeat mechanism to detect when a machine becomes unavailable inside the cluster and automatically restart and redistributes the processes among the remaining machines.

To achieve exactly-once semantics, ResiPipe requires replayable data sources or the implementation of a logging mechanism that saves the input records before processing. Regarding the sink operator, to achieve exactly-once semantics, ResiPipe requires a sink operator with idempotent capabilities or the implementation of a two-step commit mechanism in which the output results are only visible to external systems after a global snapshot encompassing these results is complete. To implement the two-step commit mechanism, ResiPipe provides a feature that allows the execution of user-defined functions after certain events in the application lifecycle, such as when an operator starts or exits, when an exception is detected, or when a local or global snapshot is complete.

However, in its current implementation, ResiPipe has some limitations that could be improved in future works. These limitations can be summarized as follows:

- **Supported parallelism**: ResiPipe only provides support for linear pipelines without feedback loops and only allows a single source and sink operator per application.
- **Failure recovery**: ResiPipe does not support dynamic process recovery. When a failure happens, the entire application must be restarted from the latest snapshot.
- **Dynamic scalability**: ResiPipe does not support dynamic process scalability. After the application starts running, it is not possible to dynamically change the replication factor of the operators.
- **High-Availability**: Currently, there is no High-Availability deployment for the monitor agent, which is responsible for detecting machine failures inside the cluster. If the machine that is running the monitor agents fails, ResiPipe is unable to restart and recover the application.

5. RESIPIPE EVALUATION

In this section, we present the results and the methodology used to evaluate the performance overhead introduced by the ResiPipe library compared to the OpenMPI, DSParLib, and MPR implementations of the same applications. We also evaluate aspects related to the throughput, failure recovery, and scalability concerning the number of nodes. Thus, Section 5.1 describes the environment where the experiments are conducted. Section 5.2 describes the data collection methodology. Section 5.3 presents the results of the performance experiments. Section 5.4 describes the results of the failure recovery experiments. Section 5.5 presents a programmability evaluation of the parallel interfaces. Finally, Section 5.6 presents the final remarks obtained from the experiments.

5.1 Execution environment

The Cerrado cluster provided by the High-Performance Computing Laboratory (LAD)[1] of the Pontifical Catholic University of Rio Grande do Sul (PUCRS) was used to execute the experiments. The cluster consists of 14 nodes running the Ubuntu 20.04.6 LTS operating system with OpenMPI 1.10.7. Each node is equipped with two Intel(R) Xeon(R) CPU E5-2620 0 @ 2.00GHz processors, totaling 12 cores and 24 threads, 24 GB of RAM, 2 Gigabit-Ethernet networks, and one Infiniband QDR 4x (32 Gbit/s) network. Among the clusters provided by LAD, we choose the Cerrado cluster since it features an Infiniband QDR network, which can benefit the execution of distributed applications.

5.2 Data collection methodology

To collect metrics regarding the throughput, CPU, and memory consumption of the applications over a distributed environment, a new utility library named "Spot" has been developed. Spot is a C++ library developed for collecting memory and CPU usage metrics in shared memory or distributed (MPI) applications. The motivation behind developing this library was the lack of monitoring tools such as Zabbix, Ganglia, or Nagios available to users of the High-Performance Computing Laboratory (LAD) at PUCRS. The Spot library gathers metrics by asynchronously querying the Linux files "*/proc/meminfo*" to obtain memory usage and "*/proc/stat*" to gather CPU usage. As a result, the collected metrics reflect the overall resource consumption of the machine, not just the consumption of the current process. Currently, the output generated by Spot is a .csv file containing the following information:

[1] https://www.pucrs.br/ideia/laboratorios-ideia/lablad

- Probe timestamp (ms)
- Machine name
- Process rank (only for distributed applications)
- Total machine memory (Kb)
- Baseline free memory (Free memory before the application starts in Kb)
- Baseline available memory(Available memory before the application starts in Kb)
- Baseline used memory (Used memory before the application starts in Kb)
- Free memory (Kb)
- Available memory (Kb)
- Used memory (Kb)
- Overall CPU usage (%)
- Per core CPU usage (%)

At the end of the application execution, a .csv summary file is also generated containing the following information:

- Machine name
- Process rank (only for distributed applications)
- Startup timestamp (ms)
- End timestamp (ms)
- Total elapsed time (ms)

Listing 5.1 presents an example of usage of the Spot library for MPI applications. This library can be easily integrated into existing applications by including either the header file "spot/distributed.hpp" for MPI applications or "spot/sharedmemory.hpp" for shared memory applications. Spot provides a "start" function to initiate monitoring and a "stop" function to stop the metrics collection. Additionally, custom parameters can be set at the "start" function, such as the probe interval in milliseconds, output directory, output file name, format, and the default MPI communicator. Since the Spot library asynchronously collects the metrics using a different thread from the main program, it is expected that it does not introduce a significant overhead over the application execution. However, as with any other process in the machine, this metrics collection thread also competes for the machine's resources, and theoretically, if the host machine does not have enough resources to run simultaneous processes (e.g., a single-core CPU), it can introduce a noticeable overhead due to resource contention.

Listing 5.1: Spot library usage example in an MPI application

```
#include "spot/distributed.hpp"

int main(int argc, char **argv)
{
    // Initialize MPI
    MPI_Init(&argc, &argv);

    // Spot configuration
    spot::distributed::Config cfg;
    cfg.outputPrefix = "sample";

    // Start monitoring
    spot::distributed::start(cfg);

    // Do some work
    compute();

    // Stop monitoring
    spot::distributed::stop();

    // Finalize MPI
    MPI_Finalize();
}
```

5.3 Performance experiments

This section introduces the applications chosen to evaluate the ResiPipe performance compared to DSParLib [49], MPR [46], and OpenMPI [52], as well as the methodology used to collect the metrics (throughput, CPU, and memory usage) over a distributed environment. Due to time constraints, this work does not include a comparison with Java-based stream processing systems, such as Apache Flink, as it would require writing the experiments in both Java and C++. The experiment set consists of 6 stream processing applications with varied computational characteristics. Since the current implementation of the MPR framework only supports 3-stage pipelines, five of six applications on the experiment set (prime numbers, mandelbrot, eye detection, and bzip2 compression and decompression) consist of simple 3-stage pipelines. As depicted in Figure 1, a 3-stage pipeline contains one source operator responsible for producing data, one middle operator (with possible replication) responsible for processing the stream items, and one sink operator responsible for accumulating the results. To evaluate the support of pipelines with multiple processing stages on ResiPipe and DSParLib, one of the applications (sentiment analysis) consists of a four-stage pipeline containing two different middle-stage operators.

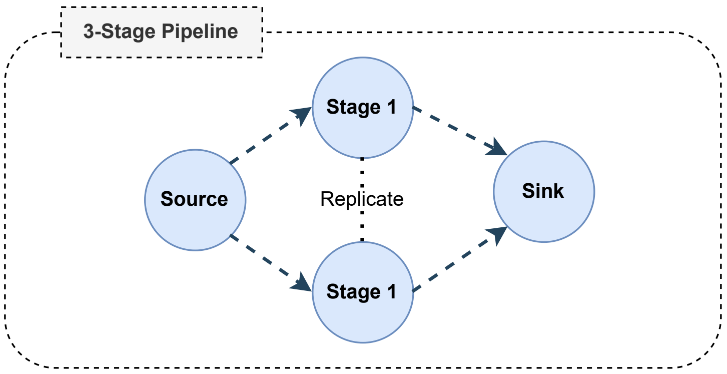


Figure 5.1: Simple 3-stage pipeline graph

All performance experiments were conducted with the fault tolerance mechanism disabled in ResiPipe, as the goal is to evaluate the API overhead compared to other evaluated libraries that do not offer this feature. A set of experiments specifically designed to assess the overhead of the fault tolerance mechanism is detailed in Section 5.4. Each experiment was conducted 10 times for greater precision. All the applications were developed using on-demand scheduling for load balancing.

5.3.1 Prime Numbers

Prime Numbers is a simple synthetic application that calculates the number of primes within a given range using a brute-force algorithm [31]. The algorithm verifies if, for a given number *n*, there is any number between 2 and *n*-1 that *n* is divisible. Despite being simple, this application is highly unbalanced since even numbers are much easier to process than odd numbers. Our implementation of the prime numbers application is based on the implementation made by the DSParLib work [49], which uses a farm-like pattern (3-stage pipeline) with reordering of the records disabled. In this application, a source operator produces a range of integer numbers (from 2 to 500000), a middle operator (with replication) receives a number, validates if it is a prime, and then forwards the result to the sink operator. The sink operator receives the result as a boolean and increments a counter for each positive value it receives.

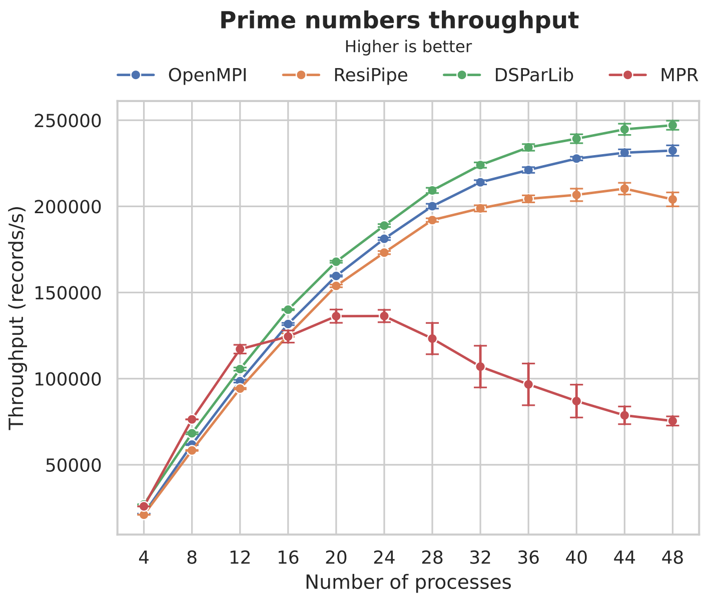


Figure 5.2: Prime numbers throughput

Figure 5.2 presents a comparative analysis of the throughput between the OpenMPI, ResiPipe, DSParLib, and MPR implementations of the prime numbers application, according to the increase in the replication factor of the middle operator (from 4 to 48 processes). DSParLib outperforms the other implementations, notably performing better than OpenMPI when using 4, 8, and 12 processes. This finding aligns with the observations made by Junior et al. [49], who attributed this advantage to the superior parallelism strategies employed in DSParLib. ResiPipe and OpenMPI exhibit very similar throughput, although ResiPipe tends to be slightly slower than OpenMPI, regardless of the number of parallel processes. Meanwhile, MPR demonstrates a higher throughput than OpenMPI and ResiPipe when using 4, 8, and 12 processes.

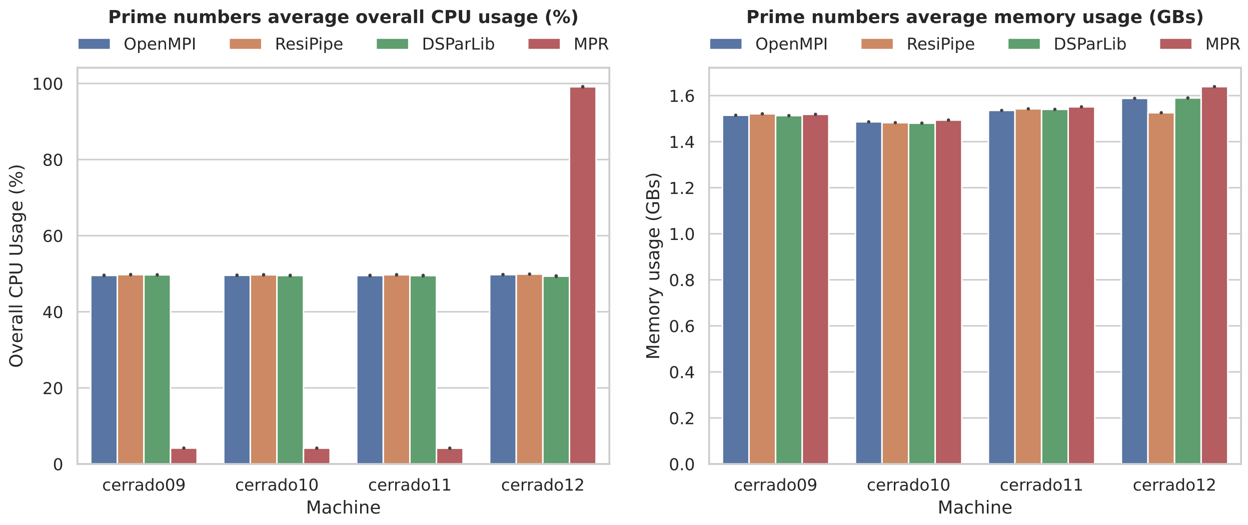


Figure 5.3: Prime numbers average CPU and Memory usage

Figure 5.3 overviews the average CPU and memory usage of the prime numbers experiment with 48 processes. The resource utilization indicates that MPR groups all the processes on a single machine. This behavior has been observed over all the experiments and may explain why it struggles to scale efficiently as the number of parallel processes increases. In terms of CPU utilization, OpenMPI, ResiPipe, and DSParLib presented a maximum of 50% CPU usage. This is expected, given that the experiment only employed 48 processes, and each machine in the cluster is equipped with a CPU with 12 cores and 24 threads. Regarding memory usage, MPR consumed 3.84 GB of the node performing the computations and, surprisingly, still used 1.55 GB across the remaining nodes. OpenMPI's memory consumption ranged between 1.76 GB and 1.86 GB. DSParLib demonstrated the lowest memory usage, with values between 1.64 GB and 1.80 GB, while ResiPipe exhibited the highest memory consumption, ranging from 1.84 GB to 1.86 GB.

5.3.2 Mandelbrot

Mandelbrot is a mathematical application that aims to create a fractal image from a set of complex numbers [23]. Our implementation of the Mandelbrot application is based on the implementation made by the MPR work [46], which uses a 3-stage pipeline graph. In this implementation, a source operator emits 2000 integer numbers representing the lines of the image, and a middle operator (with replication) calculates the pixels for the given line and forwards a matrix of pixels for the sink operator. The sink operator receives the matrix of pixels for each line in an orderly manner and assembles the final image. The final image is only kept in memory to avoid the overhead of writing the image over disk.

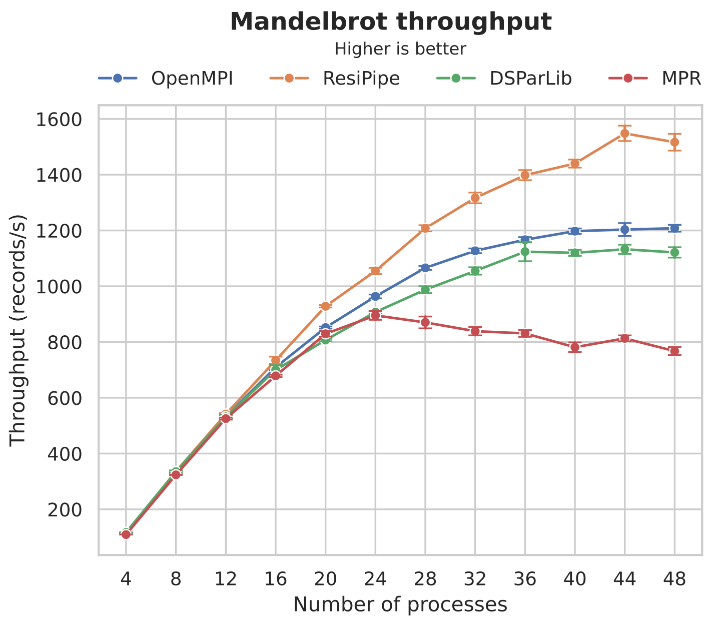


Figure 5.4: Mandelbrot throughput

Figure 5.4 illustrates the average throughput (records per second) of the Mandelbrot application using 4 to 48 parallel processes. Among the evaluated libraries, ResiPipe achieved the best speed-up, increasing the throughput from 115.07 with 4 processes to 1516.38 with 48 processes. DSParLib and OpenMPI delivered similar results, with the throughput increasing from 117.18 and 115.91 with 4 processes to 1121.28 and 1207.91, respectively, with 48 processes. In contrast, MPR presented the lowest scalability, starting at 109.46 with 4 processes and reaching its best throughput of 895.30 with 24 processes. MPR was unable to scale efficiently beyond 24 processes.

