PUCRS

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

JÚNIOR HENRIQUE LÖFF

SIMPLIFYING SELF-ADAPTIVE DISTRIBUTED STREAM PROCESSING IN C++

Porto Alegre
2023

PÓS-GRADUAÇÃO - *STRICTO SENSU*

Pontifícia Universidade Católica
do Rio Grande do Sul

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

SIMPLIFYING SELF-ADAPTIVE DISTRIBUTED STREAM PROCESSING IN C++

JÚNIOR HENRIQUE LÖFF

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. Luiz Gustavo Leão Fernandes
Co-Advisor: Prof. Dr. Dalvan Jair Griebler

Porto Alegre
2023

Ficha Catalográfica

L999s Löff, Júnior Henrique

Simplifying Self-Adaptive Distributed Stream Processing in C++ / Júnior Henrique Löff. – 2023.
146 f.
Dissertação (Mestrado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. Luiz Gustavo Leão Fernandes.
Coorientador: Prof. Dr. Dalvan Jair Griebler.

1. Stream Processing. 2. Distributed Systems. 3. Parallel Programming. 4. Self-adaptive. 5. C++. I. Leão Fernandes, Luiz Gustavo. II. Griebler, Dalvan Jair. III. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS
com os dados fornecidos pelo(a) autor(a).
Bibliotecária responsável: Clarissa Jesinska Selbach CRB-10/2051

JÚNIOR HENRIQUE LÖFF

SIMPLIFYING SELF-ADAPTIVE 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 30th, 2023.

COMMITTEE MEMBERS:

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

Profª. Drª. Lúcia Maria de Assumpção Drummond (IC/UFF)

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

Prof. Dr. Luiz Gustavo Leão Fernandes (PPGCC/PUCRS - Advisor)

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to the many individuals and organizations that have supported me throughout this research. First and foremost, I would like to thank my advisors for their guidance throughout the entire process. Their insights and feedback have been valuable in shaping this work into its final form. I am also immensely grateful to my spouse and family for their unwavering support and encouragement. Their love and motivation have helped me overcome the many challenges and obstacles I faced throughout this process. Additionally, I would like to thank my peers and colleagues, who have provided valuable feedback. Finally, I would like to acknowledge the financial support that was provided by the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPQ), Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), and SAP Enterprise. Without their support, this research would not have been possible.

SIMPLIFYING SELF-ADAPTIVE DISTRIBUTED STREAM PROCESSING IN C++

ABSTRACT

Data sources such as IoT sensors, user activity logs, health surveillance, and video streaming are becoming ubiquitous worldwide. Often, these sources produce big amounts of raw data, which traditional computing systems based on a store-first and compute-later batch paradigm struggle to handle. Stream processing is an effective solution that can manage these massive workloads while meeting low-latency and high-throughput requirements. However, developing a streaming system from scratch is a challenging endeavor. Distributed stream processing systems (DSPS) like Apache Flink and Apache Storm already provide many abstractions for transparent fault-tolerance, scheduling, communication protocols, and many other mechanisms that assist programmers in writing distributed parallel code. These tools are mostly written in higher-level programming languages like Java and Scala. Nevertheless, C/C++ distributed computing systems are preferred for high-performance computing (HPC), but in this domain, programmers lack high-level programming abstraction options. Consequently, C++ programmers usually rely on low-level MPI for coordinating distributed applications. Also, when using MPI, programmers often employ a static programming model to write their distributed applications, opposite to stream processing which dynamically deals with irregular workloads that vary in content, format, size, and input rate. Streaming systems should allow reconfiguration to self-adapt in response to data flow spikes, slowdowns, and load-balancing issues. This work aims to address these challenges by investigating the adaptability aspects of distributed streaming systems. For that, we introduce a new C++ framework called MPR (Message Passing Runtime), which simplifies the implementation of distributed stream processing applications. The framework relies on MPI's message-passing communication and implements many programming abstractions, including data transfer, serialization, load balancing, and back pressure. Moreover, we design a novel runtime system that

supports MPR's adaptability capabilities. The runtime system implements algorithms to handle dynamic process creation and includes a synchronization protocol for distributed process coordination. The experimental analysis reveals that MPR's dynamic runtime system can achieve performance comparable to a static MPI implementation. In addition, we also conduct experiments to evaluate and characterize MPR's adaptability capabilities. The characterization experiments show that MPR can readily self-configure itself in response to workload variations. Thanks to this work, MPR's runtime system on top of MPI is now a valuable tool that can be used to test and evaluate other self-adaptive algorithms for distributed stream processing.

Keywords: stream processing, distributed systems, parallel programming, programming abstractions, stream parallelism, self-adaptive, C++.

SIMPLIFICANDO O PROCESSAMENTO DISTRIBUÍDO DE STREAM AUTO-ADAPTATIVO EM C++

RESUMO

Fontes de dados como sensores IoT, logs de usuários, monitoramento de sinais vitais e streaming de vídeo estão cada vez mais presentes na sociedade. Muitas vezes, essas fontes produzem uma massiva quantidade de dados que os sistemas de computação tradicionais têm dificuldade para lidar. O processamento de stream é uma abordagem computacional que consegue lidar com essas cargas de trabalho massivas, atendendo aos requisitos de baixa latência e alta vazão. No entanto, desenvolver um sistema de streaming é uma tarefa desafiadora. Soluções como o Apache Flink e Apache Storm fornecem diversas abstrações de programação para tolerância a falhas, escalonamento, protocolos de comunicação e muitos outros mecanismos que ajudam os programadores a implementar códigos paralelos e distribuídos. Essas ferramentas são principalmente escritas em linguagens de programação de alto nível como Java e Scala. No entanto, no domínio de computação de alto desempenho, os programadores têm poucas opções de abstração de programação de alto nível quando se trata de sistemas de computação distribuídos escritos em linguagens de sistema como C/C++. Consequentemente, esses programadores muitas vezes dependem de ferramentas de mais baixo nível como o MPI para implementar aplicações distribuídas. Além disso, com MPI é comum empregar-se um modelo de programação estática para implementar aplicações distribuídas, opondo-se ao processamento de stream que lida dinamicamente com cargas de trabalho irregulares que variam em conteúdo, formato, tamanho e taxa de entrada. Os sistemas de processamento de stream devem permitir reconfigurações para se auto-adaptarem a picos no fluxo de dados, desacelerações e problemas de balanceamento de carga. Este trabalho tem como objetivo abordar esses desafios investigando os aspectos de adaptabilidade de sistemas distribuídos de processamento de stream. Para isso, introduziu-se uma nova ferramenta em C++ chamada MPR (Message Passing Runtime), que simplifica a implementação de

aplicações distribuídas de processamento de stream. Além disso, criou-se uma nova estratégia que suporta as funcionalidades auto-adaptativas do MPR. A estratégia implementa algoritmos para lidar com a criação dinâmica de processos e inclui um protocolo de sincronização para coordenação de processos distribuídos. Experimentos mostraram que o MPR consegue alcançar desempenho comparável a uma implementação MPI. Além disso, foram realizados experimentos para avaliar e caracterizar a auto-adaptatividade do MPR. Os experimentos de caracterização revelaram que o MPR é capaz de se autoconfigurar em resposta a variações na carga de trabalho. Com este trabalho, o MPR torna-se uma nova opção para implementar, testar e analisar algoritmos auto-adaptativos para processamento distribuído de stream.

Palavras-Chave: processamento de stream, sistemas distribuídos, programação paralela, abstrações de programação, paralelismo de stream, auto-adaptativo, C++.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

ABS – *Asynchronous Barrier Snapshotting*

AGAS – *Active Global Address Space*

API – *Application Programming Interface*

CPU – *Central Processing Unit*

DAG – *Direct Acyclic Graph*

DSPS – *Distributed Stream Processing System*

EOS – *End of Stream*

FIFO – *First-in First-out*

FPGA – *Field Programmable Gate Array*

GPU – *Graphic Processing Unit*

HPC – *High Performance Computing*

HPX – *High Performance Parallex*

I/O – *Input/Output*

LCO – *Local Control Object*

MPI – *Message Passing Interface*

NIC – *Network Interface Card*

OMPI – *Open MPI*

OPAL – *Open Portability Abstraction Layer*

ORTE – *Open Runtime Environment*

RAM – *Random Access Memory*

RDD – *Resilient Distributed Dataset*

RMA – *Remote Memory Access*

CONTENTS

1. INTRODUCTION

This chapter introduces the motivation, scope, and goals of this work. Also, it provides an overview of the initially proposed research plan and how it changed as research advanced. It then presents the objectives and the research contributions we achieved in this thesis. This chapter concludes with an outline of the contents that will be presented in this document.

1.1 Motivation

With the advancement of parallel hardware, it is crucial for legacy and modern sequential computing systems to adopt concurrency and parallelism for efficient resource exploitation. This has led to the modernization of software-layer systems in nearly every field of computer science. However, this process can range from being straightforward to challenging, depending on technology choices and system design. The difficulties of writing parallel computing systems have been widely documented in the literature [60, 37, 3, 82, 53]. Writing low-level parallel code is challenging and complex, requiring expertise in the application business domain and the underlying hardware. Otherwise, the computing system may face limited performance, inefficiencies, and unexpected problems.

To address these challenges, different computational approaches have been proposed over time. The first approach, ad-hoc parallelism (i.e., OpenMP [26] and MPI [42]), provide low-level mechanisms for synchronization, communication, and other specific features of each architecture, but requires the programmer to manually coordinate the computation. The second approach, higher-level parallelism abstractions (i.e., Intel TBB [67], Apache Flink [20], FastFlow [3], and WindFlow [61]), build on these low-level mechanisms to provide ready-to-use parallel patterns, like Map, Reduce, FlatMap, Pipeline, Farm, among others. The programmer only needs to focus on the parallelism semantics to assemble the correct data flow, instead of low-level complexities such as scheduling, communication, data transfer, and ordering. Recently, some attempts have been made towards even higher-levels of abstraction, using domain-specific languages, high-level programming models, and interactive interfaces that provide a unified language or model that can be compiled using different runtimes (i.e., SPar [37] and Apache Beam [6]).

Application programmers must align with the available solutions to efficiently exploit parallel hardware. Concurrently, modern applications from various domains have advanced and preemptively demand more computational power. Single multi-core machines can no longer meet these requirements, so these applications must be relocated to scalable distributed environments [53]. Streaming systems are an excellent example

of applications that must handle large amounts of continuously generated data, usually with high-throughput and low-latency. In stream processing, data items are processed on-the-fly before their value decreases over time rather than being stored in a database. Meanwhile, accumulated results are constantly updated each time a new data item flows through the stream. The streaming data items cannot randomly flow through the stream. Instead, they use a directed acyclic graph (DAG) as a guide.

Implementing distributed stream processing systems is particularly challenging as writing parallel code involves the programmer in many low-level details that require expertise for handling. Over the years, distributed streaming systems have become highly complex as they provide many transparent features for programmers, including load-balancing, back pressure, fault-tolerance, serialization, and message delivery guarantees. Most state-of-the-art distributed stream processing systems (DSPS) [7, 13] are implemented using high-level programming languages like Java and Scala as they are available in the same languages. However, in the high-performance computing (HPC) domain, C and C++ are the dominant programming languages. Despite attempts to provide programming abstractions for distributed stream processing in C++, they have not gained widespread popularity [14, 30, 24, 2, 17]. Instead, most programmers still rely on low-level parallelism abstractions such as MPI to write parallel applications.

The research challenge of distributed stream processing in C++ was already well studied by Pieper [64] during his master thesis in our research group. The author's research revealed that there are notable gaps in existing solutions for distributed stream processing in C++, with many discontinued, not compiling, or lacking documentation. Pieper's work introduced a new skeleton library (named DSParLib) [64] that aims to address these gaps by introducing a new skeleton library that provides programming abstractions via parallel patterns, improving programmability compared to MPI handwritten programs.

DSParLib was implemented on top of MPI, which is the focus of this work as we use it for the communication layer. The Message Passing Interface (MPI) is a widely used programming model for developing parallel applications in scientific and real-world domains that leverage High Performance Computing (HPC) for efficient processing. When MPI was first introduced, the target applications primarily relied on a static execution environment, executing on homogeneous computing systems and processing data with regular flow and good memory locality [25].

However, research advances in several application fields, including stream processing, data science, and machine learning, have revealed new challenges for computing systems that cannot handle dynamic scenarios. Overall, the challenges include resource exploitation in heterogeneous systems, handling irregular workloads, and scaling with varied communication patterns [25]. The stream processing paradigm, in particular, presents extra challenges, including dealing with large amounts of data, potentially infinite data in-

come, various data generation rates, and balancing low-latency and high-throughput. MPI has had to adapt to these changing demands in order to maintain its relevance. Therefore, different approaches were proposed in newer MPI releases, such as non-blocking communication and dynamic process management.

Unfortunately, a recent survey on MPI usage showed that some of these new-added features are rarely used [44], including dynamic process management. To address these challenges, a new runtime system is proposed that incorporates optimization strategies for self-adaptive distributed stream processing in MPI. We start from the assumption that using MPI's static execution paradigm is not well suited for self-adaptive stream processing applications. Adaptability is a key feature of the framework we introduce in this work, allowing applications to dynamically adjust the number of active processes during execution time. This type of adaptability is also known as horizontal scaling and enables streaming systems to self-adapt the number of active processes to workload variations.

1.2 Research Plan Roadmap

This section outlines the roadmap for our master's thesis studies. It is essential to understand the key aspects of our work and why we address different research domains with distinct challenges: programming abstractions for distributed stream processing in C++, runtime system design for dynamic process management on MPI, traces of design choices for process recovery, and self-adaptive distributed stream processing algorithms.

In the initial proposal of this thesis, the aim was to investigate recovery and adaptability in MPI, recognizing their requirement similarities. For instance, removing processes during execution time requires processes to be excluded from Pipeline execution. Recovery could be a viable solution, as to remove processes, we could simply kill a process and it would trigger the Pipeline Graph to re-configure itself, remove the process, and continue executing. However, our research showed that recovery in MPI is a challenging topic [41, 18, 51, 73, 72, 56], particularly as the MPI specification did not yet include an interface for fault-tolerance at the time of our research. Consequently, we decided to focus on the adaptability feature for the master's thesis, given the limited time available and the lack of recovery specification in MPI standardization.

Our original plan for this thesis was to extend DSParLib [64] (from our research group) to support dynamic process management capabilities, allowing for the addition and removal of processes during execution time. To that end, we conducted a thorough investigation of DSParLib and published some of our findings in a journal paper [55]. The results showed that DSParLib's runtime system was not designed to handle dynamic process management due to its specific design goals. We explain the limitations in further

detail later in Chapter 4. Therefore, we concluded that it would be more challenging to add this feature to DSParLib than to build a new distributed runtime system from scratch.

DSParLib was developed focusing on supporting distributed stream parallelism in SPar [37], a domain-specific language for expressing stream parallelism in C++ applications. SPar enables programmers to annotate their code using a simple and high-level language with only five attributes. Once annotated, SPar's compiler automatically generates parallel code targeting a specific architecture. By default, SPar generates code only for multi-core architectures, but DSParLib's work [64] extended SPar to also generate code for distributed systems.

At this point, our aim was to replace DSParLib's runtime system while retaining its high-level API. DSParLib drew inspiration from the structured parallel programming model [60]. The primary motivation of structured parallelism is the use of parallel patterns, which can be thought of as templates or skeletons that abstract away many of the parallelism complexities. DSParLib inherited knowledge from structured parallelism and offered two parallel patterns and their semi-arbitrary composition: Farm and Pipeline parallel patterns.

Our studies investigating DSParLib revealed some limitations regarding the use of parallel patterns. For example, the flexibility of the library is compromised since DSParLib enables stage replication to increase the degree of parallelism only via the Farm pattern. We explain the consequences of this design choice in greater detail and other limitations in Chapter 4. In addition, we discovered a simpler way of working with stream processing parallelism on top of MPI. Due to our findings, we decided not to use DSParLib's runtime system nor its high-level API. Instead, we propose implementing a new C++ framework called MPR (Message Passing Runtime) for self-adaptive distributed stream processing.

1.3 Research Contributions

The main objective of this thesis is to address the research problem of simplifying self-adaptive distributed stream processing. This research is focused on leveraging C++ and MPI, which are the leading programming language and programming model for distributed clusters, respectively. Combining stream processing with HPC and MPI posed a significant challenge, requiring extensive research to be conducted. Traditionally, applications implemented with MPI and HPC tend to be static. However, stream processing is highly dynamic, especially when it comes to enabling streaming applications to self-adapt. The focus of this work is to bridge this gap and enable stream processing to adapt to changing conditions and requirements.

We propose the following scientific contributions:

- A new C++ framework called MPR that simplifies the implementation of self-adaptive distributed stream processing. MPR provides different programming abstractions, such as dynamic process creation, scheduling, all-to-all data communication, transparent serialization, load-balancing mechanisms, ordering, and back pressure.
- A novel runtime system that is flexible, scalable, and capable of automatic self-adaptation based on a configuration file to support MPR's adaptability capabilities. The runtime system implements algorithms to handle process creation, job assignment, and data management and also includes a leader-based synchronization protocol that enables distributed process coordination on top of MPI.
- An MPI-based and self-adaptive algorithm for autonomic management on MPR's runtime system. This algorithm implements a MAPE feedback loop that monitors MPR's pipeline execution statistics and dynamically adds or removes processes to achieve a target throughput. The algorithm uses a dynamic adaptive scaling factor that varies based on how far the application's measured throughput is from the goal.
- An analysis of the performance impacts and overheads based on a set of experiments conducted on four stream processing applications with different characteristics. We provide performance comparisons with respect to DSParLib and handwritten MPI versions and characterize MPR's adaptability. These experiments demonstrate the flexibility and expressiveness of MPR's API while measuring the performance impacts and overheads introduced by its runtime system.

1.4 Outline and contents

The remainder of this thesis is organized as follows.

- *Chapter 2 - Background* presents the background to make this work self-contained. First, we introduce stream processing and its distributed concerns. Then, we present popular distributed programming models we consider to implement our work. Finally, we present an overview of state-of-the-art distributed stream processing systems, which include Apache Flink, Apache Spark (Spark Streaming), and Kafka Streams.
- *Chapter 3 - Related Work* presents our related work, which selects works targeting distributed stream processing in C++.
- *Chapter 4 - DSParLib Investigation* extends our related work by expanding the investigation in DSParLib, which is the foundation of our research.
- *Chapter 5 - MPR: Framework for Distributed Stream Processing* presents the new framework we introduce with this work called MPR for enabling self-adaptive dis-

tributed stream processing. We present MPR's details regarding its architecture, adaptive features, processing engine, reconfiguration protocols, and others.