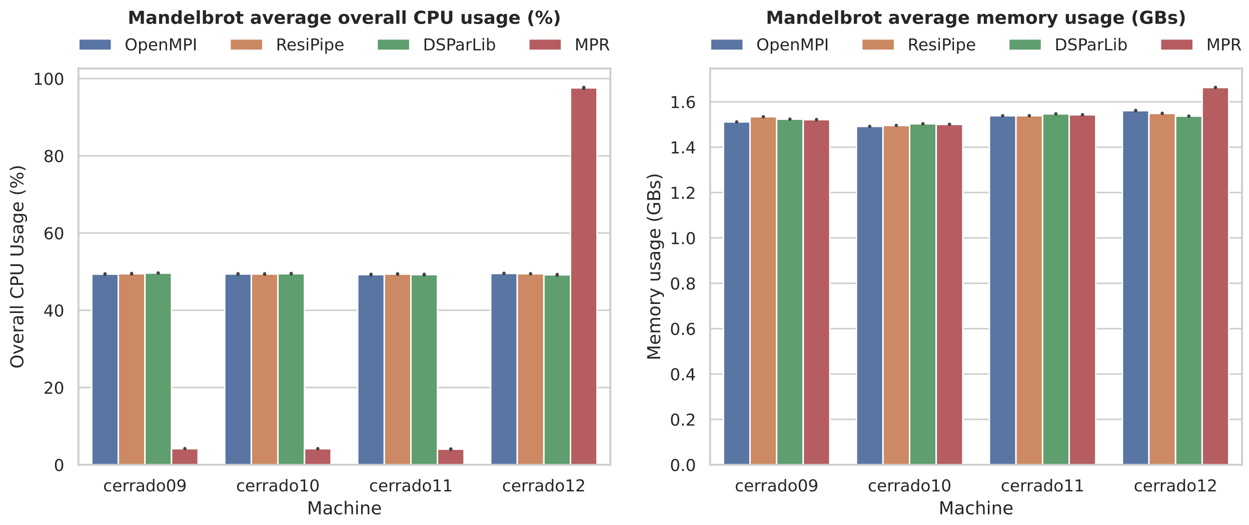


Figure 5.5: Mandelbrot average CPU and Memory usage

Figure 5.5 presents the average CPU and memory usage across the cluster machines for the Mandelbrot application with 48 processes. MPR concentrates all its worker processes into a single machine, achieving a peak CPU utilization of 97.59% and 3.86 GB of memory consumption over a single node. In contrast, OpenMPI, DSParLib, and ResiPipe distribute their workload evenly across all nodes, achieving approximately 50% CPU utilization per node. DSParLib exhibited the lowest memory usage among the libraries, peaking at 1.78 GB. OpenMPI followed closely with a peak memory consumption of 1.79 GB, while ResiPipe showed the highest memory usage, reaching 1.86 GB.

5.3.3 BZip2 Compression

BZip2 is a widely used, freely available, and patent-free data compressor that uses the Burrows-Wheeler block sorting text compression algorithm and the Huffman coding algorithm [63]. Our implementation of the BZip2 compression application is based on the implementation made by the MPR work [46]. In this application, the source operator reads an input file from the disk (we used a 532Mb text file for the experiment), splits its content into smaller data blocks of 900Kb, and forwards the blocks to the middle operator. The middle operator (with replication) receives the input blocks, runs the compression algorithm, and forwards the compressed data block to the sink operator. Finally, the sink operator receives the compressed data blocks in an ordered manner, assembles the final compressed file, and writes its content over the disk.

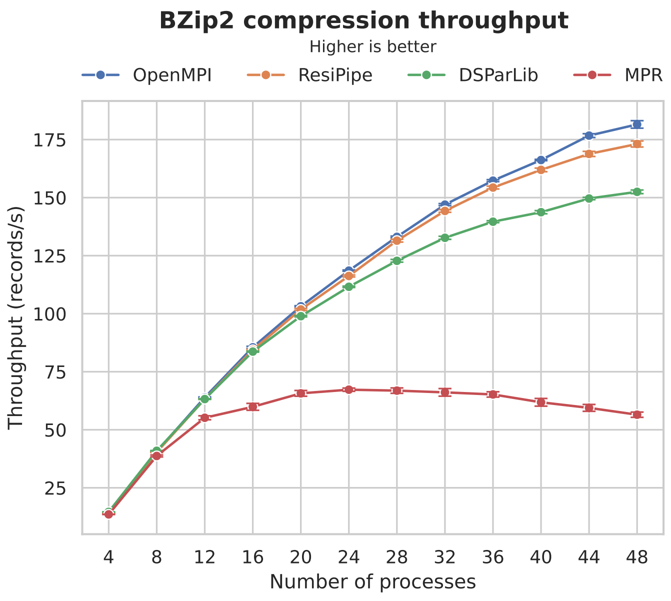


Figure 5.6: BZip2 compression throughput

Figure 5.6 presents the throughput of the BZip2 compression application, ranging from 4 to 48 parallel processes. OpenMPI, ResiPipe, and DSParLib presented very similar results of 14.15, 14.05, and 14.62 records per seconds with 4 processes, to 181.52, 173.09, and 152.49, respectively, with 48 processes. MPR could not efficiently scale after 16 parallel processes, presenting a throughput of 13.55 records per second using 4 processes and 56.49 using 48 processes.

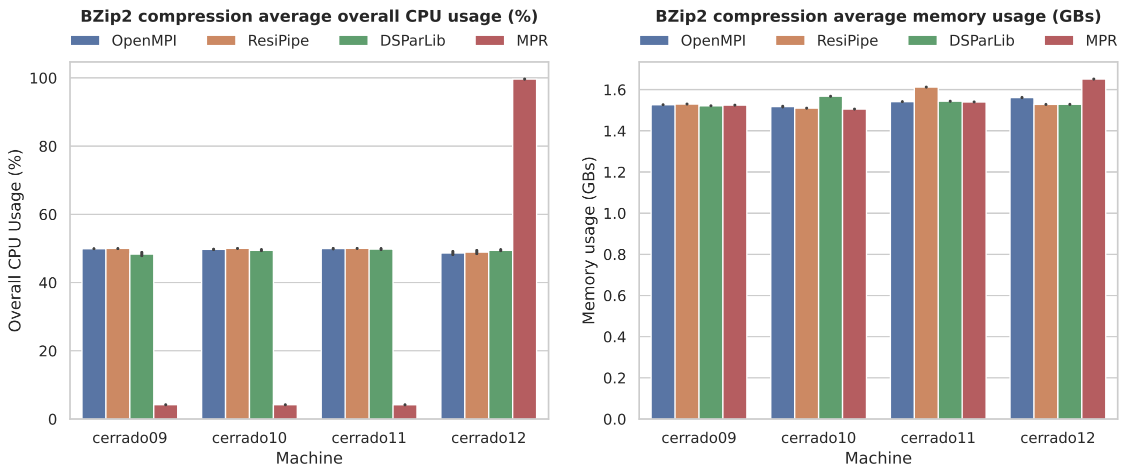


Figure 5.7: BZip2 compression average CPU and Memory usage

Figure 5.7 depicts the average CPU and memory usage across the cluster machines for the BZip2 compression application with 48 processes. DSParLib, OpenMPI, and ResiPipe presented a similar CPU utilization of around 50% across the nodes. DSParLib presented the lowest memory consumption, ranging from 1.69 to 1.95 GB. OpenMPI consumed around 1.86 to 1.92 GB of memory. ResiPipe presented the highest memory consumption, ranging from 1.89 to 1.92 GB across the nodes. MPR grouped all the working processes into a single node, reaching a peak of 100% CPU utilization and 4.09 GB of memory over a single node.

5.3.4 BZip2 Decompression

The BZip2 decompression application consists of the same application described in Section 5.3.3, using the decompression algorithm instead of compression. In this application, the source operator reads a compressed input file (we used the same output file generated by the compression application, which has 75Mb), splits the input file into smaller data blocks by searching for the BZip2 block headers (magic number) inside the file content, and forwards the data blocks to the middle operator. The middle operator (with replication) receives the compressed data blocks, runs the decompression algorithm, and forwards the decompressed data blocks to the sink operator. Finally, the sink receives the decompressed data blocks in an ordered manner, assembles the final file, and writes its content over the disk.

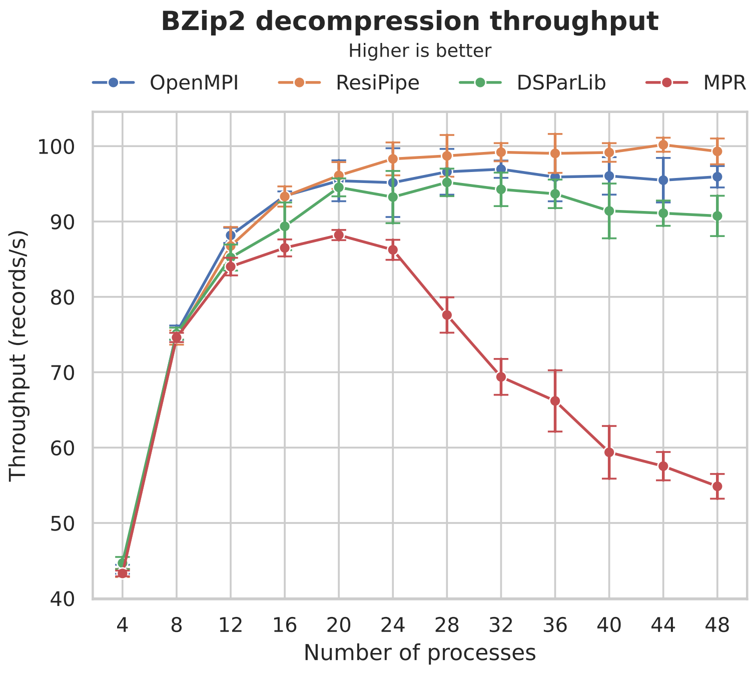


Figure 5.8: BZip2 decompression throughput

Figure 5.8 presents a comparative analysis of the throughput of the BZip2 decompression application, ranging from 4 to 48 parallel processes. MPR could not efficiently scale after 20 processes and presented the lowest throughput across all the variations. ResiPipe presented the best scalability across the implementations, reaching a throughput of 100.18 records per seconds using 44 processes. OpenMPI and DSParLib have also efficiently scaled according to the increase in the number of processes, presenting a throughput of 95.94 and 90.74 records per seconds, respectively, using 48 processes.

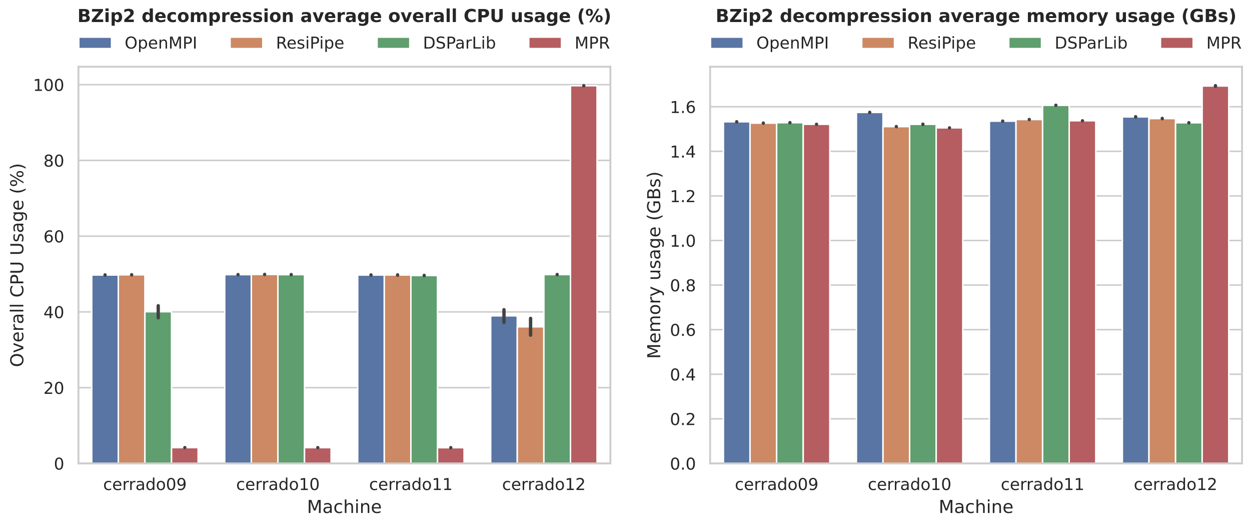


Figure 5.9: BZip2 decompression average CPU and Memory usage

Figure 5.9 depicts the average resource utilization (CPU and memory) across the cluster machines for the BZip2 decompression application using 48 parallel processes. DSParLib, OpenMPI and ResiPipe presented a similar CPU utilization of around 50% over the nodes cerrado10 and cerrado11. DSParLib presented a reduced CPU utilization of 40.05% over the node cerrado09, while OpenMPI and ResiPipe also presented a reduced CPU utilization of around 38.96% and 36.06% respectively over node cerrado12. DSParLib presented the lowest memory consumption, ranging from 1.68 to 1.87 GB. OpenMPI presented a memory consumption ranging from 1.79 to 1.87 GB. ResiPipe achieved the highest memory usage, from 1.86 to 1.90 GB. MPR grouped all the working processes over a single node, reaching a peak of 100% CPU usage and 4.07 GB of memory.

5.3.5 Eye Detector

Eye Detector is an application that performs face and eye recognition in a video. It was inspired by the Eye Detector application presented in the work of Piper et al. [55]. The application consists of a pipeline where the source operator reads a video file, transforms each video frame into an OpenCV matrix, links each matrix to a number that identifies the frame's relative position within the video, and sends each matrix to be processed by the middle operator. Upon receiving an input, the middle operator uses a pre-trained machine learning model (Haar Cascade Classifier) provided by OpenCV to detect faces in the input frame. If any face is detected, a second classifier is used to detect the eye's position within the face. The middle operator outputs the input frame with transparent boxes with blue borders drawn around the detected faces and eyes. The sink operator is responsible for receiving the frames processed by the middle operator, reassembling the frames in order, and saving the output video. Differently from the implementation presented in the work of Piper et al. [55], in which face detection and eye detection are separated into two stages, we choose to implement both the face detection and eye detection in a single stage since MPR does not support pipelines with multiple middle stages.

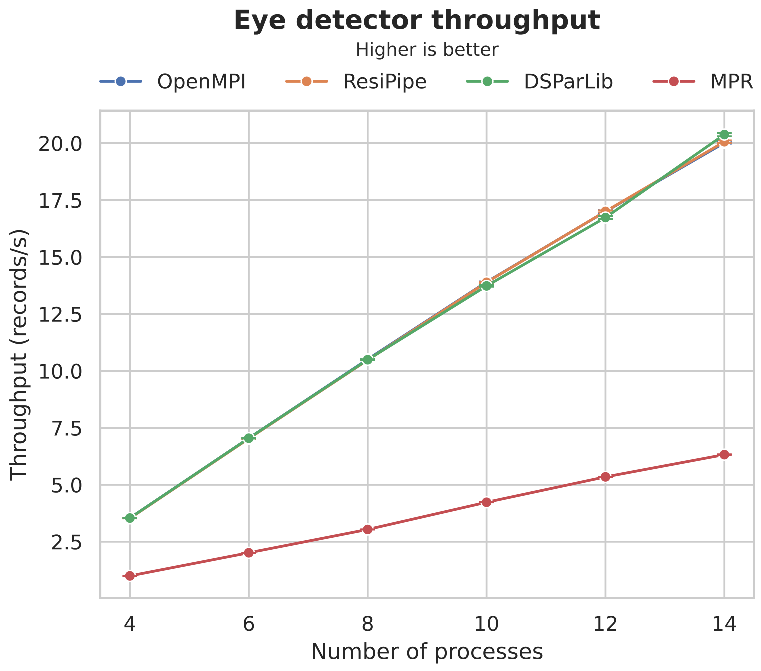


Figure 5.10: Eye detection throughput

Figure 5.10 presents the throughput (records per second) of the eye detector application, ranging from 4 to 14 processes. Unlike the other applications, this setup distributes just one process per node and uses the OpenCV parallel capabilities to spawn several threads to process the images. OpenCV has been configured to use the maximum concurrency level (number of parallel threads) supported by the CPU. In this application, OpenMPI, ResiPipe, and DSParLib presented virtually identical results, reaching a throughput of 3.53, 3.52, and 3.53 respectively with 4 processes, and 20.01, 20.06, and 20.37 respectively with 14 processes. MPR presented the lowest throughput among the parallel interfaces of 0.99 records per seconds with 4 processes and 6.32 with 14 processes.

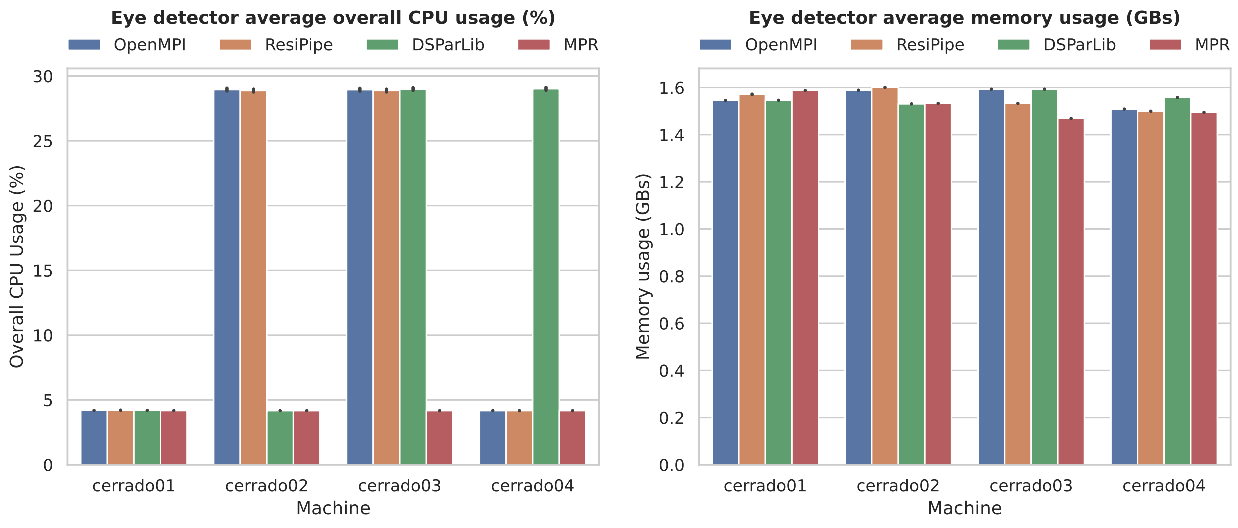


Figure 5.11: Eye detection average CPU and Memory usage

Figure 5.11 overviews the average memory and CPU usage of the Eye Detector application over 4 distinct nodes. OpenMPI, ResiPipe, and DSParLib presented a similar 29% CPU usage over the worker nodes and around 4.20% over the source and sink nodes. The CPU usage also indicates that OpenMPI and ResiPipe used a similar allocation strategy, using the nodes cerado02 and cerrado03 as worker nodes and cerrado01 and cerrado04 as source and sink nodes. DSParLib presented a different allocation strategy, using the nodes cerrado03 and cerrado04 as worker nodes and cerrado01 and cerrado02 as source and sink nodes. MPR presented a CPU usage of around 4.20% over all the nodes, indicating that some bottleneck is preventing MPR from achieving a higher processing rate. Regarding memory consumption, all libraries presented similar results. MPR presented the lowest memory consumption, ranging between 1.47 and 1.59 GB. OpenMPI consumed around 1.51 and 1.59 GB. DSParLib consumed around 1.53 and 1.59 GB. And ResiPipe presented the highest memory consumption, ranging from 1.50 to 1.60 GB.

5.3.6 Sentiment Analysis

Sentiment Analysis is a conceptual application developed in this work to evaluate the support for pipelines with multiple middle operators in ResiPipe and DSParLib, as the other abstractions do not provide support for such pipeline implementation. This application transcribes text from audio using the OpenAI Whisper model[2] and performs sentiment analysis over the resulting texts using the AFINN-111 lexicon dataset [75]. This dataset classifies the English words as positive or negative. As illustrated in Figure 5.12, the application is a four-stage pipeline in which the source operator reads four audio files with approximately one minute of conversation and forwards the binary data to the next

[2]https://openai.com/index/whisper/

stage. The first processing stage is replicated with two instances. Each instance loads the whisper model at startup using the `whisper.cpp` library[3], transcribes the text for each received input and forwards the result to the next stage. The second processing stage receives the transcription, runs the sentiment analysis algorithm, and sends the results to the sink operator. Finally, the sink operator receives the transcriptions along with the sentiment analysis and prints the results over the default output console.

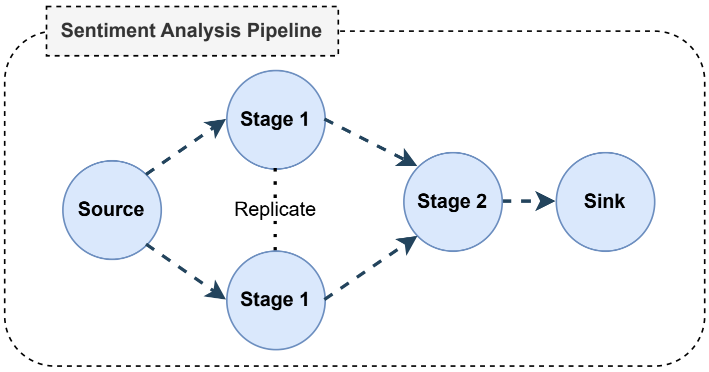


Figure 5.12: Sentiment Analysis pipeline graph

Table 5.1 presents a comparative analysis of the sentiment analysis throughput between DSParLib and ResiPipe. DSParLib presented a slightly higher throughput of 0.005025 records per seconds in comparison to ResiPipe with reach a throughput of 0.005008. The results indicate the extra operator (multiple-stage pipelines in DSParLib are composed of two 3-stage pipelines interconnected by a bypass operator) does not introduce a significant overhead over the DSParLib execution, or the optimizations employed by DSParLib compensate the overhead.

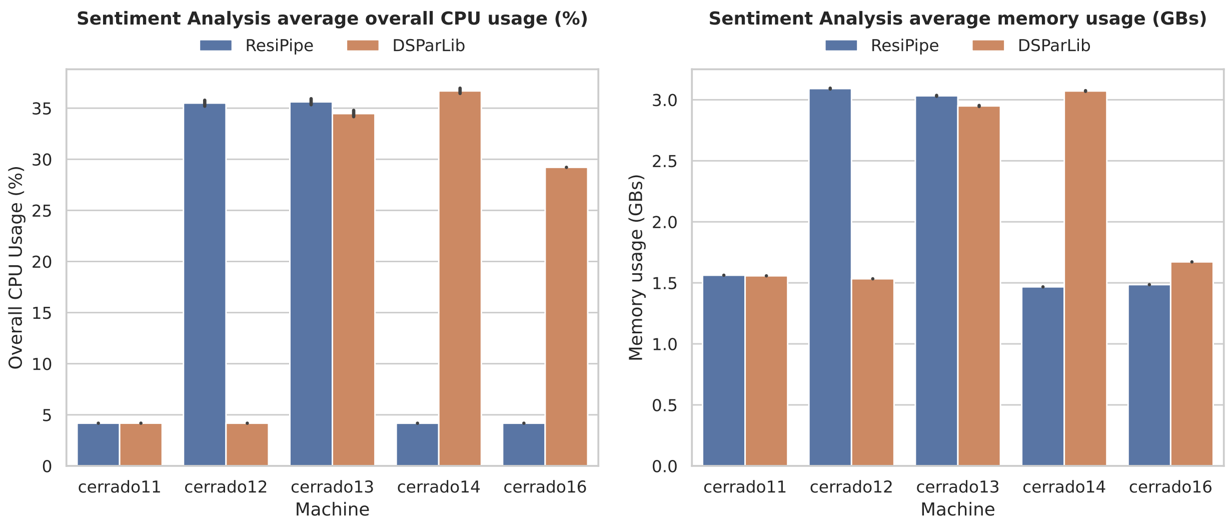


Figure 5.13: Sentiment Analysis average CPU and Memory usage

[3]https://github.com/ggerganov/whisper.cpp

Figure 5.13 presents the average memory and CPU usage of the sentiment analysis application across the cluster machines. ResiPipe and DSParLib presented similar results in both memory consumption and CPU usage. DSParLib has a slightly higher CPU usage, with a peak of 36.69%, while ResiPipe presented a peak CPU usage of 35.62%. Regarding memory consumption, ResiPipe presented a slightly higher consumption with a peak of 3.09 GB, while DSParLib presented a peak memory consumption of 3.07 GB. The metrics indicate that ResiPipe and DSParLib used a different scheme to distribute the processes across the cluster machines, with ResiPipe presenting a higher CPU usage across the nodes cerrado12 and cerrado13. In comparison, DSParLib presents a higher CPU usage across the nodes cerrado13, cerrado14, and cerrado16.

Table 5.1 summarizes the best throughput for each experiment according to the number of parallel processes. The table highlights in bold text the implementation that presented the best throughput for each application.

Table 5.1: Performance results summary

Experiment	Implementation	Best throughput (records/s)	Number of processes
Prime numbers	**DSParLib**	**247084.18**	**48**
	MPR	136354.10	24
	OpenMPI	232386.92	48
	ResiPipe	210283.26	44
Mandelbrot	DSParLib	1132.59	44
	MPR	895.30	24
	OpenMPI	1207.91	48
	ResiPipe	**1548.19**	**44**
BZip2 Compression	DSParLib	152.49	48
	MPR	67.27	24
	OpenMPI	**181.52**	**48**
	ResiPipe	173.09	48
BZip2 Decompression	DSParLib	95.19	28
	MPR	88.20	20
	OpenMPI	96.94	32
	ResiPipe	**100.18**	**44**
Eye detector	**DSParLib**	**20.37**	**14**
	MPR	6.32	14
	OpenMPI	20.01	14
	ResiPipe	**20.06**	14
Sentiment Analysis	**DSParLib**	**0.005025**	**5**
	ResiPipe	0.005008	5

5.4 ResiPipe failure recovery evaluation

This section presents a series of experiments specifically designed to evaluate the fault tolerance capabilities of ResiPipe, as well as the overhead introduced by the snapshot mechanism during regular execution. Four distinct use cases were developed to evaluate various aspects of stream processing applications. These include simple linear pipelines without replication, pipelines with replication and unbalanced workloads, pipelines experiencing multiple failures across different processing stages, and pipelines with large state sizes. The experiments for each use case were conducted with snapshot intervals of 15, 30, 45, and 60 seconds, in addition to a failure-free execution and a snapshot-disabled execution. The Spot library, detailed in Section 5.2, was used throughout all experiments to collect metrics such as the application throughput.

5.4.1 Use Case 1

Use case 1 is an experiment developed to evaluate the failure recovery capabilities by creating a synthetic failure over a stateful operator. As depicted in Figure 5.14, use case 1 is a four-stage linear pipeline in which the source operator (P0) produces 100 integer numbers. The first processing stage (P1) calculates the Fibonacci sequence for the given input and sleeps for a second to simulate a heavy computation. The second processing stage (P2) is a stateful operator that aggregates 25 items and throws a synthetic failure after processing 50 items. The sink operator (P3) receives the results and prints the values over the output console. This experiment aims to verify ResiPipe's ability to recover a stateful operator after a failure and the overhead introduced by the snapshot mechanisms over the system using different snapshot intervals.

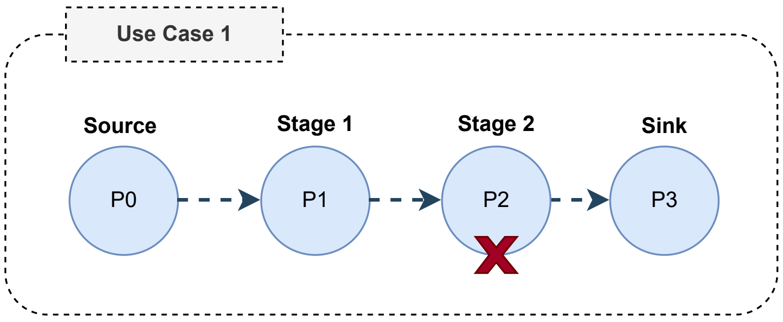


Figure 5.14: Use Case 1 Graph

Figure 5.15 provides a comparative analysis of the snapshot impact over a failure-free execution of Use Case 1 using snapshot intervals of 15, 20, 45, and 60 seconds. It also presents the application throughput in an failure-prone execution. The results indicate that the snapshot mechanism does not impose a significant impact on the regular execution of Use Case 1, independently of the evaluated snapshot interval.