- *Chapter 6 - MPR Performance Evaluation* provides a performance analysis to understand MPR's overheads and its impact on the execution of stream processing applications.
- *Chapter 7 - MPR Adaptability Evaluation* complements the experiments by presenting the evaluation of MPR's external autonomic module for application self-adaptive capabilities.
- *Chapter 8 - Conclusion* discusses the final remarks and future work.

2. BACKGROUND

This chapter aims to provide the necessary background to facilitate the understanding of the remaining content of this thesis. First, in Section 2.1, we introduce stream processing by discussing its main concepts. Then, Section 2.2 continues the discussion targeting distributed stream processing and its key mechanisms. Subsequently, in Section 2.3, we present two important programming models for communicating distributed resources. Finally, Section 2.4 presents a review of state-of-the-art DSPS and their mechanisms to support distributed stream processing.

2.1 Stream Processing

With the continuous growth of data sources, stream processing systems are becoming increasingly popular as an effective solution to handle massive amounts of fresh data. Traditional computing systems, based on a store-first and compute-later batch paradigm, struggle to keep up with the unique attributes of modern streaming workloads [53]. On the other hand, stream processing systems can process data as it arrives, extracting valuable insights before its value diminishes over time. By consuming and processing data in-memory, this computing paradigm can naturally model, express, and implement live data streams, enabling near real-time processing with low-latency characteristics [5].

The data flow is created by human interactions or devices such as cameras, IoT sensors, event logs, consumer activities, and financial transactions. Each produced information is considered a stream item and is included in the streaming data flow. In stream processing, stream data items can be processed long before all data is available. This allows computing systems to process over a consecutive, possibly infinite, data income. However, since the nature of the data is unknown, it also introduces dynamism to the system. Due to these characteristics, streaming systems should allow reconfiguration to self-adapt in response to spikes or slowdowns in the data flow. Also, data can vary in format and content, requiring flexibility with adequate pre-processing steps.

The stream processing paradigm can express and model a plethora of modern computing systems. Figure 2.1 illustrates such an ecosystem of streaming systems. From the exhibited applications, many do not yet have a practical implementation considering hardware barriers and/or software limitations. Therefore, a consolidation towards the stream processing paradigm may contribute directly to these computing systems, such as real-time efficient machine learning and its entire subset of applications, epidemic tracking, disaster forecasting, and many others.

Figure 2.1: Ecosystem of stream processing applications [54].

Stream processing systems operate as a sequence of interconnected computing operators, which can be dynamically scaled in terms of parallelism to meet the performance requirements of the workload. Each operator processes a specific computation on an incoming stream item, similar to a production line. For example, the stream item starts in the first operator (source), then it flows by each operator until reaching the final one (sink). These operators can either be stateful or stateless, where stateful means that they keep an internal state while stateless means vice-versa. Stateful operators are more complex to handle due to their potential replication challenges. In cases where stateful operators are replicated, synchronization mechanisms must be implemented to ensure the correctness of their replicated states. For instance, the Reduce parallel pattern can be used to synchronize associative computations by merging the operator's results using an associative binary operation.

2.2 Distributed Stream Processing

The exponential growth of data volume has made it clear that single-node, multi-core systems are not sufficient to handle modern stream processing workloads. Distributed stream processing systems (DSPS) have emerged as a solution to tackle the challenge of processing massive data flows. However, the current state-of-the-art DSPS have not kept up with the recent advancements in commodity clusters, virtualization techniques, public and private clouds, and other related technologies that enable distributed

computing. As a result, there is a growing demand for efficient and scalable DSPS capable of leveraging these modern technologies.

The emergent demand for DSPS introduced new challenges that remain open research issues until technology consolidation is achieved. Research advances towards DSPS are gradually being introduced within the state-of-the-art solutions, such as Apache Storm and Apache Flink, for further providing fault-tolerance capabilities, horizontal scaling, programmability aspects, transparent serialization, resource management, and job scheduling.

Over the years, DSPS have evolved to adopt a common architecture layout, although the underlying technologies may vary. A DSPS typically operates as a middleware in a three-tier streaming system, serving as an intermediary between a streaming user application and the underlying infrastructure [53]. Figure 2.2 provides a bird's-eye view of the hierarchical structure of a streaming system, which was inspired by work [53]. This figure serves to illustrate the motivation of our work, highlight open research challenges, and describe the role of a DSPS in streaming systems.

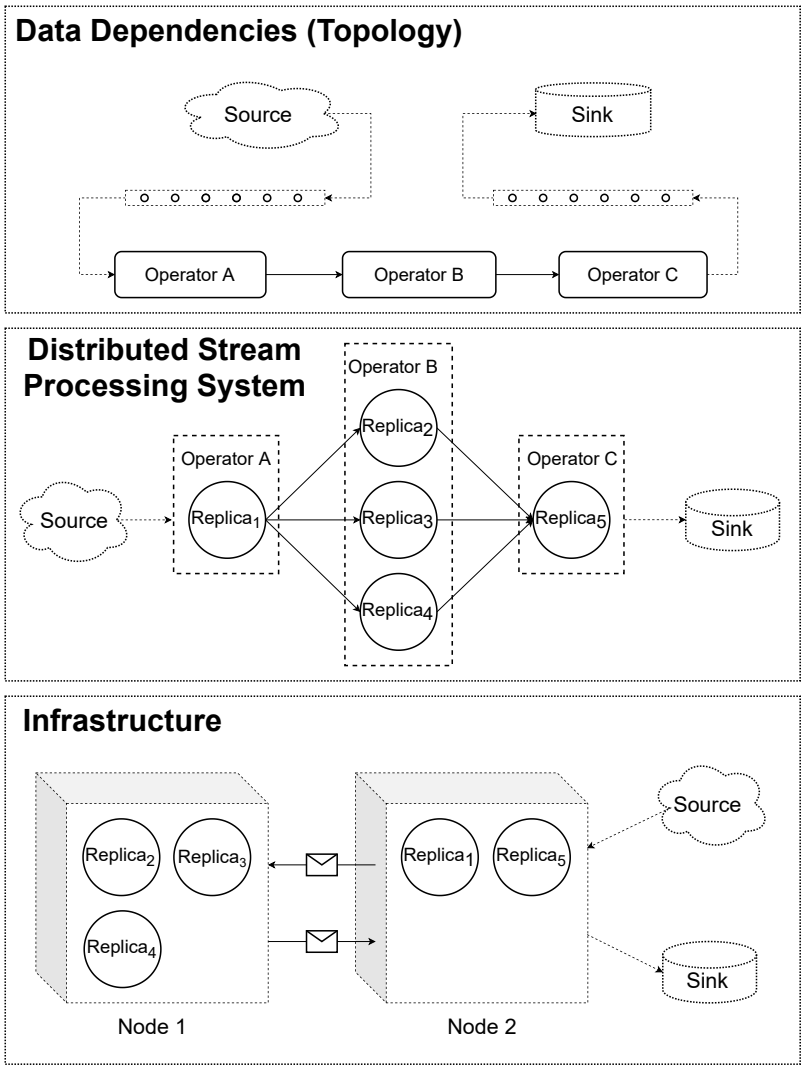


Figure 2.2: Bird's-eye view of a streaming system.

As can be seen in Figure 2.2, in the topmost abstraction layer, a data dependence schema is used to represent the streaming application topology. The representation is depicted as a sequence of operators where stream data items flow from the Source to the Sink operators. In between, stream items flow through one or more Compute operators, incrementally applying a computational step to each one of the received stream items. The processing of each Compute operator can be applied independently, where some examples are filters, aggregations, and transformations. The arrows of the topology represent the paths that stream items can travel inside the streaming system.

The DSPS is a critical component of a streaming system (middle abstraction layer), serving as a runtime system that enables high-throughput and low-latency processing while ensuring system resilience and availability. To achieve these goals, modern DSPS [7, 9, 10, 12] offer several abstractions, including fault-tolerance, load balancing, data serialization, back pressure, and task scheduling. By leveraging the data dependencies represented in the topology layer of the system, the DSPS processes a directed acyclic graph (DAG) of operators, optimizing resource utilization and exploiting parallel and distributed computing. In stream processing, Pipeline parallelism is the most common way to achieve parallelism, where independent tasks can be executed concurrently across multiple operators. Additionally, some operators can be replicated to execute in parallel on top of distributed resources to increase the application's overall throughput and decrease latency.

In the example illustrated in Figure 2.2, operators A and C run with a single process each, while operator B is replicated and runs with three processes. Together, they form a three-stage Pipeline that replicates the middle stage. This strategy is interesting when Operator A generates stream items faster than Operator B is able to process. By replicating, the Pipeline allows more items to be processed in less time. However, finding the optimal degree of parallelism is challenging as systems can experience workload incoming rate spikes or computing nodes can fail. This is where self-adaptive capabilities become essential in streaming systems to dynamically adjust the system's behavior in response to changes in the workload or infrastructure [76, 21]. In practice, achieving such self-adaptive capabilities and allowing the runtime system to adapt is still an open research issue.

Finally, the bottom-most layer of a streaming system represents its physical infrastructure, which can be a commodity cluster consisting of interconnected computing nodes or a cloud environment employing resource provisioning. In this layer, each computing node has one or more multicore processors capable of running multiple tasks in parallel. The DSPS creates replicas of the operators from the previous layer, which are then allocated to the physical resources. Protocols are needed to efficiently allocate the replicas to physical resources, taking into consideration the data dependencies expressed in terms of the streaming system's topology, as well as maintaining track of data com-

munication between different processes for load balancing purposes. Memory has new hierarchical levels for reading and writing data, and the DSPS must optimize this. For example, exchanging data between processes from the same shared space is significantly faster than exchanging data between distant processes via messages. Further complexities can arise from heterogeneous resources, such as accelerators or edge devices from fog computing. The underlying physical infrastructure layer plays a vital role in providing the necessary computing resources for the DSPS to achieve high-throughput and low-latency processing.

In the subsequent sections (2.2.1 to 2.2.8), we present and discuss in further detail the key mechanisms of DSPS that are of paramount importance to the distributed stream processing computing paradigm. These mechanisms ensure the readiness of DSPS for orchestrating the computation between a high-level programming interface for stream management and low-level efficient resource exploitation.

2.2.1 Fault-tolerance

Resilience is a critical feature for DSPS, as it enables a computing system to continue operating at an acceptable level of service in the face of various challenges and failures. Such issues may include sudden spikes in the data flow, bugs in the application logic, suboptimal memory usage, and other unexpected events. Some of these issues are further discussed in the next sections.

In this section, we are precisely interested in fault-tolerance, which is the ability of a distributed system to recover from failures while maintaining a consistent system state. Checkpoint-based rollback recovery is a popular approach for implementing fault-tolerance [28]. In it, all DSPS's processes will take periodic checkpoints (snapshots) of their consistent state and store them in persistent storage. Upon failure, the DSPS restores the system state to the most recent consistent set of checkpoints, which is known as *recovery line* [28].

There are different approaches for taking the snapshots using checkpoint-based recovery [22, 28]. Snapshots can be taken using uncoordinated or coordinated checkpointing. Uncoordinated protocols allow each process to decide when to take the snapshot, while coordinated checkpointing requires processes to synchronize their snapshots toward a consistent global state. The main advantage of coordinated protocols is it simplifies recovery and decreases storage overhead by maintaining a single consistent state for each process. However, disadvantages may include a higher latency due to extra synchronization between processes. Conversely, uncoordinated checkpointing protocols do not require synchronization. Instead, a consistent global state must be computed by piecing together all local snapshots. The main disadvantage of uncoordinated protocols are

twofold: the possibility of *domino effect* that can invalidate significant amounts of computed work and pointless snapshots that will never integrate a globally consistent state.

Figure 2.3 illustrates a recovery example using an uncoordinated checkpointing protocol. We took inspiration from [70]. In the Figure, S are snapshots, P are concurrent processes, and m are messages. Upon failure on P_2, the algorithm considers the recovery line as the most recent snapshots $S_{1,2}$, $S_{2,2}$, and $S_{3,2}$. However, $S_{1,2}$ says it received a message m_3 that was never sent in $S_{2,2}$, meaning $S_{1,2}$ is inconsistent. So, the algorithm rolls back P_1 from $S_{1,2}$ to snapshot $S_{1,1}$. Meanwhile, in the new recovery line ($S_{1,1}$, $S_{2,2}$, and $S_{3,2}$), messages m_1 and m_2 were not sent in $S_{1,1}$, which is inconsistent with $S_{2,2}$ and $S_{3,2}$. This domino effect may continue until all computed work is discarded and the system reaches the initial state.

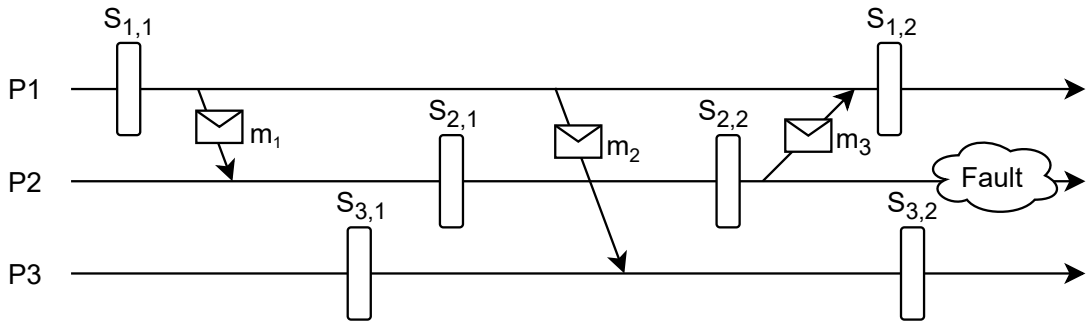


Figure 2.3: Consistent global states and domino effect.

Coordinated checkpointing is a widely used mechanism in DSPS to ensure fault tolerance. Unlike the domino effect that can occur in uncoordinated checkpointing, where the failure of one process can trigger a series of failures in other processes, coordinated checkpointing ensures consistent and successful snapshots. However, the overhead costs of synchronization can be significant. To minimize these overheads, non-blocking algorithms such as Chandy-Lamport's distributed snapshot algorithm [22] are preferred.

Chandy-Lamport's algorithm relies on the First-In-First-Out (FIFO) channel capabilities when exchanging messages throughout the distributed system. The algorithm is initiated when a process takes a snapshot and broadcasts a marker to all of its output channels. Upon receiving the marker, each process takes a snapshot of the current state and the channel and broadcasts a marker to its output channels. This process is repeated until all processes have a snapshot of their states and channels. Notably, the marker broadcast message is sent before sending any new application messages. At the end of the protocol, a consistent global state of the distributed system is obtained by saving a copy of all local processes' states and their channels.

2.2.2 Message Delivery Guarantees

The message delivery guarantees are critical for ensuring the correctness and consistency of the output in DSPS. There are three main guarantees: at-most-once, at-least-once, and exactly-once. *At-most-once* is a guarantee that ensures messages may be lost during execution time. It occurs when a message is sent at most once, which means that in case of a crash failure, the message can be lost forever, resulting in non-deterministic output. *At-least-once* ensures that messages are computed at least once, but duplicated items can be inserted into the streaming when unexpected events occur. In this case, the DSPS processes them twice and aggregates them to their computation output, resulting in a non-deterministic output. *Exactly-once* is the best-case scenario delivery guarantee because it ensures that each stream item will be processed exactly once. In case of crash failures, the runtime ensures that the output is correct and deterministic. Unfortunately, this guarantee is usually associated with costly overheads.

It is important to note that DSPSs can use a combination of these guarantees to achieve the best trade-off between output consistency and system performance. For example, some DSPSs may use at-least-once delivery guarantee with deduplication techniques to achieve output consistency and decrease system overheads.

2.2.3 Adaptability

Stream processing systems operate in highly dynamic and unpredictable environments that can be subject to fluctuations in workload size, data spikes, and other environmental changes. Furthermore, these systems may need to run indefinitely without interruption, making it crucial for them to be able to reconfigure themselves without downtime while maintaining the desired quality of service. Self-adaptation is a promising area of research that seeks to enhance a system's ability to detect and respond to unexpected challenges by making autonomous decisions and reconfiguring itself [75]. Self-adaptive streaming systems are particularly interested in managing the inherent dynamism of the stream processing paradigm and introducing autonomy to the computing system.

Self-adaptive systems can improve the performance, energy consumption, and operational cost of stream processing systems, among other aspects. During runtime, the system may adjust the degree of parallelism, processor frequency/voltage, or even completely reconfigure the streaming data flow. By doing so, self-adaptive systems can increase the intelligence of the streaming system, thereby reducing the need for manual intervention from programmers. This not only saves time and effort, but also reduces the

potential for human error and makes it possible to optimize the system in ways that would otherwise be infeasible.

Although self-adaptive streaming systems exhibit great potential, implementing and evaluating them remains a challenge [75]. One concern is that self-adaptive strategies have the potential to affect the safety of a streaming system. For example, changing the degree of parallelism during runtime can result in stream data item losses or duplication. Another difficulty is the integration of a self-adaptive algorithm with existing DSPS, as there is often a gap between industry solutions and new research proposals. This creates challenges for testing new approaches. Finally, self-adaptive strategies can introduce undesired overheads that significantly impact application execution and are difficult to detect. For example, other researchers [21] integrated their solution with Apache Storm's framework at the cost of performance losses due to halting the application each time a re-configuration was applied.

According to a recent survey [75], self-adaptive strategies hold great promise for improving stream processing systems, particularly in terms of performance. However, there are still many challenges to implementing and evaluating these strategies. Currently, most research in this area focuses on reactive strategies that use feedback loops and concepts from control theory to create instant responses based on application and environment monitoring. While some researchers are exploring proactive strategies that leverage machine learning to anticipate variations, this approach has its drawbacks, such as the need for large amounts of data to train and the time required to learn. Nevertheless, despite the current limitations, the state-of-the-art self-adaptive systems holds great potential for practical applications in the next generation of stream processing systems.

2.2.4 Serialization

Distributed computing environments are inherently designed to work with limited or no shared memory space. This property introduces many of the challenges that are currently faced by distributed systems, particularly with respect to how resources can be effectively coordinated to achieve optimal computing capabilities. In this section, we focus on the critical need for efficient serialization mechanisms that enable data to be exchanged as messages across a distributed stream processing system (DSPS).

Traditional DSPS rely on third-party software to convert application data structures stored in memory into architecture and language-independent formats that will be later transmitted throughout the network or stored on a disk. Serialization mechanisms can become a bottleneck because there are many computing steps involved each time new data is received over the network. For example, when new data is received over the network, the computing system must first prepare the data, resulting in access misses

until the data is copied from the network interface controller (NIC) to the main memory or cache. Then, the CPU becomes involved, computing the encode/decode step of the new data, which can be a time-consuming process. Finally, the data becomes available to the application.