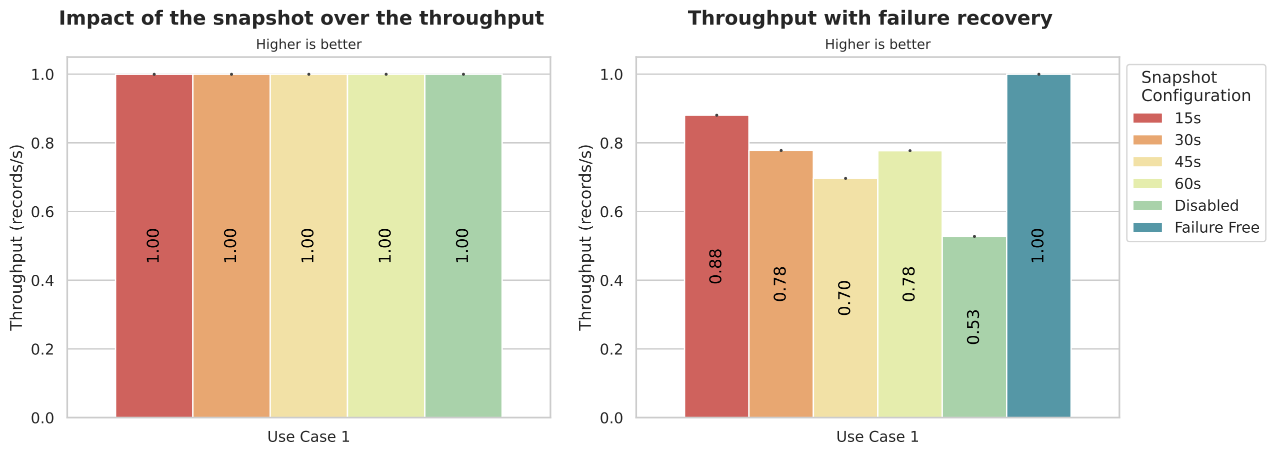


Figure 5.15: Use Case 1 throughput and failure recovery

Regarding failure recovery, the results demonstrate that independent of the frequency in which the snapshots are taken, restarting the application from a snapshot in a failure scenario is way faster than performing a cold restart (restarting the application from scratch). However, the results also demonstrate that a higher snapshot frequency is not necessarily associated with a faster recovery time. In this specific scenario, it appears that the best snapshot frequencies are every 15, 30 and 60 seconds.

5.4.2 Use Case 2

Use case 2 aims to verify the recovery capabilities for pipelines with stage replication and unbalanced workloads. Figure 5.16 presents the pipeline graph of the use case 2 application, in which a source operator (P0) produces a sequence of 100 integer numbers. The first processing stage has two replicas (P1 and P2) that calculate the Fibonacci sequence for the input value and sleep for one second to simulate a heavy computation. The second replica (P2) has an additional 300-millisecond delay to simulate an unbalanced workload and throws a synthetic failure after processing 50 items. The second processing stage (P3) is a stateful operator that performs an aggregation of 25 items. The sink operator (P4) receives the results and prints the values over the standard output console.

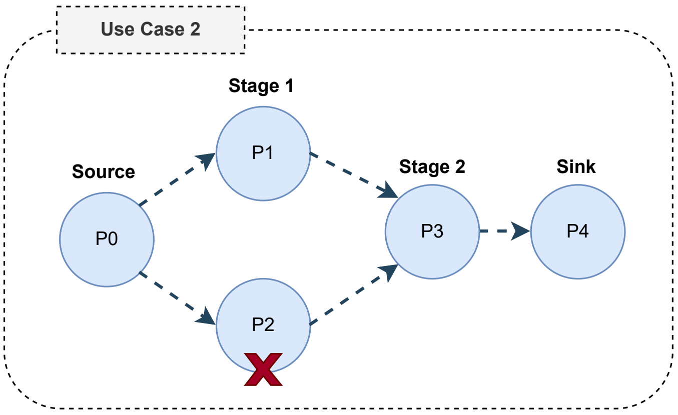


Figure 5.16: Use Case 2 Graph

Figure 5.17 presents the impact of the snapshot over Use Case 2 in a failure-free and failure-prone scenario. The results indicate that, independent of the evaluated snapshot interval, the snapshot mechanisms do not introduce a significant overhead over the application throughput, even when using a replicated stage and an unbalanced workload.

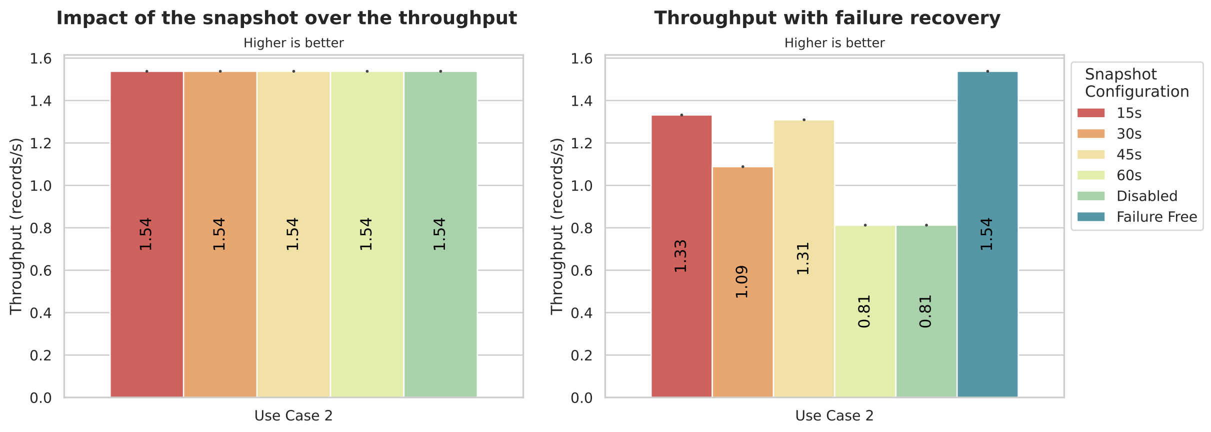


Figure 5.17: Use Case 2 throughput and failure recovery

Regarding the Use Case 2 failure recovery results, the 15-second snapshot interval presented the best recovery time across the variations, exhibiting a throughput of 1.33 records per second, 13.64% slower than the failure-free execution. The 45-second snapshot interval presented the second-best recovery time, exhibiting a throughput of 1.31 records per second. The 30-second snapshot interval presented a considerably slower recovery time, achieving a throughput of 1.09 records per second. The 60-second snapshot interval was ineffective in this use case, taking the same time as the snapshot disabled (cold restart) execution. This behavior happens because the total execution time of a failure-free execution (65.03 seconds) is too close to the 60-second snapshot interval, so the snapshot mechanism does not have enough time to complete a snapshot before the failure happens.

5.4.3 Use Case 3

Use case 3 aims to verify the recovery capabilities for pipelines with multiple failures across different processing stages. Figure 5.18 presents the use case 3 pipeline graph, in which a source operator (P0) produces a sequence of 100 integer numbers. The first processing stage has two replicas (P1 and P2) that calculate the Fibonacci sequence for the input values and sleep for one second to simulate a heavy computation. The second replica (P2) has an additional 300-millisecond delay to simulate an unbalanced workload and throws a synthetic failure after processing 26 items. The second processing stage also has two replicas (P3 and P4). Similarly to the first processing stage, it calculates the Fibonacci sequence for the input value and sleeps for one second to simulate a heavy computation. In the second processing stage, the first replica (P3) has an additional 300-millisecond delay and fails after processing 51 items. In this application, the operators exchange messages using an even/odd affinity scheme based on the message identifier, in which the messages with odd identifiers are always sent to the first replica of the stage (P1 and P3), and the messages with even identifiers always are sent to the second replica of the stage (P2 and P4). The third processing stage (P5) is a stateful operator that performs an aggregation of 25 items. Finally, the sink operator (P6) receives the results and prints the values over the standard output console.

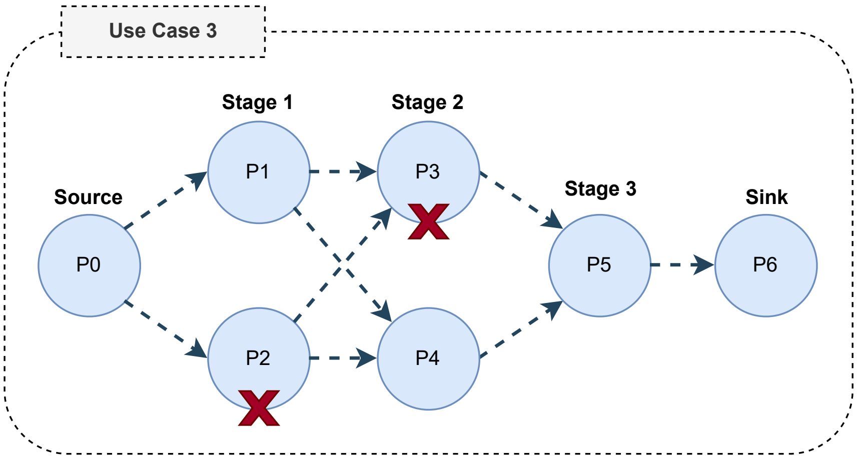


Figure 5.18: Use Case 3 Graph

Figure 5.19 presents a comparative analysis of the Use Case 3 throughput in a failure-free scenario and a failure-prone scenario with varied snapshot intervals. The results demonstrate that the regular throughput of Use Case 3 was not affected by the snapshot mechanism, independent of the snapshot interval variation. This result indicates that the snapshot mechanism implemented by ResiPipe can efficiently handle complex pipeline graphs in which multiple operators need to perform an alignment phase (P3, P4, and P5) without introducing major overheads during regular execution.

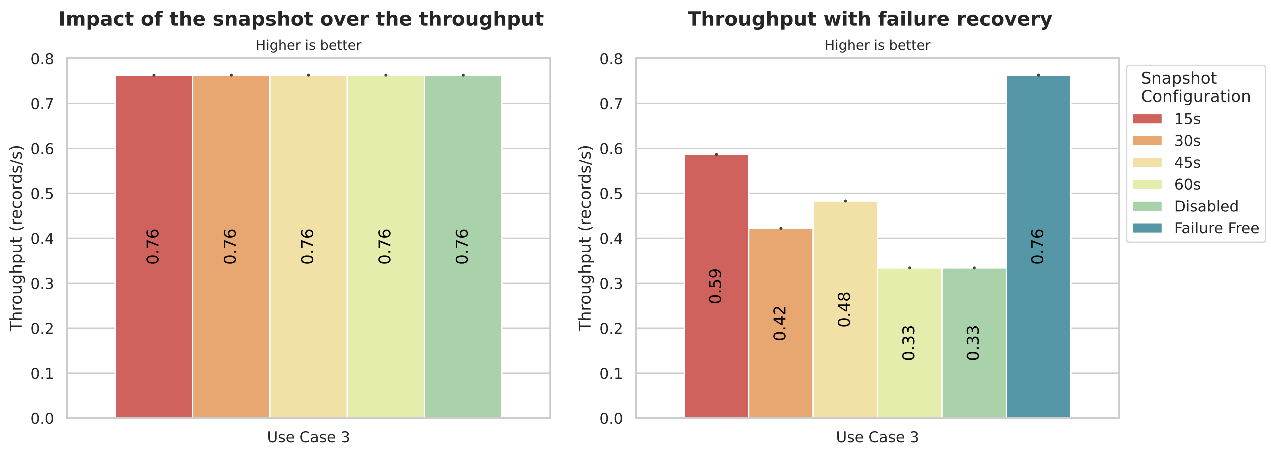


Figure 5.19: Use Case 3 throughput and failure recovery

The Use Case 3 failure recovery results demonstrate that even with a 15-second snapshot interval, the system exhibits a throughput of 0.59 records per second in the failure-prone scenario, which is 22.37% slower than the failure-free execution. The 30-second and 45-second snapshot intervals presented a significantly longer recovery time, exhibiting a throughput of 0.42 and 0.48 records per second, respectively. The 60-second snapshot interval was ineffective, taking the same time to finish the execution as the snapshot disabled (cold-restart) execution. This behavior happens because the 60-second snapshot interval is so long that the system does not have enough time to complete a snapshot before a failure happens between operators P2 and P3, generating a cold restart of the application.

5.4.4 Use Case 4

Use case 4 is a simple 3-stage pipeline designed to verify the impact of the snapshot mechanism over the throughput of pipelines containing stateful operators with a large state size. Figure 5.20 illustrates the pipeline graph of the use case 4 application, in which a source operator (P0) produces a sequence of 100 integer numbers. The middle stage operator (P1) does not apply any computation over the input; it just holds a 100 MB static state to simulate a large state size. The sink operator (P2) receives the results and prints the values over the standard output console. This use case does not simulate failures since its primary goal is to evaluate how a large state size affects the application throughput across different snapshot intervals.

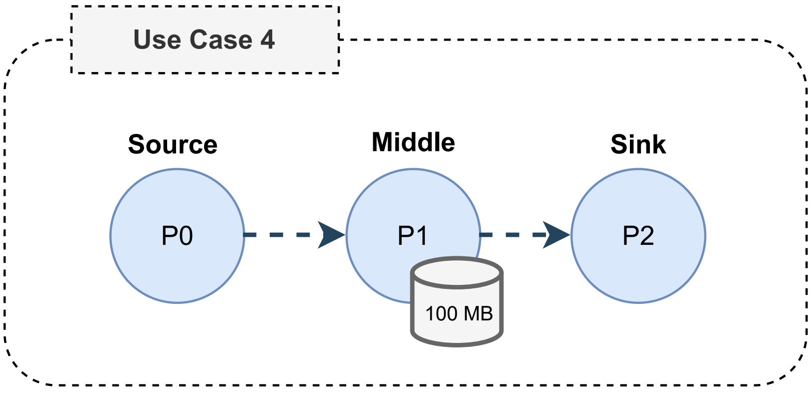


Figure 5.20: Use Case 4 Graph

Figure 5.21 provides a comparative analysis of the throughput for Use Case 4 in a failure-free scenario with varied snapshot intervals. The results indicate that for a pipeline containing a single operator with 100 MB of persistent state, the fault tolerance mechanism introduces an overhead of 8.17% when using a 15-second snapshot interval.

When using snapshot intervals of 30, 45, and 60 seconds, the overhead decreases to 4.26%, 6.25%, and 7.22%, respectively.

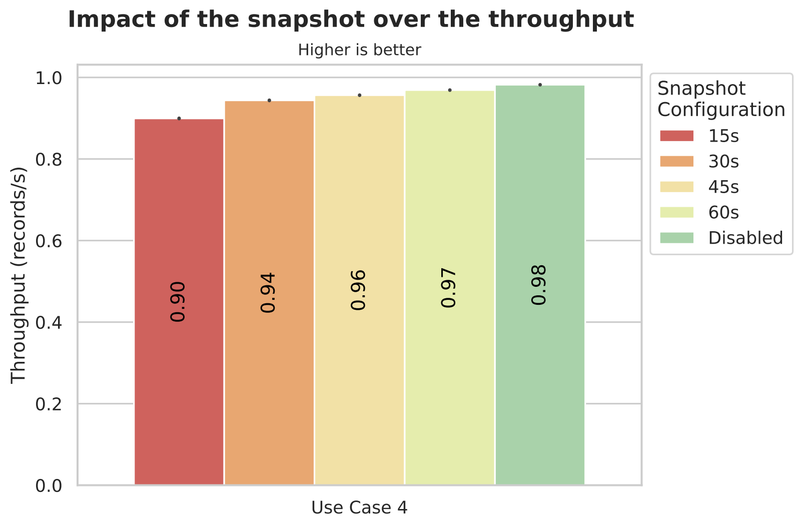


Figure 5.21: Use Case 4 throughput evaluation

These results indicate that the snapshot mechanism employed by ResiPipe is sensitive to large state sizes, presenting a decrease in the application's throughput as snapshot frequency increases. This behavior happens because ResiPipe implements a synchronous snapshot mechanism, requiring operators to pause regular processing to store the snapshot files over reliable storage. Consequently, this synchronous approach introduces significant I/O overhead, which scales according to the state size.