In applications, data structures are often composed of multiple pointers (i.e., graphs and binary trees). Pointers are common for optimizing memory operations that otherwise would have to re-organize all memory contiguously each time a new data is stored [66]. However, communicating these data structures between computing nodes in a distributed system can lead to well-known overheads that increase the memory wall. The work [48] revealed that serialization represents a great slice of all the computation used by the applications executed on Google machines. It can be seen as a "datacenter tax" each application pays to run in a distributed environment.

The high cost of serialization is not the only issue but also the frequency with which these operations are performed. In the stream processing domain, communicating fine-grained and numerous stream items is commonplace in many applications from this domain. Therefore, serialization needs to be carefully addressed to not become a bottleneck and deteriorate the application performance.

A wide range of serialization libraries and protocols are available in the literature, each with different goals and trade-offs. Programmers can choose between performance and flexibility, or human-readability and data size, depending on their needs [80]. However, the design principles of these solutions can significantly impact their performance. For example, Java serialization provides reflection meta-data to facilitate serialization, but this comes at the cost of performance. Alternatively, compiled-based serialization can achieve higher efficiency than Java serialization, but they are more difficult to employ. Other options also include zero-copy serialization to reduce the overhead of multiple memory copies each time a network message is received.

Achieving zero-copy I/O has long been a goal of distributed computing research, as it can significantly reduce serialization and deserialization overhead [80]. Zero-copy serialization and deserialization strategies ensure that data is represented the same way in memory and on the message, so that processor cores are not involved with memory operations such as encode/decode steps. Today, some computing abstractions can exploit this and other approaches for optimization purposes by employing optimal strategies from hardware and software co-design [66, 80] (i.e., Direct Memory Access - DMA).

2.2.5 Resource Management

Job is a runtime entity representing a computation that needs to be done. An application may be expressed by a single or many jobs [5]. In stream processing, due to

fine-grained workloads and other characteristics, the applications from this domain are usually conceived using many jobs executed in parallel. Additionally, a job may be further subdivided into smaller computing slices to improve some aspects of application execution, specially latency-critical ones.

One of the main advantages of jobs relies upon the fact that they introduce certain degrees of isolation between distinct sections of an application. The benefits of that are manyfold. Unexpected faults that may occur in a part of the application do not directly affect the other running parts. Meanwhile, it also reinforces the placement of computations along different cores in a multi-core machine or different nodes in a distributed system. Those are fundamental mechanisms for allowing stream processing applications to further scale [5].

Resources may be described as any physical or virtual component of a computing system that has limited availability [53]. This term is broad and can be interpreted in many ways depending on the abstraction levels we are referring to in a computing system. In a local cluster, a DSPS may consider resources the memory, storage, network bandwidth, and processor cores. However, in public clouds, a DSPS may concern about IP addresses, network traffic, and object storage. More recently, an increased resource diversity is reached when the managed components become heterogeneous, such as specialized hardware that co-exists with general-purpose hardware in a computing system. DSPS may have to orchestrate computational jobs between dozens of multi-cores at the same pass they deal with GPUs and FPGAs.

Resource management is a critical aspect of DSPS, as these systems need to ensure that their resources are available and functioning optimally. One approach to achieve this is through the use of a heartbeat mechanism, where resources are required to either report their availability (push) or respond to probes (pull) to confirm their status [5]. This technique can help DSPS maintain an up-to-date inventory of resources and detect any anomalies that may arise. Additionally, it is important for DSPS to collect and analyze usage metrics to determine how much of each resource is being utilized. The metrics should be continuously collected and monitored throughout the system's execution to enable prompt adaptations to resource requirements, whether it's due to scaling up or down, or in the event of unexpected faults. Implementing proper resource management mechanisms can help DSPS efficiently exploit the available resources and improve system performance.

There are two approaches for adding or removing resources, they are known as horizontal and vertical scaling. In horizontal scaling, the user adds or removes computing nodes within the distributed system infrastructure. On the other hand, vertical scaling means adding or removing resources within a single node (i.e., adding storage disks, adding RAM memory, etc.). The solutions for supporting dynamic changes in the distributed environment are in their early stages and exhibit many limitations. When some

resource adaptation occurs, it often involves downtime to detect the modification due to system or application restart [53]. The main challenges of DSPS regarding resource management are avoiding application downtime and achieving transparency when dealing with the underlying parallel resources.

2.2.6 Task placement

Once resources are actively being monitored and checked for availability, the next step is to ensure that streaming systems efficiently exploit the available resource by optimal mapping decisions [53]. However, optimal task placement has been a long-standing challenge in distributed systems. For instance, in multi-core systems, placing intensive communicating threads as neighbors can improve communication through high-speed cache memory. Similarly, in distributed systems, placing communicating tasks in the same computing node can reduce overheads such as serializations and network traffic. Despite being reasonable, placing tasks in local resources may not always lead to optimal performance due to the potential for resource contention. When tasks compete for the same resources, the streaming system may experience severe performance degradation. Therefore, it is essential to employ efficient resource allocation and scheduling techniques that can mitigate resource contention and optimize task placement.

There is a great interest in resource-aware schedulers [53]. Traditional batch workloads can leverage historical data for gathering information to achieve load balancing, as their raw data is already stored and can be statically analyzed. However, streaming systems operate over an infinite stream of data items produced by sources and compute them even before all the data is available, leading to dynamic characteristics with volatile workloads that introduce uncertainty to the DSPS. This way, scheduling strategies are stuck in this dilemma of finding an optimal task placement. So, they may appeal to real-time scheduling decisions to deal with the workload and environment variations.

During the decision-making process, there are many considerations that can be taken into account as they represent trade-offs between different system capabilities. For example, a DSPS may achieve optimal resource usage by constantly moving around streaming tasks to balance the workload. However, migrating tasks between different computing nodes is computationally intensive. Besides, it considerably escalates in complexity when streaming tasks are stateful. This means that they need to repartition and migrate their internal states along with themselves to a new computing node. Therefore, the current strategies implemented in state-of-the-art DSPS may suffer from severe instability and high latency during the adjustment period.

2.2.7 Stream and Batch Processing

We have already characterized the stream processing paradigm in Section 2.1. In a nutshell, it is a natural and intuitive paradigm that can be used to model continuous flows of data. It works similarly to the human body, in which it constantly receives endless stimuli (i.e., visual, tactile, olfactory, etc.) and generates actions for these. When it comes to computing systems, despite the timely nature of data production, the majority of DSPS forgets it and often produces artificial batched stream items [20]. Data batches are sets of data collected over time that are dealt with together rather than processing individual data items. The batches may be created using a maximum time interval (i.e., hours, days, months, etc.), a maximum number of stream items, or a composition of both. The main reason for that is technology still imposes many overheads that cannot cope with the stream items' individual requirements. For example, each individual stream item communicated between different computing nodes includes its own share of overhead by serialization, message queuing, and network transportation.

More recently, some DSPS are moving towards a hybrid approach that integrates stream and batch processing as a unified model [1, 20]. This way, programmers can leverage a single programming model to implement real-time systems that continuously aggregate stream items in windows or that can process huge amounts of static data stored somewhere. Having these two streaming flow manipulation options is of paramount importance as they represent a trade-off between throughput and latency. For example, batch manipulation may be used to increase the maximum throughput of a streaming system, where the batching size stands for how much latency it is willing to give up for achieving higher throughput. Although streaming is the preeminent flow manipulation choice, batching is still needed for legacy streaming systems and algorithms that yet cannot execute using a true streaming flow.

Modern streaming systems impose strict requirements for low-latency and high-throughput aspects. This conceives a new streaming flow called micro-batching [71]. Compared to a true stream processing flow that handles each item at its arrival, micro-batching rather groups the streaming items in small batches with fixed time intervals. In practice, the ideal size of a batch or micro-batch depends on the DSPS capabilities, the streaming application needs, and workload characteristics. It is difficult to be predicted, and cumbersome for a programmer to statically configure an optimal size. For example, enlarging the micro-batching may result in higher waiting time intervals (compromising latency), but it reduces the frequency of communication calls (increasing throughput).

2.2.8 Streaming Flow Manipulation

In traditional computing systems, processing a finite amount of data is relatively easy, as the data can be divided into static chunks that match the number of processing nodes. In some cases, chunks of different sizes may be used to achieve load balancing. However, when dealing with streaming systems, the challenge of dividing computations in a balanced fashion becomes more complex, as the data stream is infinite and its size is not known in advance. To address this challenge, several strategies have been proposed, many rely upon slicing the infinite data stream into a collection of bounded datasets [1]. In the following paragraphs, we will discuss some of the most popular strategies used to address this issue.

When dealing with infinite workloads, one popular approach is to filter the data and transform the unordered data flow into a finite dataset of filtered stream items based on a time constraint. Another strategy is to use approximation algorithms, such as k-means or Latent Dirichlet allocation, to group the data into finite datasets that share common characteristics. By doing so, a mass of raw data that may seem odd at first glance can be reorganized into more manageable datasets.

Windowing is a widely used and essential concept in stream processing. It involves dividing data from a finite or infinite source into smaller chunks or windows. Window-based strategies are used in many DSPS to implement operations that would be infeasible or impractical otherwise. For instance, aggregations and averages do not need to store the entire streaming flow, and instead, windowing can be used to perform partial computations and incrementally aggregate their results using proper synchronization mechanisms.

One possible strategy of windowing is sliding windows. They are defined by fixed size and fixed time period. Figure 2.4 sketches some windowing strategies: (1) When the period is less than the size (left-hand side), windows will overlap and the same data may belong to multiple windows at the same time. (2) When both parameters are equal (middle one), it creates fixed windows. (3) When the period is greater than the size (right-hand side), windows will periodically select samples of the streaming flow.

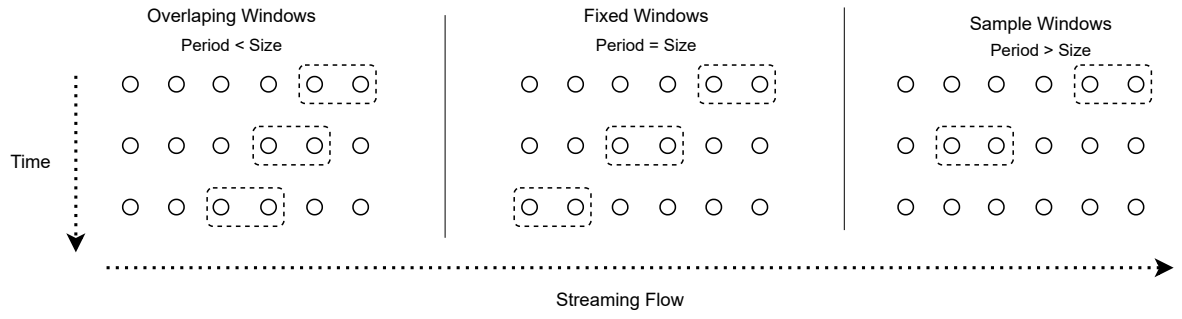


Figure 2.4: Windowing strategies.

Besides sliding windows, windowing also encompasses strategies for dynamic windows. One such strategy is sessions. The window expands dynamically and ingests stream items until a gap of inactivity greater than a given timeout is reached. This strategy is important for some classes of streaming applications. For example, a streaming system that analyses user behavior may adopt such an approach to process data in a proper way.

2.3 Parallel Programming Models

Managing the execution of a computing system with ease and maximum efficiency is a major challenge in computer science, particularly in distributed environments with heterogeneous resources. To address this challenge, many parallel programming models have been developed to optimize the utilization of computational resources and manage communication between processes. Two of the most important models in the high-performance computing (HPC) domain are Message Passing Interface (MPI) and High Performance ParalleX (HPX). In this section, we will discuss these models and their efficient approaches for coordinating the execution of parallel applications in distributed environments.

2.3.1 Message Passing Interface

The Message Passing Interface (MPI) is the leading parallel programming model for distributed computing in HPC and scientific applications. It was first conceived and released in 1994 and since then, the official standard has been updated multiple times. We are currently in standard MPI 4.0 [62], released in the middle of 2021. The intentions of newer MPI standards are to maintain relevance to the community by identifying and updating the interface according to the needs of modern computing systems [50]. For instance, ongoing working groups are actively exploring methods to facilitate the incorporation of standard techniques for fault-tolerance and integration with hybrid programming models, such as OpenMP, CUDA, and OpenCL, among others.

MPI is a specification rather than an implementation [62]. It defines a library interface with distributed operations represented as function routines. This does not make MPI a programming language, but rather a standard interface and programming model that can be used to optimize mechanisms in both upper and lower levels of abstraction. This allows vendors to exploit lower-level aspects to further optimize their proprietary hardware and programmers to exploit higher-level aspects to create new runtime systems that abstract message passing complexities using these same standard MPI routines.

MPI is a well-established standard interface that aims to provide efficient communication among processes in distributed systems. While MPI is mainly concerned with the message passing parallel programming model, it also offers a wide range of routines that support other parallel programming paradigms. These routines include collective operations, remote memory access (RMA), dynamic process management, and more.

The primary goal of MPI is to become a modern and self-contained standard for writing distributed programs. In pursuit of this objective, MPI has been designed to offer several essential features, including efficient communication, portability, flexibility, and language-independent semantics. These features enable MPI to provide a common foundation for developing parallel programs that can run efficiently on many hardware platforms and software environments.

MPI is designed to work on various platforms, ranging from a single machine to a combination of shared and distributed memory machines. By default, MPI processes communicate via TCP/IP sockets, but MPI also implements optimizations to exploit high-performance interconnect technologies, such as InfiniBand and Intel Omni-Path. MPI provides a wide range of functions to move data efficiently between processes and orchestrate communication among them.

While MPI targets efficiency and portability, it lacks programmability. Programmers need to manually implement mechanisms such as serialization, fault-tolerance, and elasticity, which may be challenging and time-consuming. MPI does not provide any of these mechanisms, although recent versions have made some efforts in fault-tolerance. Nonetheless, they are limited and might not be available in the short term. Thus, programming using MPI requires careful consideration of these factors to achieve optimal performance and functionality.

We mentioned MPI is a specification and not an implementation. Examples of existing MPI implementations are Open MPI and MPICH. Both are in conformance with standard MPI 3.1 and are slowly starting to support some specifications from MPI 4.0. MPI standard 3.1 exposes message passing and other mechanisms to efficiently orchestrate the computation between distributed resources [50]. Some of these mechanisms are: (1) *Point-to-point communication* are concerned with message passing between two processes in a cooperative fashion. (2) *Collectives* are concerned with coordinating communication and computation within a group of processes. (3) *Persistent communication* optimizes message passing that use the same argument list between two processes. (4) *One-sided communication* allows reading or writing data between processes, where only a single process is involved. (5) *MPI-IO* supports multiple processes to perform concurrent reads/writes to a single file. (6) *Dynamic process management* allows MPI processes to be created during execution time.

2.3.2 High Performance ParalleX

High Performance ParalleX (HPX) is a runtime system for concurrency and parallelism that was written in conformance with the newest C++ standards. The focus relies on high efficiency, scalability, and ease of programming for heterogeneous exascale distributed computing systems. HPX leverages asynchronous computing [45], which is a concurrent programming model that allows coordinating a large number of concurrent tasks, instead of a small number of processes. The asynchronous programming model is an alternative to the traditional message passing model. Moreover, it shows some advantages with respect to the latency barriers often implicitly imposed by MPI to synchronize communicating processes. In HPX, due to its programming model, each message may become an isolated task that can be computed asynchronously.

The HPX architecture is composed of different components that cooperate for achieving high efficiency through asynchronous computation [46]. Some components are: (1) a manager for the lightweight user-level threads created by HPX, which have tiny context switching overheads. (2) An active global address space (AGAS) that provides mechanisms to move objects in between nodes in a distributed system using a unique global address to identify them. (3) A message-driven networking layer (Parcel) that allows moving data to the work as well as assigning the work to data. (4) A light-weight local control object (LCO) are mechanisms that may be synchronous (i.e., barrier, semaphore, etc.) or asynchronous (i.e., future, dataflow, etc.) and have the ability to be triggered by a set of events that once are satisfied they authorize a thread to execute. (5) Performance counters that provide system metrics that are accessible through global addresses provided by AGAS mechanisms.

In a cluster, HPX nodes communicate via the parcel transport layer. The objects are identified using global IDs that point to the physical location of the data. This is supported by AGAS dynamic and adaptive capabilities that may move data between nodes for improving efficiency while holding the same global ID. Parcel messages by default use MPI via non-blocking calls for sending and receiving messages [15]. Upon receiving a Parcel message, a lightweight thread is created by HPX and is included in a thread manager system. In HPX, its asynchronous protocol enables messages to overlap communication and computation, meaning a message contains either data or remote method calls. Each message becomes a lightweight thread and is scheduled using the threading subsystem manager, which implements work-stealing scheduling to improve load balancing. This is one of the main contrasting characteristics of HPX with respect to MPI. While MPI creates coarse-grained threads (processes) that are placed in cores usually in a one-to-one fashion, HPX embraces fine-grained threads that may be handled in a thousand-to-one placement fashion.

The coordination between HPX lightweight threads is achieved with LCOs that can be in the shape of futures, dataflows, or traditional synchronization mechanisms. Futures represent results that are not ready yet, their states vary between executing and suspended depending on the availability of the required resources. Dataflows are operations that become available once a global state is achieved, for instance, when the runtime identifies that a set of futures finished their computations. Traditional synchronization mechanisms of HPX are in conformance with newer C++ standards and implement protocols such as mutexes, spinlocks, semaphores, and global barriers.

HPX is a recent distributed runtime system compared to MPI implementations, but it is increasing in popularity over the last few years. HPX leverages a new programming model that aims at achieving the high scalability required by the new era of exascale computing systems [46, 44]. Unfortunately, although HPX adopts C++ standards to improve writing parallel and distributed code, its programmability levels still rely closer to MPI than other state-of-the-art distributed systems.

2.4 Distributed Stream Processing Systems

This section presents the state-of-the-art frameworks and libraries that belong to the distributed stream processing domain. Our work is not directly compared to these tools, since our goal is not to create a new DSPS to compete with them. Instead, we want to leverage a lower-level parallel programming model, such as MPI or HPX, and provide higher-level abstractions. By studying these frameworks, we aim to investigate and understand the state-of-the-art mechanisms employed in different DSPS. This allows us to gain a broad perspective of the advancements in the stream processing domain, and to provide better functionalities to the runtime system introduced in this work.