5.5 Programmability evaluation

This section presents an evaluation regarding the programmability aspects of ResiPipe in comparison to DSParLib, MPR, MPI, and SPar, including metrics such as the number of source lines of code (SLOC) and the estimated development time using Halstead's method. Table 5.2 exhibits a comparative analysis of the number of source lines of code for each application described in section 5.3. The table highlights the lowest values using bold text. The results indicate that SPar requires fewer lines of source code (SLOC) across the analyzed programming interfaces. However, this is already expected since SPar is a domain-specific language with a higher abstraction level than the remaining analyzed libraries. Removing SPar from the analysis, ResiPipe presented the lowest number of lines of code for most applications, except for the BZip2 compression, in which ResiPipe and DSParLib presented the same number of lines.

Table 5.2: SLOC Evaluation

Framework	SLOC (Source Lines of Code)					
	Prime Numbers	Mandelbrot	Eye Detector	BZip2 Decompression	BZip2 Compression	Sentiment Analysis
SPar	**25**	**56**	**64**	**104**	**72**	-
ResiPipe	60	130	106	133	106	**120**
DSParLib	65	157	146	139	106	196
MPR	74	140	138	164	131	-
MPI	119	157	154	204	171	-

The number of source lines of code required to implement the applications varies between DSParLib and MPR. While DSParLib presents the lowest number of lines for the application prime numbers, BZip2 decompression, and BZip2 compression, MPR presents the lowest number of lines for the applications mandelbrot and eye detector. MPI presents the highest number of lines for all the applications. However, this was already expected since MPI primarily abstracts process communication. In contrast, the other analyzed programming interfaces also abstract the parallel patterns and data serialization.

Table 5.3 presents a comparative analysis regarding the estimated development time in hours of each application described in section 5.3. The PHalstead (Parallel Halstead) tool developed by Andrade et al. [7] has been used to estimate the development time of each application. The table highlights the lowest values using bold text. The results indicate that SPar requires the lowest development time across the analyzed programming interfaces. However, this is already expected since SPar is a domain-specific language with a higher abstraction level than the remaining analyzed libraries. Removing SPar from the analysis, the results indicate that for most of the applications (mandelbrot, eye detector, BZip2 compression, decompression, and sentiment analysis), ResiPipe requires the lowest development time, except for the prime numbers application, in which DSParLib presented the lowest development time.

Table 5.3: Halstead Evaluation

Framework	Halstead (development time in hours)					
	Prime Numbers	Mandelbrot	Eye Detector	BZip2 Decompression	BZip2 Compression	Sentiment Analysis
SPar	**0.67**	**2.61**	**1.63**	**6.12**	**2.82**	-
ResiPipe	1.53	6.43	3.46	6.29	3.41	**6.52**
DSParLib	1.39	10.27	7.39	6.94	3.49	14.50
MPR	2.10	7.09	6.77	10.19	5.90	-
MPI	8.33	13.04	11.61	18.44	13.46	-

The development time between DSParLib and MPR also varies, with DSParLib presenting the lowest development time for the applications: prime numbers, BZip2 compression, and BZip2 decompression. Meanwhile, MPR presents the lowest development time for the applications mandelbrot and eye detector. MPI presents the longest development time across all the analyzed programming interfaces, taking more than 6x of the time required to implement the prime numbers application using DSParLib, and more than 5x the time to implement the same application using ResiPipe.

From a qualitative perspective, although MPR requires fewer lines of code than MPI, it is the most challenging programming interface since, until the writing of this thesis, it does not provide any documentation regarding its usage other than two examples (mandelbrot and prime numbers) in its official GitHub repository[4]. Additionally, the MPR API presents confusing methods and functions. For instance, in the source operator class, the developer must call the *Produce* or *ProduceMulti* methods to send an item to the next processing stage. For the remaining processing stages, the developer must call the *Publish* or *PublishMulti* methods. However, the *Publish* and *PublishMulti* methods also exist in the source operator class; they are just not implemented, making the application stall if the developer makes a mistake and calls the wrong method. DSParLib also does not provide extensive documentation; however, its API is way more consistent. While MPI, despite exhibiting the highest number of lines and the longest development time, benefits from decades of robust documentation and widely available examples.

5.6 Final remarks

This chapter presented a set of six performance experiments with varied computational characteristics, a set of four failure recovery experiments that evaluate the ResiPipe fault tolerance capabilities and the impact of its fault tolerance mechanism over regular execution, and a programmability evaluation of ResiPipe, DSParLIB, MPR, and MPI using SLOC and Halstead's metrics. The performance and failure-recovery experiments were validated by comparing an MD5 hash of the output generated by the parallel implementations with an MD5 hash of the output produced by their equivalent sequential programs to ensure the correctness of the results.

Overall, the ResiPipe library presented similar performance results and, in some cases (Mandelbrot and BZip2 decompression), as good as those of the remaining analyzed libraries while providing fault tolerance, out-of-the-box data serialization, and support for pipelines with multiple processing stages, a set of features that together are unseen over the remaining analyzed C++ stream processing libraries. Regarding resource utilization, ResiPipe, DSParLib, and OpenMPI presented similar CPU usage. However, ResiPipe presented the highest memory consumption across the experiments (disregarding MPR, which grouped all processes over a single machine), and DSParLib presented the

[4]https://github.com/GMAP/MPR/tree/main

lowest memory consumption in most of the experiments (except for the eye detector application).

MPR struggled to scale efficiently with the increasing number of parallel processes across all experiments. The cluster resource utilization metrics indicate that this issue happens due to the load-balancing scheme employed by MPR, which, by default, allocates all processes to the same machine until it reaches the limit of parallel MPI processes instead of distributing the processes evenly across the cluster nodes. While it is possible to change this behavior by manually specifying the maximum number of processes per machine in the MPI configurations, fine-tuning these settings for each variation of every experiment for MPR would be unfair to the other libraries. For this reason, all libraries were tested using the same default MPI configurations set by the cluster, and the results exhibit the performance that each library reaches using its out-of-the-box configurations.

Regarding fault tolerance, the experiments indicate that ResiPipe can efficiently handle complex pipelines in which multiple operators must perform a snapshot alignment phase (use case 3) without introducing significant overheads during regular execution. Independently of the snapshot interval, the snapshot mechanisms do not introduce a significant overhead over the throughput of the use cases 1, 2, and 3 since these applications have a relatively small state size. However, the snapshot mechanism employed by ResiPipe appears to be sensitive to large state sizes (use case 4), exhibiting a decrease in the application's throughput as snapshot frequency increases. This happens because ResiPipe persists the snapshot file over disk using a blocking operation, which makes the system susceptible to I/O overheads relative to the state size of the operator. The results also demonstrate that a higher snapshot frequency is not necessarily associated with a faster recovery time, with the 45-second snapshot interval presenting a faster recovery time than the 30-second interval for use cases 2 and 3.

Finally, the programmability evaluation demonstrates that SPar requires both fewer lines of code (SLOC) and less development time, according to the Halstead method, for all the applications. However, this is already expected since SPar is a domain-specific language with a higher abstraction level than the remaining analyzed libraries. Removing SPar from the analysis, the results indicate that ResiPipe requires fewer lines of code to implement all the applications and less development time for most of the applications (Mandelbrot, Eye Detector, BZip2 Decompression, BZip2 Compression, and Sentiment Analysis) when compared to the other evaluated abstractions. This happens because ResiPipe abstracts the process communication, parallel pattern, and data serialization, while the remaining analyzed libraries only abstract a subset of these features.

The number of lines and the development time between DSParLib and MPR varies. DSParLib presented the lowest development time for the applications: prime numbers, BZip2 compression, and BZip2 decompression. Meanwhile, MPR presented the lowest development time for the applications mandelbrot and eye detector. MPI presented the longest development and the highest number of lines across all the analyzed programming interfaces, taking more than 6x the time required to implement the prime numbers application using DSParLib, and more than 5x the time to implement the same application using ResiPipe. However, from a qualitative perspective, it is worth noting that MPR and DSParLib do not provide documentation, while MPI, despite exhibiting the highest number of lines and the longest development time, benefits from decades of robust documentation and widely available examples.

6. RESILIENT SPAR CODE GENERATION

The goal of this research involves exploring ways of introducing fault tolerance and exactly-once message delivery guarantees over the SPar software ecosystem. In the last chapter, ResiPipe was introduced as a C++ library designed to simplify the development of fault-tolerant distributed stream processing applications on clusters of commodity machines. This chapter describes how SPar was modified to generate ResiPipe code, as well as the limitations of its current implementation, and a performance evaluation regarding native ResiPipe code and SPar generated code.

6.1 Implementation

SPar is a domain-specific language that aims to simplify the development of stream processing applications by introducing high-level C++ annotations over sequential source code. These annotations are compiled (source-to-source) into an intermediate code that calls a parallel programming interface. To handle these C++ annotations, SPar provides its compiler based on CINCLE (Compiler Infrastructure for New C/C++ Language Extensions), which uses the GNU C++ compiler for the semantic analysis of the source code, and Flex and Bison for token extraction and the generation of the Abstract Syntax Tree (AST). The AST serves as input for the middle-end and back-end components of the compiler, which perform code transformations and ultimately generate the output binary.

Differently from GSParLib [59, 62, 60] and DSParLib [49, 56], this work implements ResiPipe code generation support over an experimental version of SPar built upon the CLang compiler front-end rather than the CINCLE compiler infrastructure. The SPar CLang project has been started as an effort to bring SPar to a commercial-grade compiler front-end since the CINCLE project was first developed as a prototype and does not have a dedicated team to keep the project updated. Currently, CINCLE is limited to the C++14 standard and operates under the GNU license. By adopting CLang, SPar gains support for newer C++ standard features and a license compatible with commercial software.

As depicted in Figure 6.1, the SPar CLang project comprises two major stages. In the first stage, the input source code is processed by a customized version of the CLang front-end, which has been modified to accept the SPar attributes as valid code. The CLang front-end performs the lexical and semantic analysis, parses the source code tokens, and generates an Abstract Syntax Tree (AST). In the second stage, the resulting AST is processed by an implementation of SPar developed using the LibTooling library, which is a library that eases the creation of standalone tools based on CLang. In this stage, a new semantic analysis step is performed to identify the SPar annotations inside the AST. The input and output variables of the SPar stages are identified and transformed into a *struct* that represents the data exchanged between stages, and the SPar stages are transformed into a pipeline for the target programming interface. Currently, the SPar CLang prototype project only supports ResiPipe and Intel TBB as targets for code generation. Finally, the stream item *struct* and the resulting pipeline are assembled into the output source code.

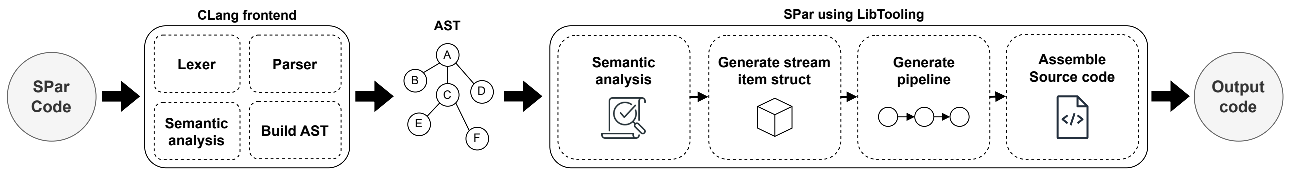


Figure 6.1: SPar code generation pipeline

Internally, the LibTooling stage of the SPar CLang project follows the same transformation rules as the original SPar work [30, 32] by implementing a set of CLang classes responsible for consuming the input AST. The first class, *SParFrontendAction*, is the entry point of the project and is responsible for calling the *SParConsumer* class, which consumes the AST by allowing the execution of recursive functions (also named *RecursiveASTVisitor*) that navigate through the AST. The SPar CLang project only implements visitors for function declarations (*VisitFunctionDecl*) and attributed statements (*VisitAttributedStmt*), which encompasses the SPar annotations. When the visitor identifies a *ToStream* annotation inside the AST, it captures the variables inside the *Input* and *Output* attributes and the definition of the first loop after the annotation. Then, a second visitor named *StageVisitor* recursively navigates the *ToStream* statement, searching for *Stage* annotations.

When the *StageVisitor* identifies a *Stage* statement inside the AST, it captures the variables inside the *Input*, *Output*, and *Replicate* attributes, and the block of code inside the stage scope. With this information, a validation is performed to verify if the *ToStream* has more than zero *Stages* and if the input variables of a stage always match the output variables of the previous stage. If the validation is successful, a function that generates code for the target runtime is called, and the block of code wrapped by the *ToStream* annotation is replaced by the resulting code. Otherwise, a compilation error is generated.

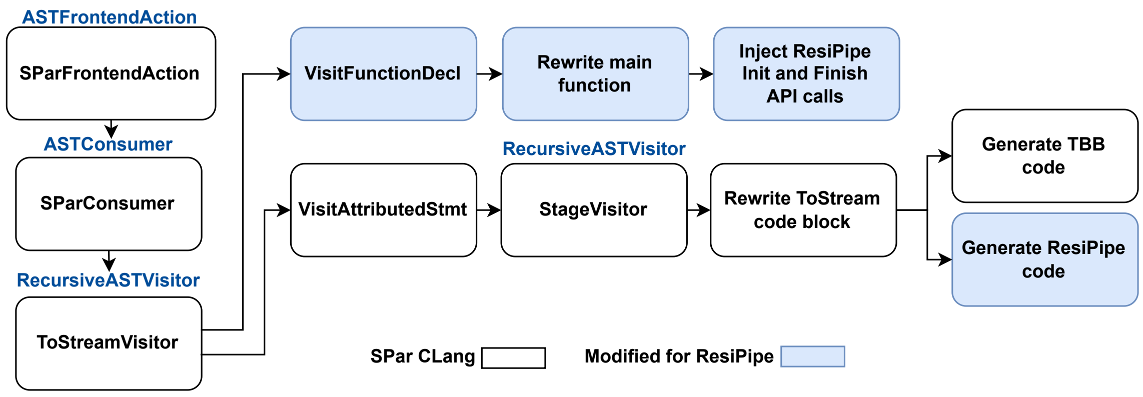
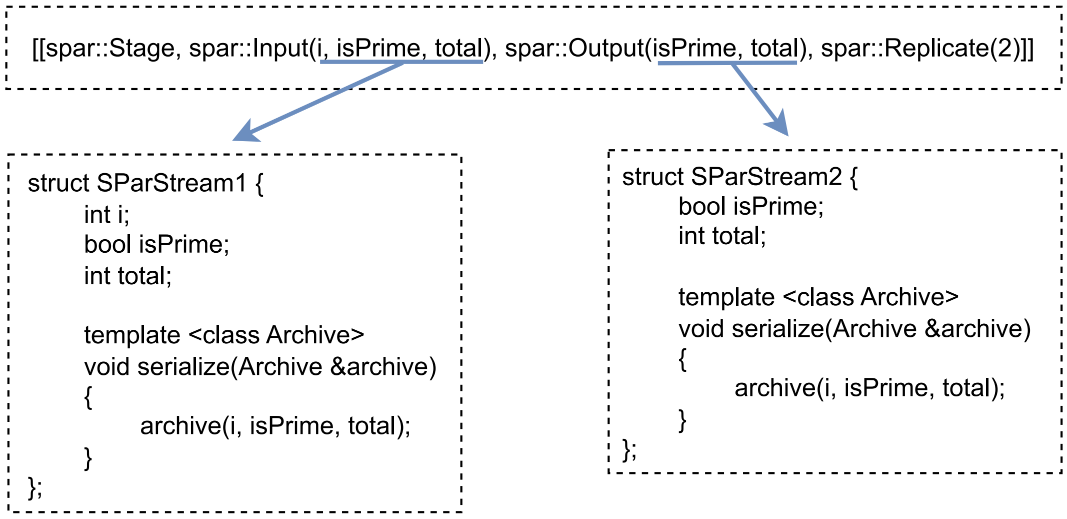


Figure 6.2: Modifications in SPar CLang for ResiPipe code generation

Figure 6.2 presents the components of the SPar CLang project that have been modified to support ResiPipe as a runtime for code generation. The modifications include changes at the *VisitFunctionDec* visitor to inject the API calls that initialize and finalize the ResiPipe environment at the *main* function of the input source code. The modifications also include implementing the code generation function that generates the ResiPipe operator classes and the *structs* representing the data exchanged between the stages based on the information extracted from the SPar annotations.