2.4.1 Apache Flink

Apache Flink [7] is a unified stream and batch processing framework from Apache Software Foundation written in Java and Scala. It executes streaming flows in a pipeline and data-parallel fashion. Apache Flink provides fault-tolerance with exactly-once guarantees using a mechanism called Asynchronous Barrier Snapshotting (ABS) [20], which resembles the Chandy-Lamport's algorithm we explained in Section 2.2.1. Fundamentally, the synchronization is performed by control barriers inserted inside the streaming flows. Periodically, an operator finds a barrier in each one of its input streaming ports. Once all the input path barriers arrive, the operator takes a snapshot of its current state and writes it into persistent storage. When the state has been backed up, the operator forwards new

barriers to its output streaming ports. Eventually, all operators from the streaming system obtain a backed-up snapshot of their own states, and a consistent global snapshot is accomplished. Upon failures, the system reverts all operator states to the last consistent global snapshot and reloads all streaming items outside the snapshot barrier.

For serialization, Apache Flink provides built-in serializers for basic types (Java primitives), arrays, tuples, and a few others. User-defined or more complex data types may require a third-party library called Kryo. Apache Flink does not implement self-adaptive mechanisms yet, although there has been some progress in adaptive scheduling for a beta version [7] and also in research [47]. Still, the adaptability support does not make applications self-adaptive. Instead, it makes the application take a consistent snapshot of the job state, then halt the computation, and redeploy it with the updated parallelism. Additionally, the mechanisms are not fully integrated with the runtime and there are many limitations, especially regarding fault-tolerance and resource management tools.

2.4.2 Apache Storm

Apache Storm [13] is an open-source framework for distributed stream processing. With Trident, a high-level abstraction built on top of Apache Storm, it also supports batch and micro-batch processing with aggregations, filters, grouping, and join operations, as well as a better fault-tolerant strategy. By default, Apache Storm provides at-least-once message delivery guarantees. Additionally, it may provide exactly-once guarantees through Trident. Since Trident uses micro-batching tuples to process over the streaming flow, it attributes a unique identifier to each tuple. When tuples are replayed, they produce the same exact unique identifier. Operator states are stored in persistent storage. To keep a consistent global view of states, the mechanisms keep track of each tuple that updates the state and writes its identifier in an ordered manner. Upon failure, the fault-tolerant mechanisms recover the last stored state and replay the tuples. Considering that the state updates are ordered, the fault-tolerant mechanisms know exactly what tuples were already executed and which ones are the next. As a consequence, the strongest fault-tolerance guarantee of Apache Storm is satisfied with a considerable cost regarding storage space and computational overhead.

Apache Storm leverages the serialization mechanisms provided by a library called Kryo, the same as Apache Flink. By default, the DSPS can deal automatically with the serialization of primitive data types, strings, byte arrays, and some others. Otherwise, the user is expected to provide a custom serializer. If no serialization is provided, Apache Storm uses Java serialization whenever possible, which is known for being computationally expensive. Apache Storm does not implement self-adaptive mechanisms. Nonethe-

less, some efforts were made from the literature to implement adaptive mechanisms on top of Apache Storm [29, 81, 27, 32, 52]. Storm has more adaptive scheduling extensions than any other DSPS. We attribute this to the possibility of replacing Apache Storm's default scheduler with a custom scheduler. This highlights the contributions of our work that aims at providing a modular framework to enable programmers and researchers to focus on an isolated part of the runtime system.

2.4.3 Apache Spark

Apache Spark [12] was initially conceived for dealing with batch processing only. However, it includes spark streaming, an extension built on top of Apache Spark to further support near real-time stream processing. Spark streaming supports micro-batching to meet the low-latency requirements of stream processing. Apache Spark was introduced as an optimized runtime system with respect to Apache Hadoop [8], which is historically relevant in Big Data. For instance, Apache Hadoop suffers from severe performance degradation when dealing with iterative workloads that require frequent access to a storage disk. Therefore, Apache Spark introduced a new concept based on RDDs (Resilient Distributed Datasets) that load data in memory. The strategy is similar to a cache hierarchy from shared memory architectures, in which constantly used data items remain in memory closer to the processor. Therefore, avoiding the overhead cost of reading data from distant memories.

By default, Apache Spark uses Java serialization mechanisms to provide data serialization. Also, it supports integration with the Kryo library, similar to the previous frameworks. Apache Spark implements a fault-tolerant system that provides at-least-once message delivery guarantees. For that, is uses a mechanism based in a write-ahead logging and checkpointing. Checkpointing in Spark streaming is straightforward and consists in periodically writing the operator's state in a persistent storage. Write-ahead logging is a complementary strategy that can detect if a data item has already been processed or not, thus obtaining a consistent view of a streaming system. All state updates are first written to ahead logs and only after the operation is successful the modifications are made available for processing. When failures occur, Apache Spark uses these logs and combines information from the metadata and state from the last checkpoint. This strategy cannot detect if streaming items are processed twice, and because of that Spark streaming guarantees are at-least-once only. Although Apache Spark introduced a few adaptive strategies in its release version, Spark streaming does not enable such adaptive strategies by default. In spite of that, there are experimental attempts from research to provide adaptive mechanisms to replace the default Spark streaming scheduler [23, 63].

2.4.4 Apache Heron

Apache Heron [9] is a distributed stream processing engine written mostly in Java. It is another incubated DSPS within the Apache Software Foundation. It was first conceived and implemented by developers from Twitter to replace Apache Storm, that according to them, could no longer suit their needs due to some limitations [49]. Apache Heron uses Kryo for serialization and deserialization, the same as other Apache tools. Additionally, it does not support self-adaptive mechanisms by default. However, Dhalion [31] is an adaptive system from research that provides self-regulation capabilities and that can be enabled in Apache Heron. It is not completely clear how the fault-tolerance mechanism works for Apache Heron according to the available documentation.

Apache Heron declares they provide at-most-once, at-least-once, and effectively-once that will take place depending on how the user architectures the topology. The later guarantee is a term used by Heron since they claim that exactly-once is misleading and impossible to guarantee. Instead, they define effectively-once as a system that may compute twice but the output is only considered once. Also, they claim that to ensure effectively-once, the topology must be stateful and idempotent, while embracing a data source with strong consistency. In this case, stateful component means that the streaming operators from the topology must store their own state each time a new stream item is processed (this can be done manually or using state managers); and the idempotent component means that executing over the same data twice always produces the same outcome/result. Upon failure, the fault-tolerance mechanism uses an ordered identifier incremented for each new state update and a deduplication strategy in the sink to ignore duplicated writes. According to Apache Heron's documentation, effectively-once is only guaranteed when a de-duplication sink is available. If that is the case, Spark Streaming and other DSPS also could ensure exactly-once guarantees using a de-duplication sink.

2.4.5 Kafka Streams

Kafka [10] is a distributed streaming framework that has tremendously increased in popularity over the last years. The framework embraces tools like Apache Kafka, Kafka Streams, and Kafka Connect under the same umbrella, each one with a different goal: (1) Apache Kafka provides a set of mechanisms to deal with the steaming flow. Sources produce data and publish onto Kafka topics. These topics are a collection of messages that last forever inside a Kafka cluster. Users can read, modify, or even replay old messages. Furthermore, Apache Kafka implements fault-tolerant mechanisms to support the never-ending workload of stream processing systems. Because of that, Apache Kafka is used

as a middle layer between real data sources and streaming systems. Consequently, it is used by almost every state-of-the-art DSPS and plays an essential role in each of their own strategies for ensuring fault-tolerance. (2) Kafka Streams is a DSPS that allows the processing of streaming items. This is the API layer that compares to our work and we will give more details in the following. (3) Kafka Connect is a high-level API of Apache Kafka that allows the integration between Kafka topics and different DSPS, such as Apache Spark, Apache Storm, Apache Flink, etc.

Kafka Streams inherit fault-tolerance capabilities due to the native integration between Apache Kafka and its runtime system. In this strategy, all local state updates are stored in a durable changelog, which is an Apache Kafka topic by default. Once the state is stored in a topic, Apache Kafka ensures it is persistent using the exact mechanism that enables stream messages to be persistent. Kafka has a strong notion of "committed" messages to ensure they are actually stored in a log and de-duplicates them using transactions. To prevent a topic from growing indefinitely, a log compaction mechanism keeps track of the most recent values and purges the old ones. Kafka Streams can provide exactly-once guarantees due to this fault-tolerant mechanism. Upon failure, the system restarting time depends on the size of the state that was stored in a changelog topic. To minimize the overhead, Kafka Streams enable replicated standby copies of local states. The user specifies the number of standby replicas and what state they should replicate. If tasks reinitialize or are migrated, Kafka Streams will assign them to nodes that already contain a replicated state. Kafka Streams provides built-in serialization mechanisms for several Java primitives. Additionally, it equips users with templates for serializing and deserializing data using the Avro and Protobuf libraries.

2.4.6 Apache Samza

Apache Samza [11] is written in Scala and Java and is another DSPS supported by Apache Foundation. It was developed by LinkedIn and was designed in conjunction with Apache Kafka. Apache Samza provides data serialization through Java serialization. It expects the user to register their own custom data types or to use Apache Kafka topics for communication. Apache Samza proposes a fault-tolerant mechanism different from the others, which is a composition of Kafka state changelogs and message checkpointing. Upon failure, which can be of a software nature, Apache Samza leverages a strategy they call host-affinity. Therefore, the state is stored on the same machine disk as the processing task to establish data state locality. The complementary strategy to ensure fault-tolerance is message checkpointing. Some requirements are expected from the streaming source, like a fixed order of messages and their offset indicating a position in the stream.

Consequently, each Apache Samza task stores the current offset of read messages, in which each new message moves the offset forward. Periodically, another processing component checkpoints the current offset of each processing task. Because of this synchronization gap, the current offsets checkpoint may not be consistent, compromising the strong exactly-once delivery guarantee but ensuring at-least-once. Both Kafka Streams and Apache Samza do not provide self-adaptive mechanisms.

2.4.7 Akka Streams

Akka is a framework leveraging an actor-based model that simplifies the construction of concurrent and distributed applications. It was written in Scala, but it allows Scala and Java bindings. An actor-based system is basically composed of actors, similar to object-oriented, which is essentially composed of objects. Actors communicate using message passing. Upon receiving a message, actors may take actions such as: processing the message itself, forwarding it, initializing a new actor and forwarding the message, or completely ignoring it. In this work, we are interested in Akka streams, a library built on top of the Akka framework for stream processing.

A user may implement a DSPS in Akka using different strategies. The most natural way is integrating Akka streams with Akka actors. This way, Akka streams manages the streaming flow while each actor manages its state. Concerning the type of actor, persistent actors are more suitable than traditional ones because they ensure that internal states are persistent and can be recovered when an actor fails or is migrated. Additionally, the streaming system may become even more resilient when integrating it with the Akka persistence API. Akka streams uses Protobuf to serialize data types. Alternatively, it supports Java serialization, Kryo library, and Chill.

By default, instead of snapshotting the entire internal states, Akka uses an event-sourcing approach that only stores the events that modify the state. Therefore, in case of recovery, the modifications are replayed to rebuild the internal state. Upon failure, a persistent actor uses the latest snapshot and replays the subsequent modifications to obtain a consistent state. Durable states are an alternative fault-tolerant mechanism with respect to event sourcing. In this mechanism, the entire state is stored in persistent storage. Only the latest state is saved, and old ones are discarded. The state is first stored in memory and after written into the storage. During this period, all actions are halted until the state is successfully stored. Akka persistence provides message delivery guarantees for at-least-once.

2.4.8 Summary

Table 2.1 summarizes our findings regarding the distributed stream processing systems (DSPS) investigated in this section. Our research indicates that Java is the dominant programming language used to write the DSPS. The performance gap between a higher-level language abstracting hardware details (i.e., JVM-based) and lower-level system languages (i.e., C/C++) is well known [82]. The work [82] concluded that streaming systems parallelized with state-of-the-art DSPS presented bottlenecks due to inefficient resource exploitation in a single node. Meanwhile, when providing an equivalent C++ and MPI implementation, the throughput increased, achieving gains of up to two orders of magnitude with respect to DSPS like Apache Flink, Apache Storm, and Apache Spark [82].

Our investigation also revealed that only a few DSPS support a hybrid processing model that integrates stream and batch processing. Micro-batching, which is critical for stream processing, allows the system to balance throughput and latency by adjusting the batch size. The serialization is accomplished using the Kryo library for almost all DSPS. Alternatively, some implement built-in serialization for primitive data types (i.e., Apache Flink and Kafka Streams) or use standard Java serialization support. However, nearly all DSPS do not recommend using Java serialization due to its overhead penalties.

All DSPS are fault-tolerant. They provide different implementations and strategies to achieve fault-tolerance, which explains the different delivery guarantees. Exactly-once is the strongest guarantee that allows streaming systems' results to be deterministic and reliable, similar to the sequential programming paradigm. DSPS that provide exactly-once store the state in persistent storage and monitors the progress made in the streaming system. For others, their strategy does not allow the detection of duplicated items. Therefore, they provide exclusively at-least-once guarantees. For low-level systems like MPI implementations or HPX, fault-tolerance is non-existent and the runtimes provide at-most-once guarantees. In case messages are not successfully received, the user must manually implement a strategy to resend them.

Self-adaptive mechanisms are not commonly integrated in stable releases of DSPS. Although many solutions have been proposed in the literature, their practical application is limited due to the lack of experimental setups using realistic workloads and real-world streaming systems. Dynamic changes during execution time can have side-effects, such as halting the DSPS for a long time or requiring a system restart to apply the modifications. Consequently, self-adaptive mechanisms are still under active research and development, and how they can be effectively integrated into stream processing Pipelines remains an open issue.

Maintaining the exact ordering of stream data items is required in some applications such as video or compression, where the output results must follow the same

order as the input stream. However, integrating ordering with other mechanisms such as fault-tolerance and self-adaptive adds further complexity to streaming systems. Our investigation revealed that some DSPS do not support ordering, while others employ different approaches for addressing it. Some DSPS can reorder stream items but only in batch mode, while others employ actor-based systems or are integrated with streaming producers like Apache Kafka. However, we found no information about how ordering is handled when failures occur.

Table 2.1: Comparison between state-of-the-art Distributed Stream Processing Systems.

Framework	**Programming Language**	**Serialization**	**Fault-tolerance**	**Self-adaptive**	**Delivery Guarantee**	**Processing Type**	**Stream Ordering**
Apache Flink	Java Scala	Built-in for primitives; Kryo library	Asynchronous Barrier Snapshotting (ABS)	-	Exactly-once	Stream; Batch; Micro-batch	No
Apache Storm	Java Clojure	Java serialization; Kryo library	Re-playable data; State management	-	At-least-once; Exactly-once via Trident	Stream; Micro-batch via Trident	Yes, via Trident (batch only)
Apache Spark	Scala Java	Java serialization; Kryo library	Write-ahead logging; Checkpointing	-	At-least-once	Batch; Micro-batch via Spark Streaming	No
Apache Heron	Java C++	Kryo library	Re-playable data; State management	-	At-least-once	Stream	No
Apache Samza	Scala Java	Java serialization	State Changelog; Message Checkpoint	-	At-least-once	Stream	Yes
Kafka Streams	Java	Built-in for primitives; Avro; Protobuf	State Changelog	-	Exactly-once	Stream	Yes
Akka Streams	Scala	Protobuf	Event sourcing; Durable state	-	At-least-once	Stream	Yes

3. RELATED WORK

This chapter presents related work to ours. We selected papers that provide high-level abstractions which aim at supporting distributed stream processing in C++. Most of these works introduced libraries and programming APIs based on the structured parallel programming paradigm. Their majority was conceived employing low-level mechanisms built on top of MPI. HPX is more recent and was not used by related work.

FastFlow is a C++ programming library that was originally designed for multi-core processors and later extended to support heterogeneous and distributed computing. In previous work [2], FastFlow used ZeroMQ for inter- and intra-node communication, and provided programming templates for zero-copy message passing. In a recent study [74], FastFlow was further extended to implement a new runtime system that supports the execution of FastFlow shared-memory applications on a distributed domain. This new runtime system allows for a single structured parallel programming model to run applications in hybrid shared/distributed-memory environments. The modifications to make a program distributed include identifying parallelism opportunities in FastFlow's building blocks, and assigning them to different groups accordingly. Then, FastFlow enables parallel execution by mapping these groups to distributed nodes using a Json configuration file. Serialization is done using Cereal or by manually implementing a programming template for zero-copy data transfer. Fault-tolerance and adaptability are not covered by their work.

Thrill [17] is a C++ framework for computing distributed batch workloads. Their goal is to implement a new Big Data framework to compete against the ones we discussed in Section 2.4. Their solution does not consider self-adaptive and they do not implement mechanisms to include fault-tolerance. To communicate messages they provide custom routines that resemble MPI calls and, optionally, the user can use MPI itself. Serialization is done using built-in templates or Cereal library for complex data types.

The next two works [78, 59] represent early attempts to provide stream processing abstractions over MPI. However, when these studies were published, the stream processing programming model was not as widespread as it is today, so their proposed frameworks and libraries are not high-level abstractions for MPI-based stream processing. Instead, their contributions are closer to stream-oriented programming models for MPI. In [59], the authors study how to integrate MPI and stream processing to exploit network locality and topology. Meanwhile, in [78], a lightweight strategy is introduced for employing MPI in stream processing to support workflow computation using a directed acyclic graph (DAG). However, neither work implements abstractions for serialization, fault-tolerance, and self-adaptive, nor do they ensure delivery guarantees or implement ordering.

MPI Streams [68] proposes an extension to MPI targeting the stream processing paradigm. It specifies a new set of MPI functions that are able to support the Map and Reduce patterns. For example, using standard MPI functions (i.e., `MPI_reduce`, `MPI_sum`, etc.) the authors created new functions (i.e., MPIStream_send, MPIStream_Operate). As consequence, the proposed extension is not a high-level abstraction over MPI. Instead, it helps building streaming systems but still couples the user with low-level MPI aspects.

ESkel [14] is a programming abstraction for MPI that was written in C. It leverages parallel patterns (or simply, skeletons) that already implement many parallelism complexities. For example, eSkel provides the Pipeline and Farm patterns that are popular in stream processing. These parallel patterns are prototypes that already implement mechanisms such as scheduling protocols, an underlying DAG, communication strategies, and others. However, the abstractions are still very similar to native MPI and the programmer deals with lower-level aspects such as pointers, MPI data types, and buffer sizes. ESkel also does not implement fault-tolerance or self-adaptive approaches. Serialization is done via default MPI data types.

Muesli [24] is a pattern-based parallel programming abstraction for shared and distributed architectures. It was written in C++ and uses MPI to perform communication between different nodes in a cluster while using OpenMP to support multi-cores. Muesli supports both data and stream parallel patterns. It does not implement fault-tolerance and self-adaptive mechanisms. Serialization is done using built-in templates that require to be manually implemented by the user.

Quaff [30] is another C++ pattern-based parallel programming library written in C++ to abstract low-level MPI. Due to optimizations at compile time and template-based meta-programming techniques, their programming abstraction reaches a good balance between performance and programmability. Their library provides parallel patterns using a object-oriented paradigm. Same as the others, Quaff does not mention fault-tolerance and self-adaptive, also does not provide mechanisms for ordering and delivery guarantees.

DSParLib [64] is a pattern-based programming abstraction from our research group. It can be used for expressing distributed stream parallelism on C++ via arbitrary composition of parallel patterns. DSParLib provides abstractions to ease the development of streaming systems using MPI. It offers building blocks that can "wrap" existing sequential code, and then interconnect between themselves to obtain a DAG of computation. DSParLib uses dynamic processes management from MPI-2 that for now is statically configured at compile time. DSParLib does not provide fault-tolerance and only tries to send messages once, ensuring at-most-once guarantees. Additionally, it does not implement self-adaptive. DSParLib serializes data types using built-in C++ templates. More details are presented in Section 4 as we extend our study on DSParLib’s investigation.

Table 3.1 summarizes our findings when investigating C/C++ parallel programming abstractions for distributed stream processing. The results revealed that almost half of the programming abstractions do not tackle modern stream parallelism or do so partially. Serialization is often done using low-level MPI data types or via built-in C++ templates that require manual implementations. Most programming abstractions proposed by our related work were discontinued and documentation is scarce. Fault-tolerance and self-adaptive are topics that remain research issues. None of the related work approached these topics. Only two mention it by proposing fault-tolerance as future work. Delivery guarantees are also not focused on any work. Messages are sent once and are expected to always be received on the other side. Ordering is a challenging feature because it requires to be implemented inside the runtime system in conformance with the remaining mechanisms. Only DSParLib made an effort to fully support it.

Our work addresses critical gaps in the field of distributed stream processing for C++. We go in a different direction from the related work solutions presented in Table 3.1, that mostly leverage a static MPI programming model. Instead, we focus on the dynamic characteristics of stream processing and try to bridge it with dynamic process management in MPI. Our primary objective is to enable adaptability in streaming systems by allowing processes to be added or removed during execution. In order to achieve this goal, we propose the novel design of a flexible, efficient, and modular runtime system called MPR (Message Passing Runtime), which to the best of our knowledge, is the first to enable adjusting parallelism during execution time for self-adaptive distributed stream processing in C++. MPR's runtime system includes algorithms for dynamic process creation, Job assignment, data management, and a leader-based synchronization protocol that is used to coordinate the MPI processes.

In addition to the lower-level runtime system, we also implement a set of programming abstractions for MPR to become a viable solution for implementing distributed streaming systems in C++. We implemented programming abstractions for dynamic process management, scheduling, data communication, serialization, load balancing, ordering, and back pressure. To develop MPR, we leverage insights from state-of-the-art solutions and aim to contribute to a modular framework that can be extended with new algorithms and optimizations by other researchers. Currently, there is a significant research gap between conceiving a new self-adaptive algorithm for distributed mechanisms and integrating it with a real DSPS. Existing tools and related work make it challenging to implement extensions or optimizations, which can limit progress in the field. Our work addresses this limitation by providing a new framework that enables researchers to develop and test their own self-adaptive algorithms and optimizations. In Chapter 4, we extend our investigation on DSParLib, and in Chapter 5 we present the MPR framework.

Table 3.1: Comparison between related work.

Work	**Transport Layer**	**Serialization**	**Fault-tolerance**	**Self-adaptive**	**Delivery Guarantees**	**Stream Parallelism**	**Ordering**	**Modular**
FastFlow	TCP/IP; MPI	Built-in templates	-	-	at-most-once	Yes	No	No
Thrill	TCP/IP; MPI	Built-in templates; Cereal library	-	-	at-most-once	Partially	No	No
MPI Streams	MPI	MPI data types	-	-	at-most-once	Partially	No	No
MPI Hybrid	MPI	MPI data types	-	-	at-most-once	No	No	No
MPI Light-weight	MPI	MPI data types	-	-	at-most-once	No	No	No
eSkel	MPI	MPI data types	-	-	at-most-once	Yes	Partially	No
Muesli	MPI	Built-in templates	-	-	at-most-once	Yes	No	No
Quaff	MPI	Boost	-	-	at-most-once	Yes	No	No
DSParLib	MPI	Built-in templates	-	-	at-most-once	Yes	Yes	No
MPR	MPI	Programming abstraction	-	Parallelism adaptability	at-most-once	Yes	Yes	Yes

4. DSPARLIB INVESTIGATION

In this chapter, we extend our related work study and research DSParLib in further detail as it will be the foundation of our research. We report on the DSParLib investigation and its low-level mechanisms that enable distributed stream processing. The features we incorporate into our runtime system are influenced by DSParLib, and even its limitations have provided us with valuable insights to enhance our system. Additionally, we present our findings from the experiments we conducted to parallelize a complex streaming application using DSParLib's API. By analyzing the flexibility and performance of DSParLib, we were able to identify areas of improvement for our runtime system.

This chapter is organized as follows. In Section 4.1, we present an overview of DSParLib by describing its runtime system and programming model. In Section 4.2, we introduce the strategy created to parallelize a stream processing application with DSParLib. Subsequently, in Secion 4.3 we present the experimental evaluation. Lastly, in Section 4.4, we describe the limitations found in DSParLib and propose improvements that will be later used to implement the new runtime system.

4.1 DSParLib Overview

DSParLib (an acronym for Distributed Stream Parallelism Library) is a parallel library for implementing distributed stream parallelism on C++ applications. DSParLib implements programming abstractions on top of MPI to simplify parallel programming via a higher-level API. DSParLib was designed taking inspiration from the structured parallel programming paradigm. It introduces the notion of building blocks that enables programmers to "wrap" existing code. Then, these building blocks can be interconnected to model the application data flow. In addition, DSParLib provides some compilation time verifications that help the programmer in determining if the interconnected building blocks are correct. For example, DSParLib checks that a stage receiving an integer is interconnected with a stage sending integers.

Figure 4.1 illustrates how DSParLib's building blocks are interconnected to produce a streaming flow. The fundamental component of a building block is the sequential wrapper (white block), which wraps a block of code containing the application's computational logic. The other two blocks are the Input and Output communicators (yellow and blue), which are used to serialize data through the network. They can also be seen as blocks that wrap up the mechanisms for sending and receiving data in DSParLib.

To enable distributed stream processing with DSParLib, programmers are equipped with two parallel patterns that can be used to "wrap" sequential code: (1) The Pipeline

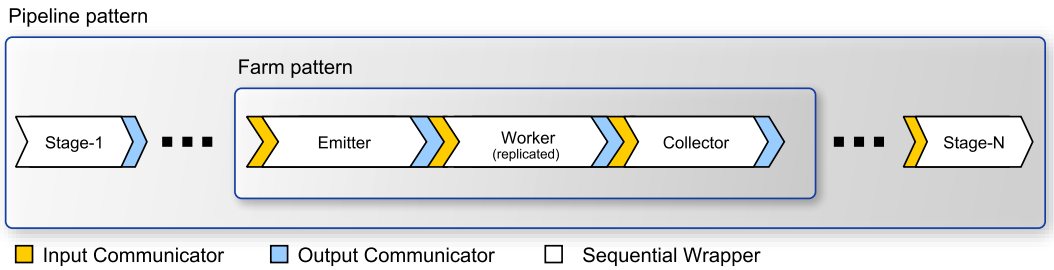


Figure 4.1: DSParLib Composable and Reusable Building Blocks [55].

pattern is a sequence of interconnected sequential stages. It can be implemented with a Sequential Wrapper + Output communicator at the beginning and an Input communicator + Sequential Wrapper at the end. A pipeline stage can only be sequential and cannot be replicated to execute in parallel. (2) The Farm parallel pattern can be seen as a particular case of the Pipeline pattern with always three stages (Emitter, Worker(s), Collector). Sometimes stages that have intensive computational routines can be replicated to improve performance. In these cases, DSParLib offers the Farm parallel pattern, where the intensive computation is assigned to the middle stage (Worker) for replication. To maintain the functional correctness of the data flow, the Farm pattern requires a scheduler (named Emitter) and gatherer to collect the parallel Workers' results (named Collector). The streaming data flow moves from left to right.

In addition to the ready-to-use parallel patterns (Farm and Pipeline), DSParLib supports semi-arbitrary pattern composition. For applications that require multiple replicated stages, it enables nesting Farms into Pipeline stages, as also depicted in Figure 4.1. Other compositions are currently not supported. This is an essential feature since DSParLib enables stage parallelism only via composition with the Farm pattern. Each Pipeline parallel stage must be implemented as a Farm, otherwise, it will be a sequential Pipeline stage.

4.1.1 Building Blocks

In DSParLib, programmers can wrap sequential code using DSParLib's building blocks. The available blocks are of three types regarding their data interaction: send only, send and receive, and receive only. We show a code example for each sequential wrapper type in Listing 4.1. For example, let us consider an application that processes the prime numbers from `0` to `totalNum`. First, `Stage1` extends the Emitter template and is responsible for generating the data (lines 1 to 6). Each new loop iteration (line 4) emits a new data item (line 5). Then, the `Stage2` is the Worker that receives the items, processes them, and emits a boolean indicating if that number is a prime number (lines 8 to 18).

Any C++ computation or function could replace the computation from lines 11 to 17. The `Process()` method executes over each stream item received (lines 10 and 23). Finally, `Stage3` is the Collector, which accumulates all the results (lines 20 to 26). In DSParLib, the inputs from the previous building block are received using the `Process(inputs)` and the resulting outputs are scheduled using the `Emit(outputs)` function. This strategy is inherited from structured parallel programming. Lower-level data communications and message passing implementations are usually abstracted from the programmers. Other complexities can arrive in serializing the data, but we will discuss them later.

```
1  class Stage1: public Emitter <int> {
2  public:
3    void Produce() override {
4      for(int i=0; i<totalNum; i++){
5        Emit(i);
6  } }; };
7
8  class Stage2: public Worker <int, bool> {
9  public:
10   void Process(int &i) override {
11    bool isPrime = true;
12    for (int j = 2; j < i; j++) {
13     if (i % j == 0) {
14      isPrime = false;
15      break;
16    } }
17    Emit(isPrime);
18 }; };
19
20 class Stage3: public Collector <bool> {
21 public:
22   int primes;
23   void Process(double &isPrime) override {
24    if(isPrime) {
25     primes++;
26 } }; };
```

Listing 4.1: Example of DSParLib sequential wrappers.

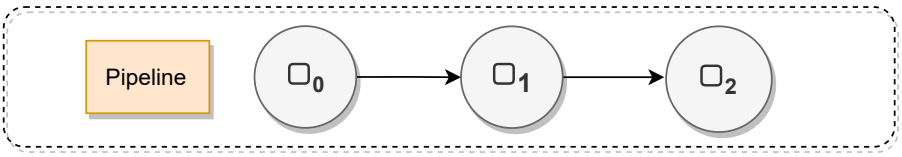


Figure 4.2: Parallel activity graph first example [55].

To understand how building blocks are assembled in DSParLib consider the parallel activity graph *pipe*($\square_0, \square_1, \square_2$). We represent $\square_n$ (square box) as a wrapped block

of code. It represents a Pipeline with three sequential stages, as illustrated in Figure 4.2. The Stage2 wrapper implementation in previous Listing 4.1 (line 8) can be assign only to $\square_1$. However, it can not be assigned to $\square_0$, nor $\square_2$. The reason is that Stage2 expects an input and an output, which is not satisfied when being the first or last stage of the Pipeline. To introduce the last stage, for instance, programmers must first adapt the sequential wrapper. Note that both the Emitter and Collector have special rules. The former cannot receive data items since it only generates data, and the latter cannot send items (`Emit()`) since it only collects data.

4.1.2 Data communication

DSParLib provides two programming abstractions to simplify data communication: `MPISender` and `MPIReceiver`. It offers SendTo and Receive methods to implement messages that are being sent or received. The programmers must call these methods in the correct order to serialize and deserialize the data. For example, if programmers send an `int` and a `float` message, they must receive the data in the same order. When data is contiguously allocated in memory, DSParLib provides zero-copy operations since data is *sent as it is* and thus does not involve the CPU.

DSParLib implements another programming abstraction on top of `MPISender` and `MPIReceiver`. It is called SenderReceiver. The purpose of SenderReceiver is to provide an abstraction to serialize and deserialize data being sent/received through the network when using native C++ data types. In that case, the SenderReceiver becomes a communicator object that transparently deals with data serialization.

```
1  struct CustomType{
2   double num;
3   unsigned char * buffer;
4  };
5  class CustomAbstraction : public SenderReceiver <CustomType> {
6   void Send(MPISender &sender,MessageHeader &msg,CustomType &data) override {
7    sender.SendTo(msg,data.num);
8    sender.SendTo(msg,data.buffer,size);
9   }
10  CustomType Receive(MPIReceiver &receiver,MessageHeader &msg) override {
11   receiver.Receive(msg,num);
12   receiver.Receive(msg,buffer,size);
13   return CustomType(num,buffer);
14  }
15 }
```

Listing 4.2: Example of Custom Type serialization.

Listing 4.2 showcases DSParLib's SenderReceiver API when dealing with custom data types. Lines 1 to 4 describe a custom data type containing two fields. To implement data communication, programmers can implement a class extending SenderReceiver (line 5) and overrides its functions for sending (line 6) and receiving (line 10). Then, programmers call SendTo() for the sender and Receive() for the receiver to each struct field in the same order. DSParLib handles the communication details and serializes the MPI communications. DSParLib's templates can deal with up to 3-dimensional statically or dynamically allocated arrays or other contiguously allocated data.

4.1.3 Pattern Composition

In this section, we discuss the parallel patterns available in DSParLib and their composition. Programmers may use C++ type inference and the auto keyword to ease the DSParLib implementation. Listing 4.3 showcases an example of how programmers can use DSParLib's building blocks. The example presents pattern composition using the following schema: $pipe\{\square_0, farm[E(\square_1), W(\square_2), C(\square_3)], \square_4\}$. The resulting parallel activity graph is illustrated in Figure 4.3.

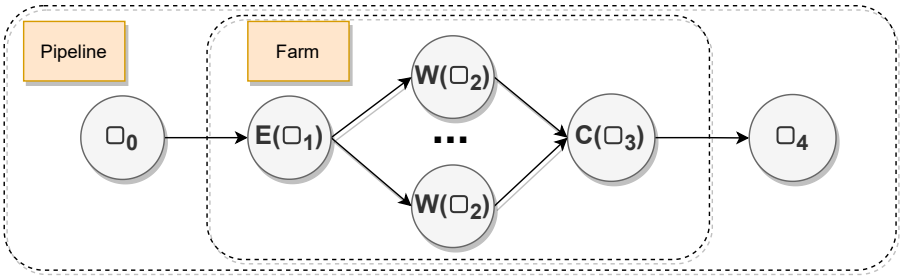


Figure 4.3: Parallel activity graph second example [55].

```
1  void PipelineComposition() {
2    auto comm = dspar::SenderReceiver<double>();
3    auto stageBeforeFarm = dspar::Stage(FirstStage,comm);
4    auto farm = dspar::Farm(comm, Emitter, comm, Worker, comm, Collector, comm);
5    farm.SetWorkerReplicas(10);
6    farm.SetOnDemandScheduling(true);
7    farm.SetCollectorIsOrdered(true);
8    auto stageAfterFarm = dspar::Stage(LastStage, comm);
9    Pipeline pipe(&stageBeforeFarm, &farm, &stageAfterFarm);
10   pipe.Start();
11 }
```

Listing 4.3: Example of Pipeline and Farm composition.

In this example, the processes communicate using only the double data type. Since this is a contiguous memory type (data is placed in a single chunk of memory), the

message passing communication is abstracted using `dspar::SenderReceiver<double>()` (line 2). Pipeline stages can be implemented, as showcased in lines 3 and 8, by specifying the wrapper and the communicator. The Farm is created as shown in line 4 by specifying the wrappers and respective communicators. Then, the final data stream can be modeled by building these blocks (sequential wrappers and patterns), as presented in line 9. The final parallel activity graph connects a Stage to a Farm pattern (with multiple blocks already inter-connected internally) and to the last stage. Finally, the method `Start()` (line 10) computes the complete parallel activity graph and schedules the MPI ranks. Each MPI rank will be responsible for processing a sequential wrapper.

Additionally, as show by Listing 4.3, the `Farm` object supports the some customization options. Regarding Farm's scheduling, the default option is round-robin. However DSParLib implements on-demand scheduling, which can improve load balancing if the network is not a bottleneck, especially when Workers have a different computational load or when data stream items have unbalanced computational complexity. The messages are distributed on demand as soon as the Worker finishes the previous computation. In the following, we show the customization that can be triggered:

- `SetWorkerReplicas(int)` to set the integer number of parallel Worker replicas;
- `SetCollectorIsOrdered(bool)` to enable ordering constraints in the Collector if set to true;
- `SetOnDemandScheduling(bool)` to enable on-demand scheduling if set to true.

4.1.4 Planning Rank Assignment

Ahead of the stream processing execution, DSParLib must decide which Pipeline Stage will be executed by a given MPI rank. This section describes the process of assigning DSParLib building block wrappers to actual MPI ranks.

In a Farm pattern, by default, the Emitter and Collector are placed as neighbors (Emitter on rank 0, Collector on rank 1), and parallel Workers range between 2 and $2 + \textit{degree\_of\_parallelism} - 1$. If the default MPI process allocation is used, rank 0 and 1 will be placed in the same cluster node equipped with a multi-core processor. Considering the Emitter and Collector are network or disk I/O intensive, both processes may have degraded performance since they would compete for limited resources. However, the user can change it by providing their custom `hostfiles` with different allocation configurations. In fact, we created custom `hostfiles` during our experiments with DSParLib, where we delimit that the first two nodes have only one slot each. Therefore, the Emitter and Collector are placed on isolated nodes, while computing nodes (Workers) are placed on the remaining nodes.

In a Pipeline pattern, by default the stages have their rank matching their position in the Pipeline. When combining a Pipeline with Farm, each Pipeline stage will be dislocated according to the graph topology. Each DSParLib building block has information about the stage's position and the number of precedent parallel processes. We define the information used to provide processes' ranks as input and output offsets and the total number of processes. This information is later used to assign the MPI process ranks to DSParLib building blocks. In this case, custom `hostfiles` are more challenging to provide since Emitters and Collectors can be assigned to different MPI ranks depending on their graph position.