Figure 6.3 demonstrates an example of ResiPipe stream item *structs* generated from the stage *Input* and *Output* attributes. To generate this *structs*, the *VisitAttributed-Stmt* visitor captures information regarding the type and the name of the input and output variables of the stage from the AST. Then, this information is used inside the ResiPipe code generation function to create the stream item *structs* containing the same set of variable names and types in addition to the serialization function required by the Cereal library.


```
[[spar::Stage, spar::Input(i, isPrime, total), spar::Output(isPrime, total), spar::Replicate(2)]]
```

```
struct SParStream1 {
        int i;
        bool isPrime;
        int total;

        template <class Archive>
        void serialize(Archive &archive)
        {
                archive(i, isPrime, total);
        }
};
```

```
struct SParStream2 {
        bool isPrime;
        int total;

        template <class Archive>
        void serialize(Archive &archive)
        {
                archive(i, isPrime, total);
        }
};
```

Figure 6.3: Representation of the stream item structs

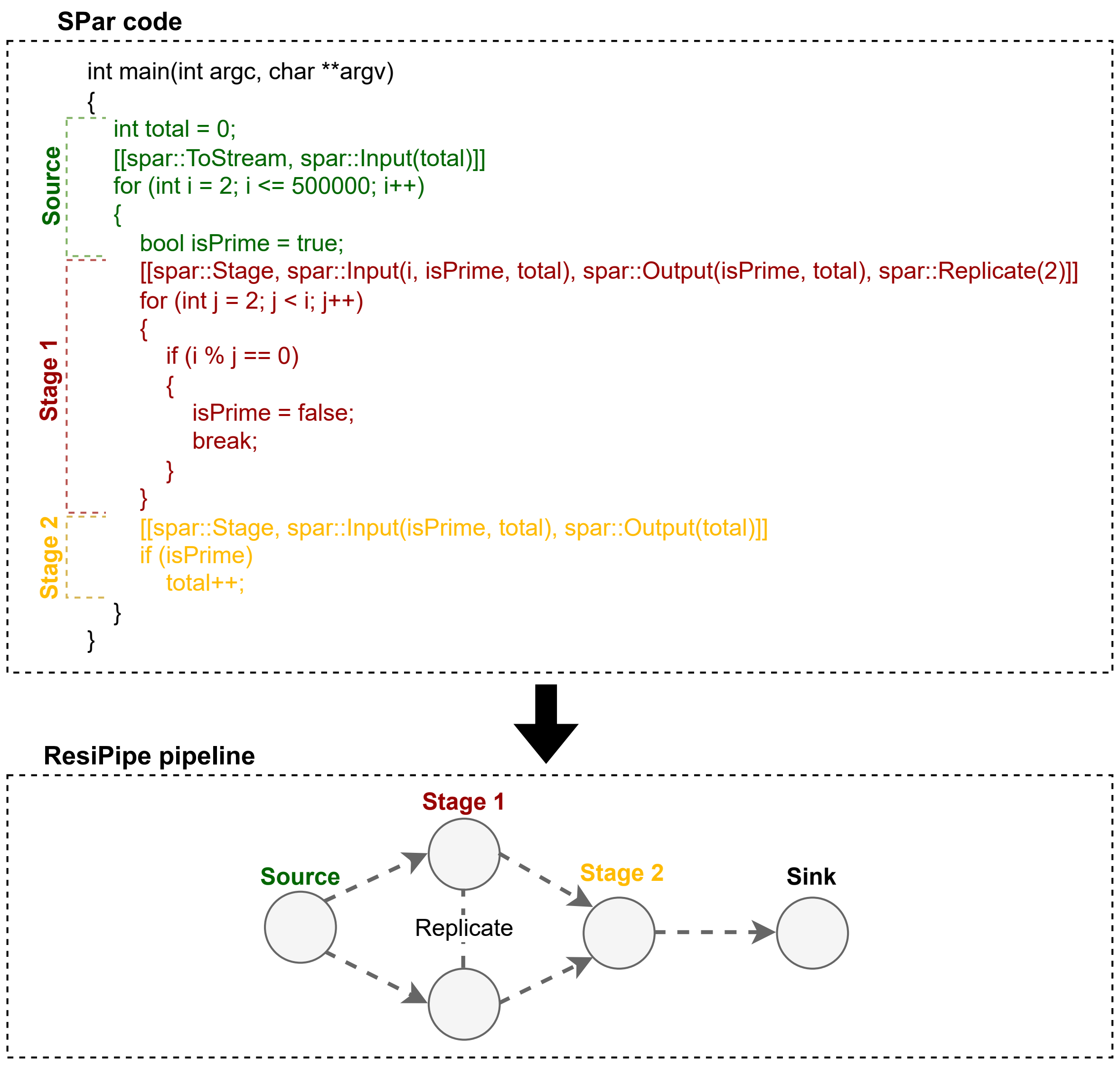
In summary, the logic behind the ResiPipe code generation function can be described by the following transformations over the input source code:

- The the first loop that follows the *spar::ToStream* annotation is wrapped inside a ResiPipe *ISourceOperator* class. The loop iterator is stored as a persistent state inside the ResiPipe key store, and in case of failure, the value of the iterator is restored from the latest snapshot. Any input variables at the *spar::ToStream* annotation are given as a reference on the class constructor. The input variables of the next *spar::Stage* are used as the output of the current stage. These variables are wrapped inside a *struct* that is serialized using the ResiPipe serialization function and forwarded to the next stage.

- The code inside the *spar::Stage* is wrapped inside a ResiPipe *IMiddleOperator* class. A new variable named *streamItem* is injected at the beginning of the stage code.

This variable receives the deserialized output *scruct* from the previous stage. All the input variables of the current stage are replaced by the values inside the *streamItem struct*. The input variables of the next stage are wrapped inside an output *struct* that is serialized and forwarded to the next stage. This pattern repeats for every *spar::Stage* annotation.

Figure 6.4 demonstrates a representation of a ResiPipe pipeline generated from SPar code. Currently, ResiPipe does not provide support for multiple sink operators, and since any non-ordered *spar::Stage* can be replicated, in the current implementation of ResiPipe code generation, an extra operator is included at the end of the pipeline. This operator only implements the ResiPipe *ISinkOperator* class and receives the outputs of the previous stage. This sink operator performs no computation over the input records and only exists to fulfill a ResiPipe limitation.



```
int main(int argc, char **argv)
{
    int total = 0;
    [[spar::ToStream, spar::Input(total)]]
    for (int i = 2; i <= 500000; i++)
    {
        bool isPrime = true;
        [[spar::Stage, spar::Input(i, isPrime, total), spar::Output(isPrime, total), spar::Replicate(2)]]
        for (int j = 2; j < i; j++)
        {
            if (i % j == 0)
            {
                isPrime = false;
                break;
            }
        }
        [[spar::Stage, spar::Input(isPrime, total), spar::Output(total)]]
        if (isPrime)
            total++;
    }
}
```

Figure 6.4: SPar code to ResiPipe pipeline representation

All the applications that SPar generates using ResiPipe as the target parallel interface benefit from the ResiPipe fault tolerance mechanism. In case of failure, the ResiPipe monitor agent automatically restarts the application. The position of the loop iterator that follows the *spar::ToStream* annotation is recovered from the latest completed snapshot, and the application continues its regular processing. Regarding message delivery guarantees, the SPar generated applications only provide at-most-once semantics by default. To achieve at-least-once semantics, the programmer must implement a replayable data source that resends the records in case of failure. To achieve exactly-once semantics, the programmer must implement a replayable data source and idempotent sinks.

As limitations, SPar does not provide a way to declare persistent states, change the default load balancing mode between operators, or implement the operator life cycle events *onInit*, *onExit*, *onError*, *onLocalSnapshotComplete*, and *onGlobalSnapshotComplete* provided by ResiPipe. This prevents the implementation of the two-step commit protocol required by ResiPipe to achieve exactly-once semantics. Additionally, exchanging raw pointers between processing stages is not supported. To enable this feature, the *spar::Input* attribute needs to be modified to accept the size of the pointer, similar to the adaptation made in [33, 56] works. Furthermore, the current implementation of ResiPipe code generation does not support multiple parallel regions within SPar code. This limitation happens because the presence of multiple *spar::ToStream* annotations results in multiple pipelines, and ResiPipe treats each pipeline as the entire application rather than just one parallel region within a sequential code structure.

6.2 Overhead evaluation

This section provides a comparative analysis regarding the throughput (records processed per second) between native ResiPipe applications and those generated by SPar. The evaluation uses the same set of performance experiments described in Section 5.3. The Sentiment Analysis application has been excluded from this comparison, as it was specifically designed to assess the capabilities of ResiPipe and DSParLib in supporting pipelines with multiple processing stages and due to development time constraints. All the experiments were executed 10 times to ensure the consistency of the results, and the fault tolerance mechanism was turned off over all the executions since the goal of these experiments is to verify the performance difference between native ResiPipe and SPar generated code, not the overhead of the fault tolerance mechanism, which was previously evaluated. The Spot library described in Section 5.2 was used to collect the throughput metrics.

Figure 6.5 presents an analysis of the throughput of the Prime Numbers application ranging from 4 to 48 parallel processes, in which it is possible to verify that the SPar generated application presents a lower throughput than the native ResiPipe application independently of the number of processes, and was unable to efficiently scale after 16 parallel processes. In contrast, the native ResiPipe application continues to scale according to the increase in the number of parallel processes.

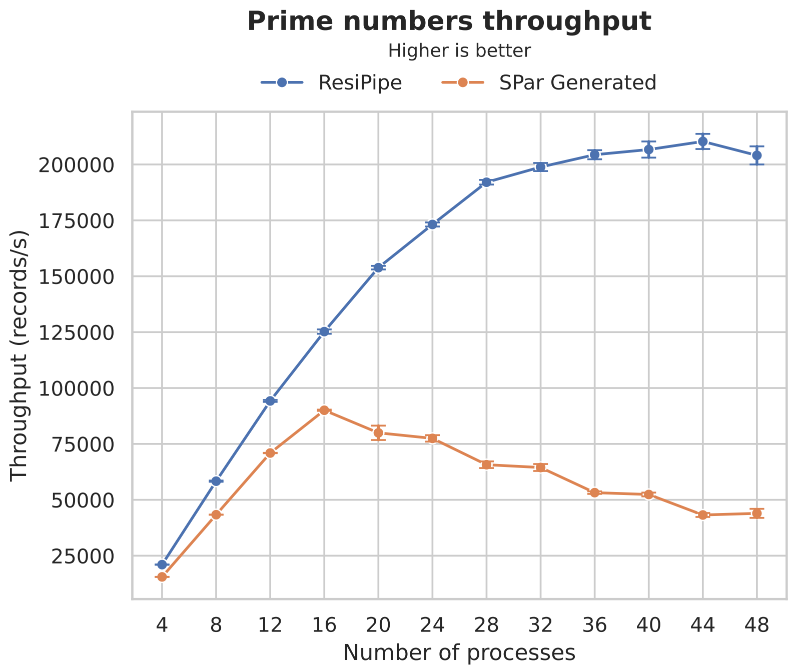


Figure 6.5: Performance comparison between ResiPipe and SPar generated code using the Prime Numbers application

This performance difference happens because the prime numbers application does not impose a heavy computational workload over the stream items, being mainly limited by the time required for serialization and message passing. In this scenario, the code generated by SPar suffers the disadvantage of requiring the serialization of a *struct*, which is used for generalizing the code generation. In contrast, the native ResiPipe code only needs to serialize primitive data types, such as the integer numbers exchanged between the data source and the processing stages. In addition, the pipeline generated by SPar also has one extra process to simulate the sink operator. Although this extra process does not perform any computations over the input records, it still introduces overhead by requiring the application to wait for the operator to receive all the messages before it finishes.

Figure 6.6 exhibits the throughput evaluation of the applications Mandelbrot, BZip2 compression and decompression ranging from 4 to 48 parallel processes, and the Eye Detector application ranging from 4 to 12 parallel processes. Unlike the other applications, the eye detector application distributes just one process per node (using 12 nodes) and relies upon the OpenCV parallel capabilities to spawn several threads to process the images. OpenCV has been configured to use the maximum concurrency level (number of parallel threads) supported by the CPU.

In the Mandelbrot application, the code generated by SPar exhibited a lower throughput than the native ResiPipe code, ranging from 92.45 records per second using 4 parallel processes to 1516.38 using 48 parallel processes. While ResiPipe presented a throughput of 115.07 records per second using 4 parallel processes and 1251.15 using 48 parallel processes. On the Eye Detector application, both the code generated by SPar and the native ResiPipe code presented virtually identical throughput. The lower throughput presented in the Mandelbrot application generated using SPar could be explained by the extra sink operator introduced as a current limitation of the ResiPipe code generation in SPar CLang.

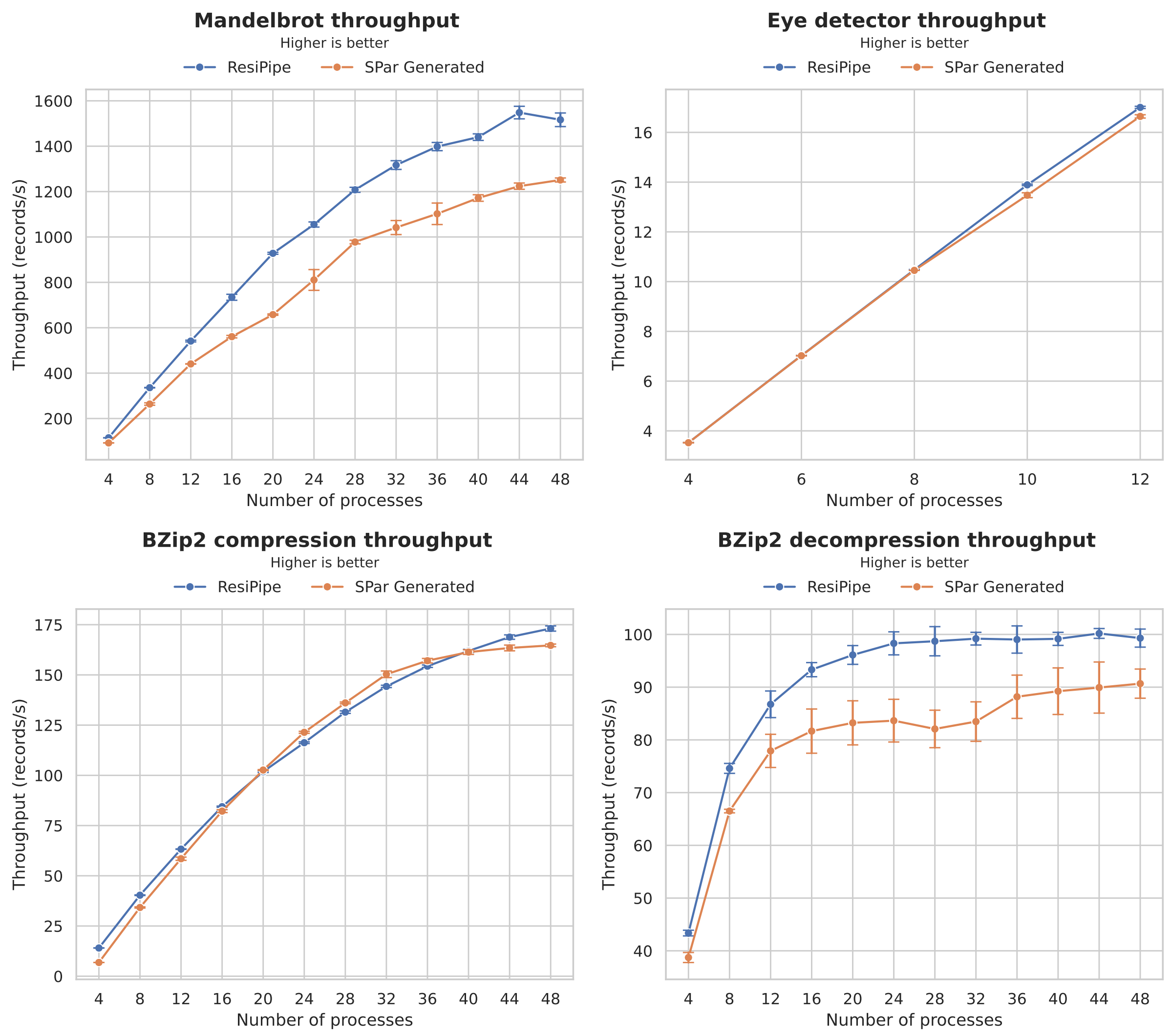


Figure 6.6: Performance comparison between ResiPipe and SPar generated code using the Mandelbrot, Eye Detector, BZip2 compression, and decompression applications

Regarding the BZip2 compression application, the application generated by SPar exhibits similar performance to ResiPipe native code, achieving a throughput of 6.84

records per second using 4 parallel processes and 164.69 using 48 parallel processes. The native ResiPipe application presented a throughput of 14.05 records per second using 4 parallel processes and 173.09 using 48 parallel processes.