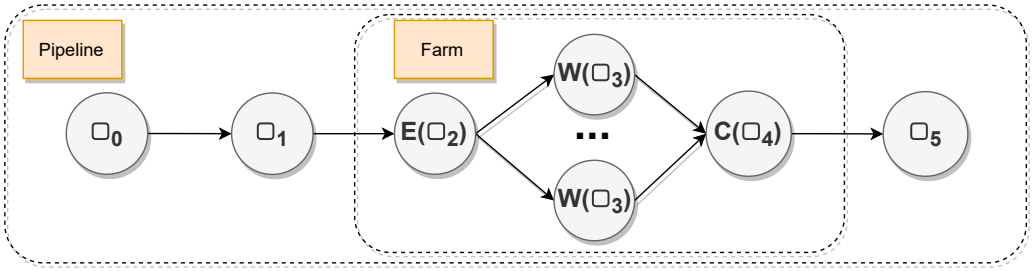


Figure 4.4: Parallel activity graph third example [55].

As example, let us define the parallel activity graph depicting a Pipeline and Farm composition: $pipe\{\square_0, \square_1, farm[E(\square_2), W(\square_3), C(\square_4)], \square_5\}$. The activity graph example is illustrated in Figure 4.4. By default, the Farm pattern communicates its input and output offsets as 0 and 1, and total number of processes as $2 + degree\_of\_parallelism$. Since there are other building blocks (stages $\square_0$ and $\square_1$) before the Farm pattern, these offsets are used to calculate the actual ranks to which the Farm will be assigned. For example, if two Pipeline stages precede the Farm, the input offset 0 is summed to 2. Consequently, $\square_0$ and $\square_1$ are positioned at rank 0 and 1, while the Farm has its Emitter on rank 2, Collector is rank 3, and parallel workers from ranks 3 to $3 + degree\_of\_parallelism - 1$. Finally, $\square_5$ is positioned at the end of the Pipeline. This means that DSParLib assigns the MPI ranks based on the final parallel activity graph.

4.2 Application Parallelizations with DSParLib

In this section, we implement three parallelization strategies with DSParLib: a Pipeline with three, five, and six stages. Specifically, we focus on the complex Ferret stream processing application from the PARSEC benchmark suite [16]. The application detects similarities between video, audio, and image files [16, 39]. The Ferret application has an original parallelization version targeting shared-memory using the Pthreads library [16]. In this version, the authors parallelized Ferret using a Pipeline parallel pattern. There are six pipeline stages, two of which are responsible for loading and collecting

the data, while the other four stages are for computational processing: Segmentation, Extraction, Vectorization, and Ranking.

Our parallelization is based on the original Pthreads version. Therefore, the first distributed version we implemented with DSParLib is a pipeline containing four computational stages, as shown in the bottom part of Figure 4.5. Note that the four computational stages are stateless. Therefore, we can replicate them to increase the degree of parallelism up to the maximum degree available on a target machine. This version was the most difficult one to implement because communicating data in Ferret is complex. The custom `struct` we implemented for communicating data has more than 20 members, varying from integers, pointers using 1 or 2 dimensions, and custom data types such as Ferret's CASS types (Content-Aware Search System). Other complexities rely on non-contiguous data and nested data structures.

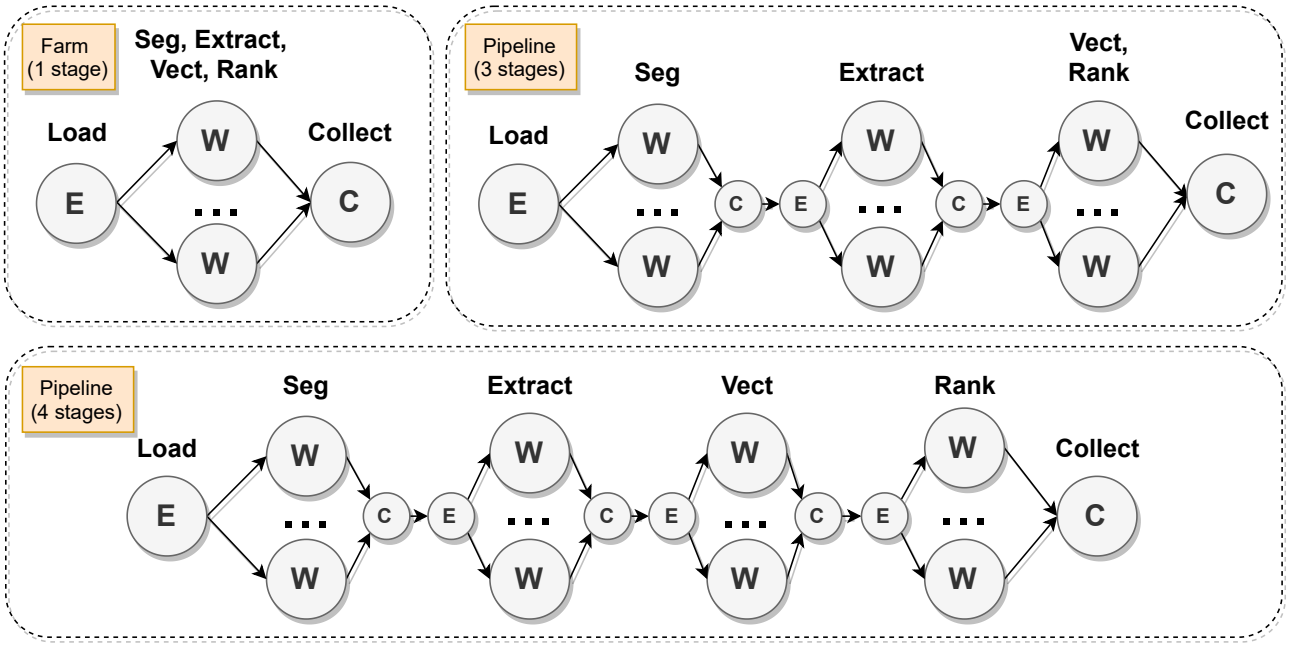


Figure 4.5: Parallel activity graph of the Ferret parallel versions [55].

To design the parallel activity graph in Ferret, the DSParLib parallel version used the composition of Pipeline and Farm parallel patterns. We created Farms to wrap the parallel Workers, as represented in Listing 4.4. Note that the Farms have a different number of parameters. The first and last Farms do not include communicators at the beginning and end, respectively. Then, all Farms that connect with others implement extra communicators. For example, Farm 0 communicates with Farm 1. Therefore, Farm 0 adds an extra communicator at the end, while Farm 1 adds an extra communicator at the beginning. Also, all Farms implement at least one Emitter or Collector stage using an empty stage. DSParLib leverages the structured parallel programming paradigm, which provides different parallel patterns that programmers can use. Therefore, parallelism cannot be designed in an ad-hoc approach. The Farm parallel pattern must consistently implement a

scheduler and a gatherer, namely Emitter and Collector. For example, a replicated stage cannot directly communicate with another replicated stage without a proper scheduling protocol. An optimized strategy, like the all-to-all communication model, can be implemented in the future. This means that each Worker from the previous stage has multiple messages passing channels to each Worker from the subsequent stage. For now, DSParLib's strategy complies with the Farm parallel pattern, clearly indicating Emitter, Worker, and Collector. Sometimes applications have natural Emitter and Collector stages, like the Load and `Collect` computations of Ferret (refer to Figure 4.5). However, when no sequential application code fits in, we use empty stages without computational processing. These are *empty stages* because they do not process anything. Instead, they simply forward the messages received from the previous stage.

```
1  auto comm = dspar::SenderReceiver<... >();
2  EmptyStage <task>  E, C;
3
4  auto farm_0 = dspar::Farm(Load, comm, Seg, comm, C, comm);
5  auto farm_1 = dspar::Farm(comm, E, comm, Extract, comm, C, comm);
6  auto farm_2 = dspar::Farm(comm, E, comm, Vect, comm, C, comm);
7  auto farm_3 = dspar::Farm(comm, E, comm, Rank, comm, Collect);
8
9  Pipeline pipe;
10 pipe.Add(&farm_0);
11 pipe.Add(&farm_1);
12 pipe.Add(&farm_2);
13 pipe.Add(&farm_3);
```

Listing 4.4: Example of Pipeline and Farm composition.

Alternatively, we have implemented an additional parallel and distributed version with DSParLib. The most challenging part of the message passing we discovered is communicating the `Vect` stage results to the Rank stage. The information about the correct memory size allocated for each data item was only found by looking deep into Ferret's source files. Therefore, in our second version, we combine the `Vect` and `rank` stages into a single one. The resulting parallel activity graph can be seen on the top right-hand side of Figure 4.5.

Finally, we developed one last version that implements the Farm parallel pattern without composition, as illustrated in the top left-hand side of Figure 4.5. This version contains a single computational stage (Worker) obtained by merging the Seg, `Extract`, Vec, and Rank stages into a single one. The message passing in this version is significantly more straightforward than the other DSParLib versions. We only communicate the data items from the Emitter to the Worker and later from Worker to Collector. Intermediate data are not sent over the network because the computation stays in the same node and is performed locally using shared memory. As discussed by the authors from [39], Ferret's stages are not well balanced, which limits further scaling. If the computing stages

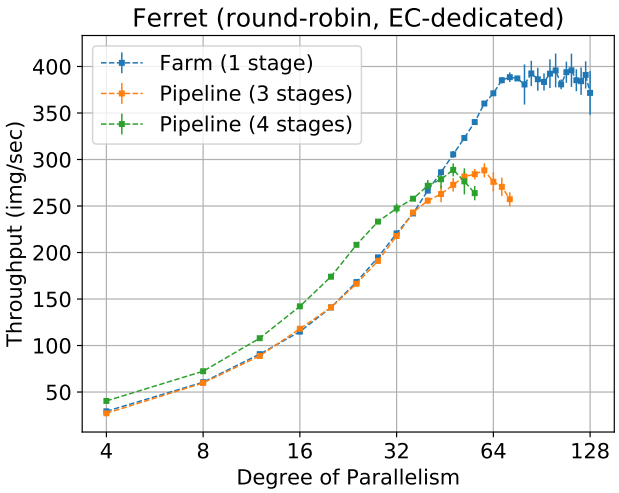
are replicated using the same factor, the unbalancing problem remains, and resource exploitation is not optimized, resulting in performance losses. This will be shown in the next Section 4.3 when we discuss Ferret's results.

4.3 Performance Evaluation in DSParLib

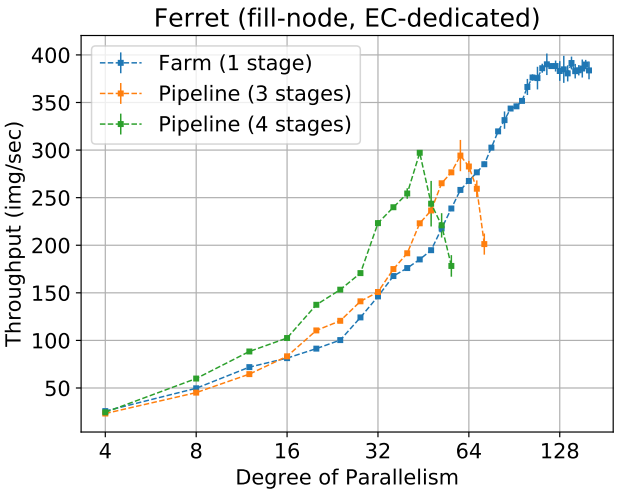
In this section, we aim to assess the DSParLib's scalability with Ferret employing the semi-arbitrary pattern composition of Pipeline and Farm. Ferret application exhibits complex data communication through message passing and data flows. We implemented three different distributed and parallel versions according to the parallel activity graphs illustrated in Figure 4.5. The experiments were executed on a cluster using eight computing nodes. Each node was equipped with 2 Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz (totaling 12 cores and 24 threads) with 24GB of RAM memory. The nodes were connected via Gigabit Ethernet and InfiniBand QDR 4x (32GBit/s). The operating system was Ubuntu 16.04 64 bits with kernel 4.4.0-146-generic. The MPI version was OpenMPI 1.4.5. The applications were compiled with GCC 9.3.0 using -O3 optimizations. We conducted tests using the round-robin and fill-node allocation strategies combined with different Emitter and Collector configurations:

- `round-robin, EC-dedicated`: This configuration uses a strategy that allocates processes in a round-robin fashion for each available node. Emitter and Collector have dedicated nodes.
- `fill-node, EC-dedicated`: This configuration uses a strategy that allocates processes in one node and only moves forward when the first node fully allocated. Emitter and Collector have dedicated nodes.
- `fill-node, EC-shared`: This configuration uses a strategy that allocates processes in one node and only moves forward when the first node fully allocated. Emitter and Collector share the same node.

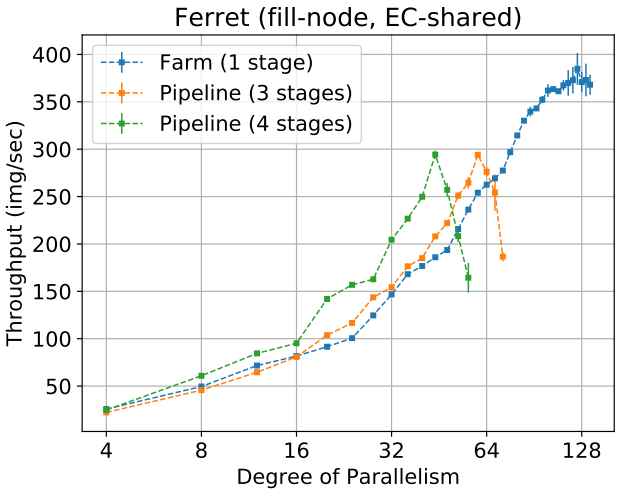
We tested the PARSEC Ferret benchmark with the default native workload composed of 3.500 images. The performance results are shown in Figure 4.6. The x-axis depicts the degree of parallelism, representing the number of working processes, excluding: Emitter, Collector, and empty processes. The y-axis shows the throughput in images per second. We depict all results until they achieve a peak and begin decreasing performance. There are two different explanations for that: (1) the workload size limits further scaling when reaching higher degrees of parallelism, which explains the limited scalability observed in Farm implementations; (2) both pattern composition versions (Pipeline) stop scaling long before that since they have reached a point where they are using all available



(a) Ferret - round-robin, EC-dedicated



(b) Ferret - fill-node, EC-dedicated



(c) Ferret - fill-node, EC-shared

Figure 4.6: Ferret evaluation in DSParLib with different processes allocation [55].

cores. For example, at the degree of parallelism 2, there are actually 12 ($6 + 2 * 3$) processes running for the Pipeline with 3 stages and 16 ($8 + 2 * 4$) processes for the Pipeline with 4 stages [1].

To help understand the results from the graphs, we prepared Table 4.1 showing the best throughput measured in each version and configuration. As expected, the Farm version scales better than the other versions and only stops scaling when the workload size becomes a problem. In the best-case scenario, the Farm version has up to 37% higher throughput than the other versions. For both the Pipeline versions, we observed that the maximum throughput is equivalent between themselves and varies less than 1%. Considering that the Pipeline (3 stages) version communicates fewer data because we have

[1] Refer to Figure 4.5 to understand the relation between the number of processes and the parallel activity graphs.

merged the last two stages avoiding a significant amount of communication, we expected this version to be slightly faster.

Table 4.1: Best throughputs measured in Ferret application [55].

Ferret Version	Metrics	Farm (1 stage)	Pipeline (3 stages)	Pipeline (4 stages)
round-robin EC-dedicated	Degree of Parallelism	112	60	48
	Number of Processes	114	186	200
	Throughput (img/sec)	395.88	288.37	288.88
	Std. Dev.	17.87	7.60	6.99
fill-node EC-dedicated	Degree of Parallelism	140	60	44
	Number of Processes	142	186	184
	Throughput (img/sec)	391.48	294.24	297.16
	Std. Dev.	6.75	16.42	2.80
fill-node EC-shared	Degree of Parallelism	124	60	44
	Number of Processes	126	186	184
	Throughput (img/sec)	384.82	294.07	294.29
	Std. Dev.	16.72	2.20	4.23

Although the parallel versions differ in performance gains, the results of different process allocation strategies are similar among the parallel versions. As can be seen in Table 4.1, The Farm version achieved the best throughput of 395.88 in *round-robin EC-dedicated*, while the worse version is *fill-node EC-shared* with 384.82 throughput. In the Pipeline versions with three and four stages, the results swapped, where *fill-node EC-dedicated* achieves the best throughput and *round-robin EC-dedicated* achieves the lowest throughput. However, if we observe the graphical representation of the same results in Figure 4.6, we can see that *round-robin EC-dedicated* depicts the smoothest curve (Figure 4.6a), while others achieve a peak and quickly start decreasing throughput (Figures 4.6b and 4.6c). These results reveal new insights about distributed stream processing using MPI. We employ this knowledge later to prepare the experiments for this master's thesis.

4.4 DSParLib Limitations and Improvements

In this section, we provide a summary of our research targeting DSParLib. After investigating DSParLib's runtime system, studying its API, and running a set of experiments, we detected some limitations and propose improvements. Our goal with this study is to implement a runtime system with enough flexibility to enable support for dynamic process management. Currently, DSParLib's runtime system is almost static, and any modification in the number of processes would break the code. In the following we describe DSParLib's limitations while proposing optimized strategies.

4.4.1 Process allocation strategy

Limitation: One of the lower-level components of DSParLib is the node allocation strategy. This strategy decides how every DSParLib building block (block of sequential code) is assigned to actual MPI parallel processes. The current implementation uses a static strategy to assign building blocks to MPI processes. We have provided a precise explanation of the node allocation strategy in Section 4.1.4. In short, each assigned MPI rank reflects the role of the processes in the main Pipeline execution. For example, the Farm's Emitter and Collector nodes that perform the heavy tasks of scheduling and accumulating are always assigned at the early ranks (0 and 1). Later, in the experimental evaluation, these ranks are allocated in two exclusive nodes so that they avoid sharing computational resources and escape resource contention. One of the main problems of DSParLib's strategy is that it considers the offset to assign new ranks. Therefore, DSParLib cannot support new added processes, as they cannot be inserted in the Pipeline execution during execution time. DSParLib would have to stop the application, reconfigure the ranks and then restart the execution.

Improvement: When we studied dynamic process management in MPI, we concluded that ranks assignment cannot be static. In addition, the mechanisms of assigning ranks to processes must support non-consecutive ranks. For example, in DSParLib if a Farm is executed with four processes, Emitter is rank 0, Collector is 1, and Workers are 2 and 3. A dynamic strategy must support non-consecutive ranks, i.e., Emitter being 1, Collector being 3, and Workers being 0 and 2. More recent versions of MPI enable programmers to organize processes in groups. Then, a better strategy would be to employ such mechanisms and later create the inter-communicators based on the group ranks during execution time.

4.4.2 Intra- and Inter-Communicators

Limitation: MPI provides two different communicator categories for usage. An intra-communicator is a single group of local processes, and an inter-communicator refers to a pair of groups. By default, MPI creates a large global intra-communicator called `MPI_COMM_WORLD` that enables all processes of this global group to communicate. That is not ideal for adaptability support since ranks are statically assigned to Pipeline stages. In that case, it is best to create different intra-communicator groups, each being responsible for a stage of the pipeline. Then, processes can be added and removed from this communicator without impacting in the global Pipeline communicator. However, DSParLib runtime system creates all processes in a single global intra-communicator. That is one

of the main reasons it uses a static strategy to assign ranks because the global communicator rank is used to reference the process. It can never be lost or modified; otherwise, processes can no longer communicate, and DSParLib obtains a deadlock state.

Improvement: Our investigation concluded that a better design principle to adopt as a strategy is creating MPI groups for the pipeline stages. A group can have one or more processes. Each group assigns local ranks to the processes. To communicate between the local groups, each stage is responsible for creating its own inter-communicators. The first and last stages of the pipeline create a single inter-communicator to send messages to the next and previous stages, respectively. The middle stages create two inter-communicators to communicate with the previous and subsequent stages.

4.4.3 Parallel Patterns

Limitation: DSParLib provides a high-level interface with two parallel patterns and their semi-arbitrary composition. These patterns are the Pipeline and Farm. During the investigation studies in DSParLib regarding the parallelization of applications, we found that the use of the Farm pattern shows limitations that impact the performance. For example, a pipeline stage cannot be directly replicated to execute in parallel. The user must implement the Farm pattern and replace the default stage with the new Farm. However, The Farm pattern requires the implementation of the Emitter and Collector. In case we have multiple parallel stages, each one becomes a Farm, and their Emitter and Collector are introduced as extra processes. In fact, since they do not contain computation in most cases, they only receive a message and forward it. That increases the latency and decreases the overall throughput; also, it can become a bottleneck in network communication.

Figure 4.5 illustrated this example when we implemented the Ferret application with DSParLib and extra empty stages must be included to be in conformance with structured parallel patterns. We implemented the Ferret application using three different graph configurations for DSParLib. The version using a single stage achieved the best results, while the version with 4 stages showed the worse results. Although some of the performance degradation comes from the application having unbalanced stages, a considerable parcel of the higher overhead is due to the extra DSParLib's Emitter and Collector stages.

Improvement: With this work, we concluded that a better approach is to allow the Pipeline stages to replicate their processes without having to introduce a Farm parallel pattern. Therefore, the Farm should be removed and replaced with another strategy that allows parallel processes to communicate between themselves. Since we introduced the notion of multiple intra-communicators in the previous section, they can now be exploited

to communicate in between the stages using local sources and target ranks. Ideally, the processes of each stage should implement an all-to-all communication style.

4.4.4 Data Communication

Limitation: DSParLib provides different levels of abstraction so that data communication between the processes is hidden from the developer. We previously described the mechanisms in Section 4.1.2. We showed that data communication is abstracted by the `SenderReceiver` class. Internally, DSParLib's runtime employs this class in every communication performed by the library. It works as a custom built-in serialization template. In fact, it is deeply coupled with other mechanisms and cannot be easily rebuilt to introduce modularity regarding serialization.

Improvement: During our studies, we observed that this design choice does not contribute to our goals. We understand that DSParLib targeted the application developer. Therefore, every feature of its interface aims at increasing the programmability levels. On the other hand, we concluded that to target system developers as well, the data communication internal representation should not be in terms of a higher-level class such as the `SenderReceiver` class. In fact, later when we present the parallelization studies and compare DSParLib's version with other parallel versions, we show that there are easier ways to implement data communication than using the `SenderReceiver` abstraction. For instance, the data communication interface can expect only the pointer of the data location and its size.

4.5 Summary

In this section, we summarize the investigation studies in DSParLib's runtime system. We focused on the process management functionalities and performance capabilities of DSParLib. We have investigated DSParLib internals and researched the strategies that provide programming abstractions on top of MPI. In addition to that, we have implemented different parallel versions on a complex streaming application using DSParLib features. The results revealed that DSParLib delivers improved programmability aspects while achieving good performance [55]. However, the investigation studies also revealed limitations regarding dynamic process management capabilities. We described a list of limitations in the previous Section 4.4, in which most of them prohibit adding and removing processes during execution time. We concluded that the existing strategies and programming abstractions implemented in DSParLib's runtime system cannot be modified to support adaptability. Moreover, during the investigation, we discover a simpler

programming model to deal with distributed stream processing. This refers to the limitation described in Section 4.4.3, in which we propose to remove the Farm pattern and enable all-to-all communication.

The initial main goal defined when we proposed this work was to investigate DSParLib, re-design its processing engine, and provide mechanisms for supporting dynamic parallelism adaptability. However, the studies revealed that DSParLib's processing engine cannot support adaptability and its programming model shows drawbacks. Unfortunately, DSParLib did not satisfy the requisites for this work, and will not be used in this thesis. Therefore, in this work, we must design a new distributed stream processing engine from the beginning.

5. MPR: FRAMEWORK FOR DISTRIBUTED STREAM PROCESSING

In this chapter, we introduce MPR (Message Passing Runtime), a framework designed and implemented on top of MPI for self-adaptive distributed stream processing [1]. The main goal of MPR is to provide a runtime system on top of MPI to ease the implementation of self-adaptive distributed applications in C++. Moreover, in this work, we focus on expanding the knowledge of dynamic process management on distributed stream processing applications. Throughout the chapter, we present MPR's details regarding its architecture, self-adaptive features, and API. Also, we discuss the optimizations we enabled in MPR, such as transparent serialization, load-balancing mechanisms, ordering, back pressure, and process coordination. This chapter is organized as follows. First, we present MPR's design goals and principles in Section 5.1. Then, Section 5.2 briefly presents an overview of MPR's architecture. From this point on, we discuss each component of its architecture: Starting with the communication layer in Section 5.3, then MPR's processing engine in 5.4, adaptability support in Section 5.5, and finally its high-level API in Section 5.6.

5.1 Design Goals

In this section, we introduce the main goals of MPR. Dynamic process management in MPI is complex and is rarely exploited by programmers despite being available since MPI 2.2 (2009) [44]. We did an extensive study on MPI newer specifications to design a flexible, efficient, and portable runtime system that supports adding and removing processes during execution time. In addition, we gathered experience with the stream processing paradigm in our past works [65, 58, 54, 55, 69], which we have applied to this work on MPR. We draw upon this experience to address some of the relevant concerns, which we describe in the following.

1. **Pipeline pattern:** A fundamental stream processing parallel pattern is the Pipeline. Almost any computation can be expressed in terms of a multi-stage Pipeline. Therefore, MPR should support the Pipeline pattern. We define the Pipeline stages that should be supported in MPR, which can be of three types: Source, Compute, and Sink. We briefly categorize them: (1) *Source*: Represents the computational logic that schedules data into further Pipeline stages. A *Source* has no predecessor stages and usually is the first stage of the Pipeline. Data can be fed to the Pipeline from sensors, cameras, stored data, and other producers. (2) *Compute:* This stage is responsible for processing the streaming data. The computation can be any block of

[1] MPR is available at: https://github.com/GMAP/MPR

valid C++ code. The computation is applied over each stream item received by this stage. A *Compute* receives data from a predecessor stage (*Source* or *Compute*) and forwards it to a subsequent stage (*Sink* or *Compute*). (3) *Sink*: Represents the computational logic that accumulates results into any location. The results can be of any format, such as reports for human inspection or interconnected to other devices that will take action accordingly.

2. **Data serialization:** Throughout the years, many different approaches for marshalling and unmarshalling data have been proposed. Some of them focus on human readability, while others target performance. MPR should allow different types of serialization to be employed since they vary depending on the programmer's preference. For that, default data serialization for a stream item should be described as pair containing a reference to data and the data size. Additionally, MPR should provide standard built-in programming abstractions for data communication that application developers can leverage without needing third-party libraries. These programming abstractions may support or not zero-copy serialization.

3. **Portability:** As computing systems evolve into heterogeneous environments, supporting these different architectures and platforms becomes mandatory. Nowadays, clusters composed of heterogeneous architectures are becoming commonplace. MPR should leverage MPI's portability for supporting architectures and platforms from different vendors. MPR should exploit the MPI's portability to allow a program written using a single programming model to execute on a wide range of architectures.

4. **Ad-hoc parallelism:** Eventually, MPI's portability will not support an architecture or platform, or will not be able to efficiently exploit the underlying parallel resources. In these cases, new programming models that can improve resource exploitation may be used along with MPI to write parallel applications. Therefore, MPR should allow programmers to leverage different parallel programming models. For instance, it should not prevent the programmer from spawning system threads or offloading computational logic to heterogeneous devices such as GPUs.

5. **Third-party libraries:** MPR should not demand the programmer to install and configure third-party libraries other than an MPI implementation (i.e., OpenMPI and MPICH) in order to work properly. Libraries and frameworks are constantly changing and their updated versions may not have backward compatibility. We consider it a drawback when other researchers or programmers are inhibited from using a tool because they cannot install or execute it. Sometimes, understanding the problem becomes challenging when the tool requires multiple dependencies. For instance, the problem may be incompatibilities between versions or a missing installation step due to unclear documentation.

6. **Modularity:** MPR should isolate the different concerns into modules. Our intent with this design principle is to enable different programmers and researchers to focus only on a specific part of the framework related to their domain of expertise. State-of-the-art stream processing systems are large projects that usually compose hundreds of thousands of code lines and where multiple mechanisms are integrated. For an average programmer unfamiliar with the tool, implementing modifications in such systems demands a steep learning curve that sometimes takes years for being able to implement an algorithm in these distributed systems.

5.2 MPR Architecture Overview

In this section, we provide more details about the MPR framework. We designed the framework to support the features we need in our distributed stream processing system. Figure 5.1 is organized according to three abstraction levels. Note that we provide a clear separation between the application and system developer concerns. Application programmers are those interested in implementing stream processing systems. They want to use our framework to implement real-world streaming applications that execute efficiently and are able to scale in distributed environments. The framework's topmost abstraction level provides the required mechanisms and interfaces so that application developers can write code for streaming systems without exposing low-level architecture details. For example, the framework provides an API close to what programmers are familiar with in state-of-the-art DSPS.

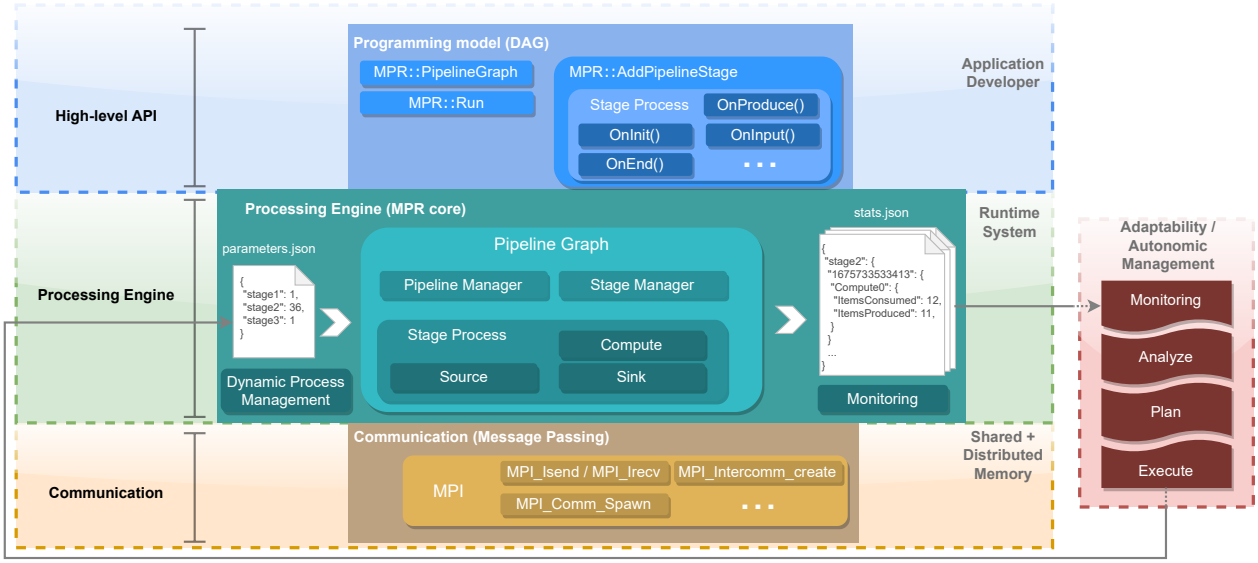


Figure 5.1: MPR architecture blueprint.

In the second level, we implement the processing engine responsible for orchestrating the computation and providing support for synchronizing all the processes during

reconfiguration, either to spawn or remove processes. It offers lower-level programming abstractions and interfaces that can be used by system programmers. Depending on their expertise, they can focus on a specific research issue to provide further optimizations and features to application programmers. This framework level represents one of the main technical contributions of our work since we want to enable other researchers to implement, test, and compare against others their original techniques, algorithms, and strategies. For instance, the self-adaptive strategy is implemented as an isolated module that interacts with Json files. Then, periodically MPR runtime system reads the Json reconfiguration file and generates updated Json files with execution statistics. This way, although MPR runtime system is implemented in an HPC programming language, namely C++, we allow system developers to write algorithms in any language that can interact with Json files. Consequently, higher-level programming languages such as python and R can be used, which give access to a wide range of libraries and packages that implement numerous data science and machine learning algorithms for implementing better prediction models. More details about the MPR's self-adaptive module will be presented later in this work.

Finally, the bottom-most level represents the communication and transport layer of MPR framework. We implement the parallel processes' communication using message passing. Specifically, we use the MPI specification due to its performance and portability. We use newer MPI specifications which give the dynamic capabilities MPR needs to deal with dynamic process management. Also, we implemented a synchronization protocol to coordinate processes into a global state during reconfigurations.

In the following sections, we describe each framework level from Figure 5.1. We follow MPR levels in a bottom-up fashion. Therefore, in Section 5.3, we start by introducing the communication level and explaining which are the MPI interfaces we leverage in MPR. In Section 5.4, we introduce the MPR processing engine that supports self-adaptive distributed stream processing. Then, in Section 5.5, we present the autonomic management API we designed to support application adaptability. Finally, in Section 5.6, we describe how application programmers can implement distributed stream processing applications using MPR.

5.3 Communication using Message Passing

In this section, we present the MPR communication layer technology. The Message Passing Interface (MPI) is the leading parallel programming model for distributed computing in HPC and scientific applications [4]. We choose MPI as our communication layer due to its performance and portability. We introduced MPI in the Background Sec-

tion 2.3.1, and now we extend the discussion focusing on the MPI interfaces employed by MPR.

Initially, MPR will only support the OpenMPI implementation of the MPI specification, which is a consolidated and extensively documented MPI implementation with a large community of programmers and researchers. However, future work will involve testing other MPI implementations, as there may be differences between them despite sharing the same interface. During our tests, we encountered several undesired effects using OpenMPI, which we plan to investigate further. Fortunately, most of the errors (i.e., segmentation fault) occurred at the end of application execution after the correct result was already generated. These failures were mostly internal to OpenMPI (i.e., OPAL, ORTE, OMPI) when combined with SLURM. We noticed that many of the errors were related to `MPI_Comm_spawn` and `MPI_Finalize`. In fact, almost all errors happen when MPR has already finished executing the Pipeline and invokes its last line of code before exiting, namely `MPI_Finalize`. Thus, it does not interfere with our experiments. Also, while investigating the spawning of new processes, we discovered some bugs that were recently reported in OpenMPI's Github at the time this thesis was written. We plan to address these issues as part of our future work.

5.3.1 Data Communication

MPI exposes a wide range of communication interfaces, they can be point-to-point, one-side, and collective communications. For MPR's first version, we selected point-to-point data communications and manually implement the synchronization when reconfiguration is active in the system.

We chose not to use collective communications in our adaptive stream processing approach because they can be unsuitable and even harmful in certain parts of MPR's protocol since they can lead to deadlocks. Collective operations are blocking and require synchronization between all processes involved in the communication, meaning that all processes in a given communicator must call the same collective function with the same parameters for the operation to succeed. Although MPR already uses some blocking operations, such as `MPI_Comm_spawn`, it does not use them for communication in the normal Pipeline execution. Problems can arrive when both MPR communications (data protocol and configuration protocol) use blocking operations without isolation. For example, a deadlock could occur if some processes are blocked in a collective communication using the data protocol while others are blocked in a collective using the configuration protocol. To avoid such issues, we decided to use non-blocking communication operations for data communication that allow us to maintain progress in the system while avoiding deadlocks.

We did not use one-side communications because they are too complex to deal with in a self-adaptive distributed stream processing system. At the beginning of MPR's design, we attempted to employ remote memory access (RMA) calls. However, we already noticed that it might not suit MPR's reconfiguration protocols. For example, depending on the reconfiguration action, we destroy all communicators and recreate them. So, we would have extra overhead to recreate the shared memory regions and the risk of losing data from some of them.

To design MPR protocols, we have opted for point-to-point communications, also known as two-sided communications. Specifically, we make use of non-blocking MPI sends and receives, such as `MPI_Isend` and `MPI_Irecv`, which return immediately (do not block) while MPI selects the most efficient method for message transfer. To check if the message has been sent or received, we employ `MPI_Wait`. In the current version of MPR, we implement `MPI_Wait` immediately after the non-blocking communication, except for request messages. The same result could have been achieved by simply using `MPI_Send` and `MPI_Recv`. However, we have designed MPR this way, as we plan to simply shift the `MPI_Wait` in a future version, allowing processes to compute something instead of immediately being blocked. This can have a direct impact on performance. While non-blocking optimizations are not explicitly utilized in the current version of MPR due to time constraints, we have constructed the MPR runtime system to already accommodate such optimizations in the future.

5.3.2 MPR Groups and Communicators

The MPI specification defines two types of communicators: intra-communicators, which include all the local processes within a single group, and inter-communicators, which connect two distinct groups of processes. By default, MPI programs are created within a single large intra-communicator called `MPI_COMM_WORLD`, which allows all processes to communicate with each other. While most MPI programs use the global communicator to exchange messages, inter-communicators become more relevant when dealing with dynamic process management in MPI. For instance, the `MPI_Comm_spawn` interface spawns processes in a new intra-communicator that is different from `MPI_COMM_WORLD`. As a result, each new MPI spawn function call creates a new intra-communicator that is local to the processes spawned. To communicate between two distinct groups of remote processes using their local intra-communicator, an inter-communicator is required.