In contrast, in the BZip2 decompression application, the code generated by SPar presented a lower throughput across all variations, ranging from 38.73 records per second using 4 parallel processes to 90.67 using 48 parallel processes. The ResiPipe native implementation for the same application represented a throughput of 43.36 records per second using 4 parallel processes and 99.30 using 48 parallel processes.

For all the experiments, the performance difference between the native ResiPipe applications and the SPar-generated ones appears to be related to the serialization and the presence of an extra sink operator. The overhead is more evident in applications with low computational costs, such as Prime Numbers, which only verify if the given integer number is prime, and BZip2 decompression, which is mainly limited by the I/O overhead of writing the decompressed output file over disk.

6.3 Final remarks

This chapter presented the implementation of ResiPipe code generation over the SPar CLang project and a set of five experiments that evaluate the performance overhead of SPar generated applications compared to native ResiPipe applications. In general, the current implementation of ResiPipe code generation allows the development of fault-tolerant distributed stream processing applications with some limitations compared to the native ResiPipe applications.

These limitations vary between some restrictions of the current implementation of the SPar CLang project, which is an experimental project and does not implement all the features present in the CINCLE implementation of SPar, and some limitations of the SPar language itself. Currently, the SPar language does not provide a clear way to implement the ResiPipe operator life cycle events *onInit*, *onExit*, *onError*, *onLocalSnapshotComplete*, and *onGlobalSnapshotComplete*. As depicted in Listing 6.1, a new set of SPar annotations *spar::OnInit*, *spar::OnExit*, *spar::OnSnapshot*, and *spar::onError* could be introduced to address this feature by defining function references for each life cycle event. The *spar::OnSnapshot* and *spar::onError* annotations would be ResiPipe specific, but the remaining annotations could be implemented regardless of the target runtime. Due to time constraints, this work has not implemented this new set of SPar annotations.

Listing 6.1: Example of SPar operator life cicle events implementation

```
#include <database.h>
```

```
void openDatabase()
{
    database.open("localhost",);
    database.openTransaction();
}

void closeDatabase()
{
    if (database.isTransactionOpen())
        database.commitTransaction();

    database.close();
}

void commitTransaction()
{
    database.commitTransaction();
    database.openTransaction();
}

void rollbackTransaction()
{
    database.rollbackTransaction();
}

int main(int argc, char **argv)
{
    [[spar::ToStream]]
    while (true)
    {
        // Ingest event from source
        int event = ingestEvent();

        // Process event
        [[spar::Stage, spar::Input(event), spar::Output(event)]]
        {
            event = processEvent(event);
        }

        // Persist the result in the database
        [[spar::Stage, spar::Input(event),
            spar::OnInit(openDatabase),
            spar::OnExit(closeDatabase),
            spar::OnSnapshot(commitTransaction),
            spar::onError(rollbackTransaction)
        ]]
        {
            database.insert(event);
        }
    }
}
```

Due to the lack of the life cycle events feature, the applications generated using SPar only support at-most-once semantics by default since it is impossible to implement the two-step commit protocol employed by ResiPipe to achieve exactly-once without introducing a new set of SPar annotations. Another way to achieve exactly-once semantics would be to implement replayable data sources and idempotent sinks. However, this will require extensive modifications to the user source code, which is against the objective of this research. It was also expected that both native ResiPipe applications and those generated by SPar would have similar performance results since both use ResiPipe as the engine for data processing. However, the limitations regarding the data serialization and the extra sink operator introduce a noticeable overhead over the application execution, which is more evident in applications that do not perform computationally intensive operations.

In future work, ResiPipe could be improved to allow more flexible pipelines, such as pipelines with only 2-stages, and support for sink operators with replication when the input ordering is disabled. These changes eliminate the need for the extra sink operator present in the current SPar ResiPipe code generation implementation, reducing the overhead in the generated applications. Regarding exactly-once semantics and the ResiPipe operator life cycle events, there is no easy way to implement these features using the existing mechanisms employed by the SPar language without introducing a new set of annotations that will bring SPar closer to regular ResiPipe classes.

7. CONCLUSION

This work introduced a new algorithm for resilient code generation inside the SPar software ecosystem, allowing parallel and distributed stream processing applications to be generated with fault-tolerance capabilities, completely abstracted from the application developer. The presence of fault-tolerance mechanisms in stream processing systems is highly important since these systems run for long periods, possibly indefinitely, and reprocessing all the data in case of failure is highly costly or even unfeasible. To address this issue, this work conducted a literature review regarding the implementation of fault-tolerance mechanisms in the current state-of-the-art stream processing systems and presented a new C++ library for distributed stream processing with fault-tolerance capabilities and exactly-once semantics.

The research goal of introducing fault-tolerance and exactly-once semantics in the SPar software ecosystem, preferably in such a way that the end user does not need to change their existing code, was partly achieved through the development of a new runtime for code generation named ResiPipe that provides fault tolerance through the implementation of the Asynchronous Barrier Snapshotting (ABS) protocol and supports exactly-once semantics through the implementation of the two-step commit protocol. The research goal was partially achieved since, despite ResiPipe supporting the two-step commit protocol for exactly-once semantics, it is impossible to implement it on the SPar generated applications since SPar does not abstract the operator life cycle events required to implement the two-step commit protocol. Currently, the only way to achieve exactly-once semantics on SPar is to implement replayable data sources and independent sink operators, which require significant modifications over the user code.

The performance experiments demonstrate that ResiPipe presents similar and, in some cases, better performance results compared to the remaining analyzed libraries while providing fault tolerance, out-of-the-box data serialization, and support for pipelines with multiple processing stages, a set of features that together are unseen over the remaining analyzed libraries. Regarding resource utilization, ResiPipe, DSParLib, and OpenMPI presented similar CPU usage. However, ResiPipe presented slightly higher memory consumption across the experiments (without considering MPR, which grouped all processes over a single machine).

The fault tolerance experiments indicate that ResiPipe can efficiently handle complex pipelines with multiple operators without introducing significant performance penalties. Independently of the snapshot interval, the snapshot mechanisms do not exhibit a significant overhead over the throughput of the use cases 1, 2, and 3 since these applications have a relatively small state size. However, the snapshot mechanism appears sensitive to large state sizes (use case 4), exhibiting a decrease in the throughput as snapshot frequency increases. This happens because ResiPipe persists the snapshot files over disk using a blocking operation, which makes the system susceptible to I/O overheads.

The programmability evaluation demonstrates that SPar requires fewer lines of code (SLOC) and less development time for all the applications. However, this is already expected since SPar is a domain-specific language with a higher abstraction level than the remaining analyzed libraries. Removing SPar from the analysis, ResiPipe requires fewer lines of code (SLOC) to implement all the applications of the experiment set and less development time (using the Halstead metrics) for most of the applications (Mandelbrot, Eye Detector, BZip2 Decompression, BZip2 Compression, and Sentiment Analysis). This advantage is justified since ResiPipe abstracts the process communication, parallel pattern, and data serialization, while the remaining analyzed libraries only abstract a subset of these features.

ResiPipe was included in the SPar software ecosystem as a target for code generation in the SPar CLang project, which is an experimental project and does not implement all the features present in the CINCLE implementation of SPar. Five experiments have been conducted to evaluate the performance overhead of SPar-generated applications compared to native ResiPipe applications. In general, the current implementation of ResiPipe code generation allows the development of fault-tolerant distributed stream processing applications with some limitations compared to the native ResiPipe applications.

The limitations vary between some restrictions of the current implementation of the SPar CLang project and some limitations of the SPar language itself for distributed applications. Currently, the SPar language does not provide a clear way to implement operator life cycle events, a feature required to implement the two-step commit protocol for exactly-once semantics on ResiPipe. The overhead evaluation also indicates that the data serialization limitations and the extra sink operator required by the current implementation of ResiPipe code generation introduce a noticeable overhead over the application execution, which is more evident in applications that do not perform computationally intensive operations.

As a direction for future works, this research could be improved and extended by exploring the following topics:

- **Dynamic scalability and dynamic process recovery**: ResiPipe currently only supports static linear pipelines without the possibility of increasing or decreasing the number of replicated operators on the fly. ResiPipe also requires a complete pipeline restart to recover the application in case of failure. Exploring ways to introduce dynamic scalability and dynamic process recovery capabilities in ResiPipe has the potential to reduce the overhead introduced by the failure recovery, allowing the application to dynamically restart a failed operator without stopping the entire pipeline.

- **Dynamic snapshot intervals**: ResiPipe currently only supports static snapshot intervals based on a time interval (e.g., every 30 seconds) or based on the number of processed records (e.g., every 1000 records). An adaptive snapshot mechanism based on metrics such as the system throughput and latency could be explored to dynamically increase or decrease the snapshot frequency, possibly reducing its overhead over the regular execution.
- **Support for more flexible pipelines**: The data stream pipeline pattern currently implemented by ResiPipe requires one data source operator, one or more middle-stage operators, and one sink operator. However, the system could be improved to support feedback loops, multiple data sources, and multiple sink operators.
- **High availability for the ResiPipe monitor agent**: Currently, ResiPipe has a single point of failure over the machine running the monitor agent, which is responsible for detecting failures and restarting the applications. A high-availability implementation of the monitor agent could be explored by distributing the process over the cluster machines and using a consensus algorithm to elect a leader. In case of failure of the leader machine, another machine on the cluster could take responsibility for running the monitor agent process.

7.1 List of published papers

Over the course of the author’s Master’s Thesis, one paper has been published at a national-level conference, WSCAD (*Workshop em Sistemas Computacionais de Alto Desempenho*), further renamed to SSCAD (*Simpósio em Sistemas Computacionais de Alto Desempenho*). A second publication, in the format of an extended abstract, was published at a regional school of high-performance computing named ERAD (Escola Regional de Alto Desempenho).

1. ALF, Lucas M.; HOFFMANN, Renato B.; MÜLLER, Caetano; GRIEBLER, Dalvan. **Análise da Execução de Algoritmos de Aprendizado de Máquina em Dispositivos Embarcados**. In: SIMPÓSIO EM SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO (SSCAD), 24. , 2023, Porto Alegre/RS. Anais [...]. Porto Alegre: Sociedade Brasileira de Computação, 2023 . p. 61-72. DOI: https://doi.org/10.5753/wscad.2023.235915.
2. ALF, Lucas M.; GRIEBLER, Dalvan. **Tolerância a Falhas para Paralelismo de Stream de Alto Nível**. In: ESCOLA REGIONAL DE ALTO DESEMPENHO DA REGIÃO SUL (ERAD-RS), 24. , 2024, Florianópolis/SC. Anais [...]. Porto Alegre: Sociedade Brasileira de Computação, 2024 . p. 119-120. ISSN 2595-4164. DOI: https://doi.org/10.5753/eradrs.2024.238679.

REFERENCES

[1] Akidau, T.; Balikov, A.; Bekiroğlu, K.; Chernyak, S.; Haberman, J.; Lax, R.; McVeety, S.; Mills, D.; Nordstrom, P.; Whittle, S. “MillWheel: Fault-Tolerant Stream Processing at Internet Scale”, *Proc. VLDB Endow.*, vol. 6–11, aug 2013, pp. 1033–1044.

[2] Akidau, T.; Bradshaw, R.; Chambers, C.; Chernyak, S.; Fernández-Moctezuma, R. J.; Lax, R.; McVeety, S.; Mills, D.; Perry, F.; Schmidt, E.; Whittle, S. “The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, out-of-Order Data Processing”, *Proc. VLDB Endow.*, vol. 8–12, aug 2015, pp. 1792–1803.

[3] Akidau, T.; Chernyak, S.; Lax, R. “Streaming Systems: The What, Where, When, and How of Large-Scale Data Processing”. O’Reilly Media, 2018, 352p.

[4] Ali, M. H.; Gerea, C.; Raman, B. S.; Sezgin, B.; Tarnavski, T.; Verona, T.; Wang, P.; Zabback, P.; Ananthanarayan, A.; Kirilov, A.; Lu, M.; Raizman, A.; Krishnan, R.; Schindlauer, R.; Grabs, T.; Bjeletich, S.; Chandramouli, B.; Goldstein, J.; Bhat, S.; Li, Y.; Di Nicola, V.; Wang, X.; Maier, D.; Grell, S.; Nano, O.; Santos, I. “Microsoft CEP server and online behavioral targeting”, *Proc. VLDB Endow.*, vol. 2–2, aug 2009, pp. 1558–1561.

[5] Andrade, G.; Griebler, D.; Santos, R.; Danelutto, M.; Fernandes, L. G. “Assessing Coding Metrics for Parallel Programming of Stream Processing Programs on Multi-cores”. In: 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2021), 2021, pp. 291–295.

[6] Andrade, G.; Griebler, D.; Santos, R.; Fernandes, L. G. “A parallel programming assessment for stream processing applications on multi-core systems”, *Computer Standards & Interfaces*, vol. 84, March 2023, pp. 103691.

[7] Andrade, G.; Griebler, D.; Santos, R.; Kessler, C.; Ernstsson, A.; Fernandes, L. G. “Analyzing Programming Effort Model Accuracy of High-Level Parallel Programs for Stream Processing”. In: 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2022), 2022, pp. 229–232.

[8] Andrade, G. L. “Improving parallel programming assessment : challenges, methods, and opportunities in coding productivity”, Ph.D. Thesis, School of Technology - PUCRS, Porto Alegre, Brazil, 2023, 201p.

[9] Andrade, H. C. M.; Gedik, B.; Turaga, D. S. “Fundamentals of Stream Processing: Application Design, Systems, and Analytics”. Cambridge University Press, 2014, 529p.

[10] Apache Software Foundation. “Spark Streaming Programming Guide”. Source: https://spark.apache.org/docs/latest/streaming-programming-guide.html, Dez 2023.

[11] Apache Software Foundation. “Structured Streaming Programming Guide”. Source: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html, Dez 2023.

[12] Apache Software Foundation. “Trident Tutorial”. Source: https://storm.apache.org/releases/2.4.0/Trident-tutorial.html, Dez 2023.

[13] Apache Software Foundation. “Apache Flink”. Source: https://flink.apache.org, May 2024.

[14] Armbrust, M.; Das, T.; Torres, J.; Yavuz, B.; Zhu, S.; Xin, R.; Ghodsi, A.; Stoica, I.; Zaharia, M. “Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark”. In: Proceedings of the International Conference on Management of Data, 2018, pp. 601–613.

[15] BSC Programming Models. “OmpSs User Guide”. Source: https://pm.bsc.es/ftp/ompss/doc/user-guide/OmpSsUserGuide.pdf, Mar 2025.

[16] BSC Programming Models. “NODES - nOS-V-based Dependency System”. Source: https://www.bsc.es/research-and-development/software-and-apps/software-list/nodes, Mar 2025.

[17] BSC Programming Models. “NODES - nOS-V-based Dependency System”. Source: https://www.bsc.es/research-and-development/software-and-apps/software-list/nanos6, Mar 2025.

[18] Bundschuh, M.; Dekkers, C. “The IT Measurement Compendium: Estimating and Benchmarking Success with Functional Size Measurement”. Springer Berlin Heidelberg, 2008, 644p.

[19] Carbone, P.; Ewen, S.; Fóra, G.; Haridi, S.; Richter, S.; Tzoumas, K. “State Management in Apache Flink®: Consistent Stateful Distributed Stream Processing”, *Proc. VLDB Endow.*, vol. 10–12, aug 2017, pp. 1718–1729.

[20] Carbone, P.; Fóra, G.; Ewen, S.; Haridi, S.; Tzoumas, K. “Lightweight Asynchronous Snapshots for Distributed Dataflows”. 1506.08603, Source: https://arxiv.org/abs/1506.08603, Apr 2025.

[21] Chandy, K. M.; Lamport, L. “Distributed snapshots: determining global states of distributed systems”, *ACM Trans. Comput. Syst.*, vol. 3–1, feb 1985, pp. 63–75.

[22] Consel, C.; Hamdi, H.; Réveillère, L.; Singaravelu, L.; Yu, H.; Pu, C. "Spidle: A dsl approach to specifying streaming applications". In: Generative Programming and Component Engineering, Pfenning, F.; Smaragdakis, Y. (Editors), 2003, pp. 1–17.

[23] Dolotin, V.; Morozov, A. "The Universal Mandelbrot Set". WORLD SCIENTIFIC, 2006, 176p, https://www.worldscientific.com/doi/pdf/10.1142/6136.

[24] Elnozahy, E. N. M.; Alvisi, L.; Wang, Y.-M.; Johnson, D. B. "A survey of rollback-recovery protocols in message-passing systems", *ACM Comput. Surv.*, vol. 34–3, sep 2002, pp. 375–408.

[25] Fragkoulis, M.; Carbone, P.; Kalavri, V.; Katsifodimos, A. "A survey on the evolution of stream processing systems", *The VLDB Journal*, vol. 33–2, Mar 2024, pp. 507–541.

[26] Garofalakis, M.; Gehrke, J.; Rastogi, R. "Data Stream Management: Processing High-Speed Data Streams". Springer Publishing Company, Incorporated, 2018, 1st ed., 537p.

[27] Gordon, R. D.; Halstead, M. H. "An experiment comparing fortran programming times with the software physics hypothesis". In: Proceedings of the National Computer Conference and Exposition, 1976, pp. 935–937.

[28] Grant, S.; Voorhies, R. "cereal - A C++11 library for serialization". Source: http://uscilab.github.io/cereal/, Jun 2024.

[29] Grant, S.; Voorhies, R. "The OmpSs Programming Model". Source: https://pm.bsc.es/ompss-2, Mar 2025.

[30] Griebler, D. "Domain-Specific Language & Support Tool for High-Level Stream Parallelism", Ph.D. Thesis, Faculdade de Informática - PPGCC - PUCRS, Porto Alegre, Brazil, 2016, 243p.

[31] Griebler, D.; Danelutto, M.; Torquati, M.; Fernandes, L. G. "An Embedded C++ Domain-Specific Language for Stream Parallelism". In: Parallel Computing: On the Road to Exascale, Proceedings of the International Conference on Parallel Computing, 2015, pp. 317–326.

[32] Griebler, D.; Danelutto, M.; Torquati, M.; Fernandes, L. G. "SPar: A DSL for High-Level and Productive Stream Parallelism", *Parallel Processing Letters*, vol. 27–01, March 2017, pp. 1740005.

[33] Griebler, D.; Fernandes, L. G. "Towards Distributed Parallel Programming Support for the SPar DSL". In: Parallel Computing is Everywhere, Proceedings of the International Conference on Parallel Computing, 2017, pp. 563–572.

[34] Griebler, D.; Vogel, A.; De Sensi, D.; Danelutto, M.; Fernandes, L. G. “Simplifying and implementing service level objectives for stream parallelism”, *Journal of Supercomputing*, vol. 76, June 2019, pp. 4603–4628.

[35] Halstead, M. H. “Elements of Software Science (Operating and programming systems series)”. USA: Elsevier Science Inc., 1977, 128p.

[36] Hoefler, T.; Dinan, J.; Buntinas, D.; Balaji, P.; Barrett, B.; Brightwell, R.; Gropp, W.; Kale, V.; Thakur, R. “Mpi + mpi: a new hybrid approach to parallel programming with mpi plus shared memory”, *Computing*, vol. 95–12, Dec 2013, pp. 1121–1136.

[37] Hoffmann, R. B.; Griebler, D.; Danelutto, M.; Fernandes, L. G. “Stream Parallelism Annotations for Multi-Core Frameworks”. In: XXIV Brazilian Symposium on Programming Languages (SBLP), 2020, pp. 48–55.

[38] Hoffmann, R. B.; Löff, J.; Griebler, D.; Fernandes, L. G. “OpenMP as runtime for providing high-level stream parallelism on multi-cores”, *The Journal of Supercomputing*, vol. 78–1, January 2022, pp. 7655–7676.

[39] Hoffmann Filho, R. B. “Impacts of parallel programming on limited-resource hardware”, Master’s Thesis, Pontifical Catholic University of Rio Grande do Sul, Porto Alegre, RS, Brasil, 2023, 102p.

[40] Hsu, M.-Y. “LLVM Techniques, Tips, and Best Practices Clang and Middle-End Libraries: Design powerful and reliable compilers using the latest libraries and tools from LLVM”. Packt Publishing, 2021, 370p.

[41] Hueske, F.; Kalavri, V. “Stream Processing with Apache Flink: Fundamentals, Implementation, and Operation of Streaming Applications”. O’Reilly Media, 2019, 310p.

[42] Laguna, I.; Marshall, R.; Mohror, K.; Ruefenacht, M.; Skjellum, A.; Sultana, N. “A large-scale study of mpi usage in open-source hpc applications”. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2019, pp. 14.

[43] Legaux, J.; Loulergue, F.; Jubertie, S. “Development effort and performance trade-off in high-level parallel programming”. In: International Conference on High Performance Computing Simulation (HPCS), 2014, pp. 162–169.

[44] Lin, C.-k.; Black, A. P. “Directflow: A domain-specific language for information-flow systems”. In: ECOOP: Object-Oriented Programming, Ernst, E. (Editor), 2007, pp. 299–322.

[45] Lin, W.; Fan, H.; Qian, Z.; Xu, J.; Yang, S.; Zhou, J.; Zhou, L. “STREAMSCOPE: continuous reliable distributed processing of big data streams”. In: Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation, 2016, pp. 439–453.

[46] Löff, J.; Griebler, D.; Fernandes, L. G.; Binder, W. “MPR: An MPI Framework for Distributed Self-adaptive Stream Processing”. In: Euro-Par 2024: Parallel Processing, 2024, pp. 400–414.

[47] Löff, J.; Hoffmann, R. B.; Griebler, D.; Fernandes, L. G. “High-Level Stream and Data Parallelism in C++ for Multi-Cores”. In: XXV Brazilian Symposium on Programming Languages (SBLP), 2021, pp. 41–48.

[48] Löff, J.; Hoffmann, R. B.; Griebler, D.; Fernandes, L. G. “Combining stream with data parallelism abstractions for multi-cores”, *Journal of Computer Languages*, vol. 73, December 2022, pp. 101160.

[49] Löff, J.; Hoffmann, R. B.; Pieper, R.; Griebler, D.; Fernandes, L. G. “DSParLib: A C++ Template Library for Distributed Stream Parallelism”, *International Journal of Parallel Programming*, vol. 50–5, 2022, pp. 454–485.

[50] Löff, J. H. “Simplifying Self-Adaptive Distributed Stream Processing in C++”, Master’s thesis, School of Technology - PPGCC - PUCRS, Porto Alegre, Brazil, 2023, 146p.

[51] Meijer, E.; Beckman, B.; Bierman, G. “LINQ: reconciling object, relations and XML in the .NET framework”. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, 2006, pp. 706.

[52] Message Passing Interface Forum. “Mpi: A message-passing interface standard version 4.1”. Source: https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf, Apr 2025.

[53] Murashko, I. “Clang Compiler Frontend: Get to grips with the internals of a C/C++ compiler frontend and create your own tools”. Packt Publishing, 2024, 326p.

[54] Murray, D. G.; McSherry, F.; Isaacs, R.; Isard, M.; Barham, P.; Abadi, M. “Naiad: A Timely Dataflow System”. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, 2013, pp. 439–455.

[55] Pieper, R.; Löff, J.; Hoffmann, R. B.; Griebler, D.; Fernandes, L. G. “High-level and Efficient Structured Stream Parallelism for Rust on Multi-cores”, *Journal of Computer Languages*, vol. 65, July 2021, pp. 101054.

[56] Pieper, R. L. “High-level Programming Abstractions for Distributed Stream Processing”, Master’s thesis, School of Technology - PPGCC - PUCRS, Porto Alegre, Brazil, 2020, 170p.

[57] Pop, A.; Cohen, A. “Openstream: Expressiveness and data-flow compilation of openmp streaming programs”, *ACM Trans. Archit. Code Optim.*, vol. 9–4, Jan. 2013, pp. 25.

[58] Qian, Z.; He, Y.; Su, C.; Wu, Z.; Zhu, H.; Zhang, T.; Zhou, L.; Yu, Y.; Zhang, Z. “TimeStream: reliable stream computation in the cloud”. In: Proceedings of the 8th ACM European Conference on Computer Systems, 2013, pp. 1–14.

[59] Rockenbach, D. A. “High-Level Programming Abstractions for Stream Parallelism on GPUs”, Master’s thesis, School of Technology - PPGCC - PUCRS, Porto Alegre, Brazil, 2020, 163p.

[60] Rockenbach, D. A.; Araujo, G.; Griebler, D.; Fernandes, L. G. “GSParLib: A multi-level programming interface unifying OpenCL and CUDA for expressing stream and data parallelism”, *Computer Standards & Interfaces*, vol. 92, March 2025, pp. 103922.

[61] Rockenbach, D. A.; Griebler, D.; Danelutto, M.; Fernandes, L. G. “High-Level Stream Parallelism Abstractions with SPar Targeting GPUs”. In: Parallel Computing is Everywhere, Proceedings of the International Conference on Parallel Computing (ParCo), 2019, pp. 543–552.

[62] Rockenbach, D. A.; Löff, J.; Araujo, G.; Griebler, D.; Fernandes, L. G. “High-Level Stream and Data Parallelism in C++ for GPUs”. In: XXVI Brazilian Symposium on Programming Languages (SBLP), 2022, pp. 41–49.

[63] Seward, J. “bzip2 and libbzip2 - a program and library for data compression”. Source: https://sourceware.org/bzip2/manual/manual.html, Apr 2025.

[64] Strom, R.; Yemini, S. “Optimistic recovery in distributed systems”, *ACM Trans. Comput. Syst.*, vol. 3–3, aug 1985, pp. 204–226.

[65] Supalov, A. “Inside the Message Passing Interface: Creating Fast Communication Libraries”. De Gruyter, Incorporated, 2018, 384p.

[66] Thies, W.; Karczmarek, M.; Amarasinghe, S. P. “Streamit: A language for streaming applications”. In: Proceedings of the 11th International Conference on Compiler Construction, 2002, pp. 179–196.

[67] Thies, W.; Karczmarek, M.; Gordon, M.; Maze, D.; Wong, J.; Hoffmann, H.; Brown, M.; Amarasinghe, S. “StreamIt: A Compiler for Streaming Applications”, Technical Report, Massachusetts Institute of Technology, 2001, 11p.

[68] Toshniwal, A.; Taneja, S.; Shukla, A.; Ramasamy, K.; Patel, J. M.; Kulkarni, S.; Jackson, J.; Gade, K.; Fu, M.; Donham, J.; Bhagat, N.; Mittal, S.; Ryaboy, D. “Storm@twitter”. In:

Proceedings of the ACM SIGMOD International Conference on Management of Data, 2014, pp. 147–156.

[69] Vogel, A.; Griebler, D.; Fernandes, L. G. “Providing High-Level Self-Adaptive Abstractions for Stream Parallelism on Multicores”, *Software: Practice and Experience*, vol. 51–6, January 2021, pp. 1194–1217.

[70] Wu, Y.; Tan, K.-L. “ChronoStream: Elastic stateful stream computation in the cloud”. In: IEEE 31st International Conference on Data Engineering, 2015, pp. 723–734.

[71] Zaharia, M.; Chowdhury, M.; Das, T.; Dave, A.; Ma, J.; McCauley, M.; Franklin, M. J.; Shenker, S.; Stoica, I. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for in-Memory Cluster Computing”. In: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, 2012, pp. 2.

[72] Zaharia, M.; Xin, R. S.; Wendell, P.; Das, T.; Armbrust, M.; Dave, A.; Meng, X.; Rosen, J.; Venkataraman, S.; Franklin, M. J.; Ghodsi, A.; Gonzalez, J.; Shenker, S.; Stoica, I. “Apache spark: A unified engine for big data processing”, *Communications of the ACM*, vol. 59–11, Nov 2016, pp. 56 – 65.

[73] Zhou, J.; Bruno, N.; Wu, M.-C.; Larson, P.-A.; Chaiken, R.; Shakib, D. “Scope: parallel databases meet mapreduce”, *The VLDB Journal*, vol. 21–5, oct 2012, pp. 611–636.

[74] Álvarez, D.; Sala, K.; Beltran, V. “nos-v: Co-executing hpc applications using system-wide task scheduling”. In: IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2024, pp. 312–324.

[75] Äruprup Nielsen, F. “A new evaluation of a word list for sentiment analysis in microblogs”, *CoRR*, vol. abs/1103.2903, Mar 2011, pp. 6, 1103.2903.

MARISTA

PUCRS

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Pesquisa e Pós-Graduação
Av. Ipiranga, 6681 – Prédio 1 – Térreo
Porto Alegre – RS – Brasil
Fone: (51) 3320-3513
E-mail: propesq@pucrs.br
Site: www.pucrs.br