The MPI specification also introduces the concept of groups. An `MPI_Group` is an ordered set of processes that are assigned unique ranks ranging from 0 to `group_size-1`. MPI groups are based on the Set Theory, which enables operations such as unions, intersections, adding or excluding processes, and others. However, groups themselves cannot

be used for communication. To enable communication among the processes, more recently, MPI introduced an interface to create a communicator by passing the group as an argument, and vice-versa. This makes groups suitable for being used to organize processes, such as excluding specific ranks, adding new processes, intersecting two groups, and so on. Once the processes are organized in different groups, MPR employs these groups and creates communicators with them, which enables communication with other processes. When dynamic process management is required in our framework, MPR converts the communicators back to groups, applies the required modifications to the group, and recreates the communicator. This approach allows MPR to leverage the flexibility of MPI groups and communicators for process organization and communication, respectively.

An illustration of MPR's internal organization is depicted in Figure 5.2. The runtime has three organizational levels. At the topmost, a Pipeline Manager is in charge of processing important decisions, broadcasting critical events, and maintaining a consistent state of the Pipeline. Then, in the middle layer are the Stage Managers. They are essential to add scalability to the system since they intermediate the communication between possible thousands of computational processes and the Pipeline Manager. Also, they are responsible for processing local stage decisions. Finally, the Stage Processes are in charge of processing the Pipeline. Each rounded square represents a local MPI group, while circles are MPI processes.

MPR employs the notion of intra- and inter-communicators previously described to improve the runtime system internal organization. As shown in Figure 5.2, MPR has four types of groups: the Pipeline Manager group, a Stage Manager group, a Stage group, and a Global Pipeline Group. Each inter-communicator (communication between a pair of groups) is represented by a white arrow. During normal execution, processes only communicate via the white arrow connections. So, the Pipeline Manager communicates only with the Stage Managers. Each Stage Manager communicates with the Stage Processes of its corresponding stage. Also, the Stage Processes communicate via inter-communicators with their adjacent stages. During reconfiguration, processes may use the global Pipeline Group and its communicator for exchanging information, where arrows are not represented for simplicity. For instance, it allows the Pipeline Manager to communicate with all processes to send updated Pipeline information.

Since we are introducing the figure, we also present the other components that will be explained later in this thesis. Figure 5.2 depicts different Json files connected to the Pipeline Manager and Stage Managers. In short, the MPR runtime system connects to the real world using the `parameters.json` and `stats.json` files. In the former file, developers can set the number of processes for each Stage. The Pipeline Manager periodically reads this file to check if there are any modifications. We plan to add other parameters to the file in the future. Moreover, each Stage Manager produces a distinct `stats.json` file containing statistics about the stage's execution. Currently, we report the total number

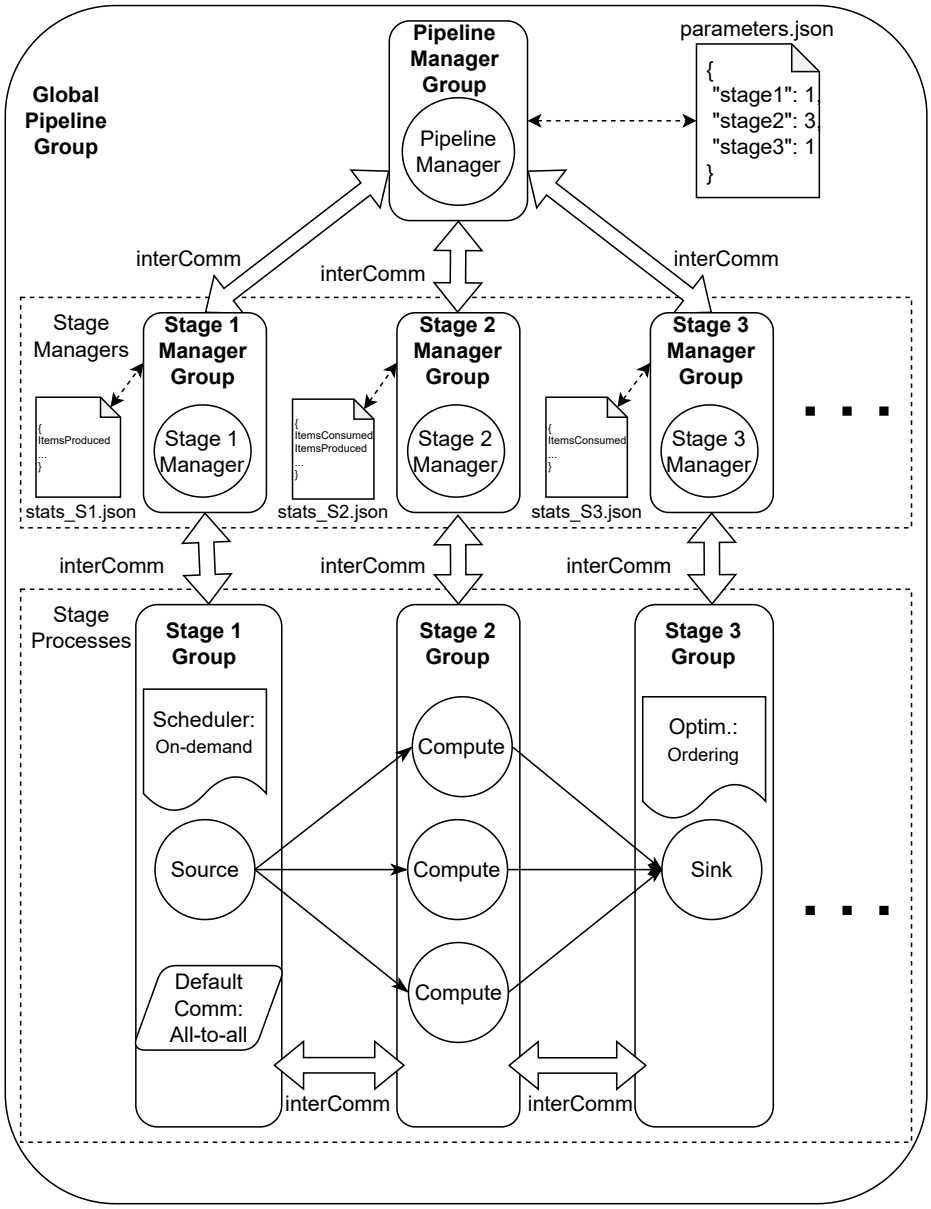


Figure 5.2: MPR's communication design overview.

of items consumed and produced by each stage in a given timestamp. In addition to the number of items, we also report the average of items produced, which is computed using the total number of items divided by the time interval.

5.3.3 Summary

In this section, we explain MPR's communication layer built on top of MPI. We introduced MPI concepts such as intra- and inter-communicators, MPI groups, and point-to-point or collective communications. Moreover, we sketched MPR's communication design in Figure 5.2. This section introduces some MPI notions and serves as a background for later explaining the MPR processing engine. Afterward, we refer to this Figure to explain

the synchronization protocol we implemented to coordinate the processes executing the Pipeline Graph. Note that we have not entered in too much detail about the MPI interfaces we use for spawning processes, creating groups and communicators, and others since we will cover them in the following section.

5.4 MPR Processing Engine

In this section, we present MPR's processing engine that orchestrates the execution of distributed stream processing applications. MPR implements different communication protocols to deal with configuration routines and normal stream data processing. The section is organized as follows. In Section 5.4.1, we introduce MPR's internal organizational structure. Then, Sections 5.4.2 to 5.4.6 elaborate on MPR's runtime system internal protocols. First, we present the configuration protocols that require distributed synchronization for adding and removing processes. Then, we discuss the data protocol used to exchange data during normal Pipeline execution. In Section 5.4.7, we introduce MPR's strategy to create processes and assign them to Pipeline Jobs. Subsequently, Section 5.4.8 explains the communication model and schedules adopted by MPR. Finally, Section 5.4.9 introduces the available programming abstractions to exchange data and how MPR's processing engine deals with the different message types flowing in the Pipeline.

Throughout this section, we include several code snippets to provide insight into the implementation of MPR using MPI. While this may seem daunting to some readers, we have chosen to present these code slices without many abstractions to aid others who are implementing MPI and may be struggling with its complexities. We recognize that many MPI interfaces are low-level and involve numerous parameters that can be challenging to understand. Therefore, we hope that others can take inspiration from our code to develop their own MPI implementations, as it can be difficult to find examples of less-common MPI interfaces. Our work is the first to introduce self-adaptive distributed stream processing, and we aim to share our implementation logic for MPR's processing engine to assist others in developing similar systems.

5.4.1 MPR Project Organization

This section outlines the internal structure of MPR's code and provides a brief overview of each MPR class, which will be elaborated on in subsequent sections. The current version implements more than 3.500 lines of code. Figure 5.3 illustrates a simplified representation of MPR's class relationships, although not all relationships, class variables, and functions are shown. This diagram is helpful for understanding the main components

of MPR's processing engine. The most important class is `PipelineGraph`, which implements the Pipeline pattern and allows programmers to construct their Pipeline graphs for streaming applications. Application code is organized into Pipeline stages and added to the `PipelineGraph` using `AddPipelineStage()`. Once the graph is fully assembled, invoking `Run()` starts the streaming application's execution.

In MPR's runtime system design, we opted for a partially object-oriented programming model, using polymorphism and inheritance only when strictly necessary. Then, we created three helper classes to encapsulate the computational logic and context-specific information: `PipelineInfo`, `ProcessInfo`, and `StageInfo`. The former two classes are initialized with the `PipelineGraph`, while the last is initialized only with the `StageProcess`. Once these objects are created, their reference is passed to all classes that need to utilize their information. Specifically, we pass the references of `PipelineInfo` and `ProcessInfo` to all entity classes that interact with the `PipelineGraph`. Likewise, we pass the reference of `StageInfo` to all classes that execute application code.

This design allows for a clear separation of concerns, as each class is responsible for its own context-specific information and logic. It also enables efficient information sharing and avoids the need for complex inheritance relationships. Overall, this approach simplifies the codebase and enhances maintainability.

We provide a brief overview of the three helper classes that were created for our runtime system. The first helper class is called `PipelineInfo`, which contains information about the Pipeline graph, such as stage sizes, ranks in each stage, global communicator, and others. The second helper class is called `ProcessInfo`, which holds information about the MPI process, like its Pipeline Job and stage identifier. Although this class does not hold much information, it is included because we anticipate that future extensions to MPR's runtime system may require more information about the process itself, which can be stored in this class. The third helper class, `StageInfo`, is only relevant to the Stage Processes, as it contains information and computational logic for processes that execute application code. This class includes information such as input and output communicators, data buffers, ordering, and other important information.

MPR consists of three entities that are created at the beginning of Pipeline execution and remain active until the execution ends. These entities are the `PipelineManager`, `StageManager`, and `StageProcess`. Processes are assigned to these classes depending on the Pipeline Job they receive when executing `SetJobs()` from `PipelineGraph`.

The Pipeline Manager is responsible for coordinating the Pipeline execution and its Job is assigned to a single process (usually rank 0). Periodically, this process reads the `parameters.json` file and notifies all Stage Managers when reconfiguration is needed. Note that the Pipeline Manager only notifies the Stage Managers, and it is the responsibility of the Stage Managers to notify all Stage Processes if reconfiguration is required.

Each Pipeline stage has its own Stage Manager. These managers receive the status of all active Stage Processes, as well as the number of items consumed and produced by each Stage Process. The Stage Managers accumulate this information and periodically write it to the respective stage's `stats.json` file. Also, each time all Stage Processes have reported their status, the Stage Managers forward an outline of their results to the Pipeline Manager.

Each Stage Process is assigned to a Pipeline stage and executes the respective application code. Periodically, they report their status to the Stage Managers and receive a response indicating if any reconfiguration action is needed. The `PipelineManager`, `StageManagers`, and `StageProcesses` classes implement their own unique responsibilities in the Pipeline Graph execution and work together to ensure that the Pipeline functions as intended.

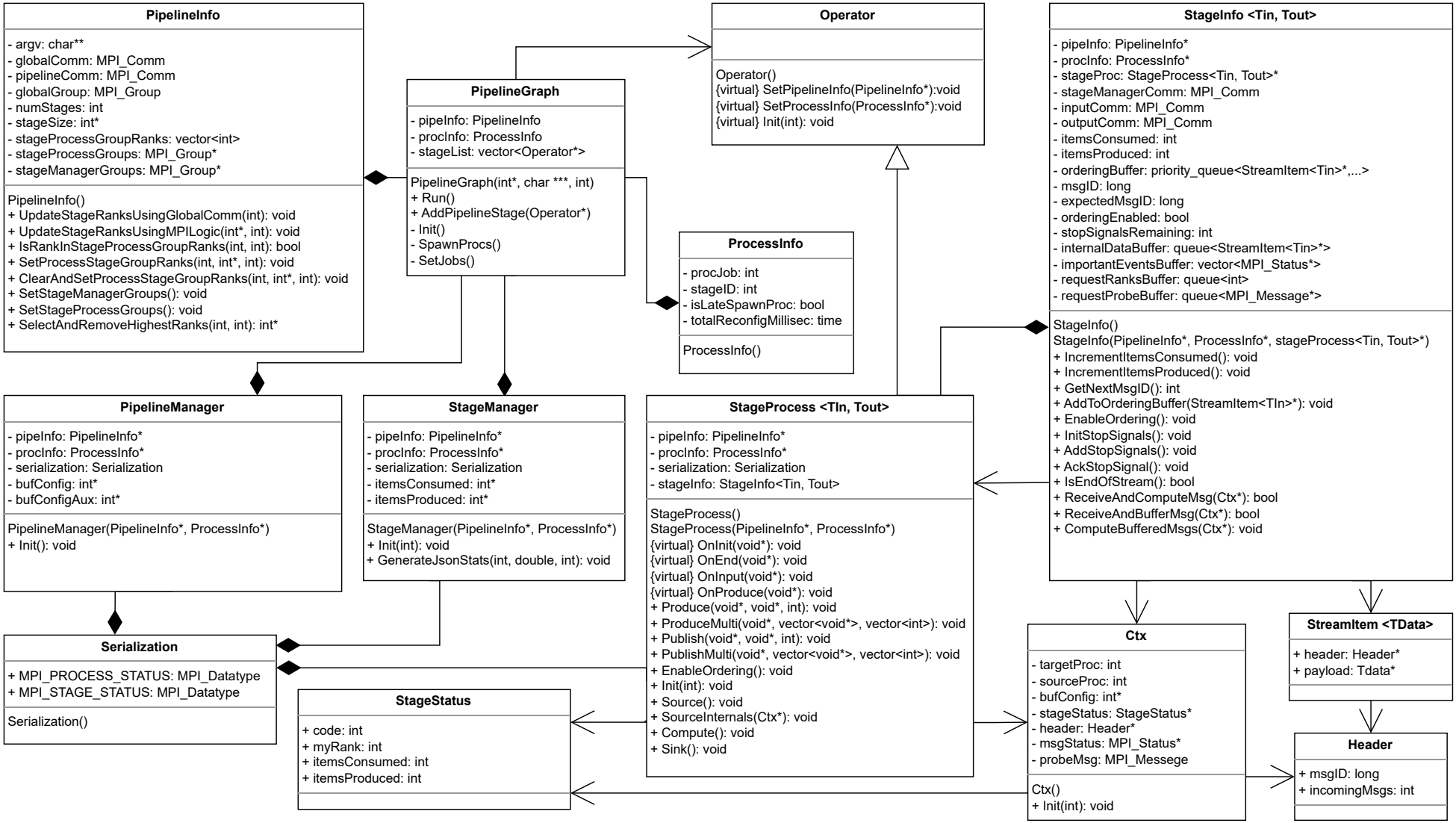


Figure 5.3: MPR's class relationship.

5.4.2 MPR Configuration Protocols

MPR implements two protocols to synchronize the Pipeline processes depending if the application is executing normally or is in reconfiguration mode. MPR's two existing protocols are (1) A *configuration protocol* to communicate Actions between the processes. This leader-based synchronization protocol ensures that processes synchronize on the runtime system's global state when the Pipeline enters reconfiguration mode. (2) A *data protocol* to exchange stream items between the processing stages. This protocol targets the normal execution of the Pipeline streaming application.

Foremost, in this and the following sections, we aim at presenting the *configuration protocol*. After, in Section 5.4.6, we introduce the *data protocol*.

It is worth noting that the *data communication* protocol is permanently active. The only reason it interrupts is that, eventually, the processes enter into a synchronization mode in which they exchange configuration Actions. Actions are synchronization routines/protocols the processes execute that are apart from normal Pipeline execution. For instance, an Action may tell processes to do something different such as adapting the number of processes, monitoring a local resource, or waiting in a barrier. In the following, we show a list of Actions currently implemented in MPR. The first three actions are used during normal pipeline execution, while the remaining four are used to support dynamic process management.

- **Action 0:** Notification to maintain the same course of action without any variations. Sent from Pipeline Manager to all processes and from Stage Manager to Stage Process. Requires two-sided synchronization.
- **Action 1:** Process alive notification. Processes also send their status containing the number of items consumed and produced. Sent from Stage Process to Stage Manager. Requires two-sided synchronization.
- **Action 2:** Process EOS (End of Stream) notification. Processes also send their status containing the number of items consumed and produced. Sent from Stage Process to Stage Manager. Requires two-sided synchronization.
- **Action 3:** Reconfiguration notification. Synchronization is required to *ADD* processes to the Pipeline graph. Sent from Pipeline Manager to Stage Manager and from Stage Manager to Stage Process. Requires global synchronization.
- **Action 4:** Reconfiguration notification. Synchronization is required to *BAN* processes from the Pipeline graph. Sent from Pipeline Manager to Stage Manager and from Stage Manager to Stage Process. Requires global synchronization.

- **Action 5:** Reconfiguration notification complementary to Action 3. All newly spawned processes receive information about which stages need processes and which stages have enough processes. Sent from PipelineManager to *NEW* Stage Processes. Requires partial synchronization.
- **Action 6:** Spawn notification. All *SPAWNED* processes receive information on how many processes are to be spawned. Requires partial synchronization.

This study assumes strong guarantees about the properties and nature of the distributed system, as fault-tolerance is not exploited. The focus are High-Performance Computing (HPC) clusters equipped with reliable high-speed and low-latency network interconnections. Moreover, MPR's runtime system bases on some guarantees provided by MPI. For instance, MPR leverages message ordering in the same communication between pairs of processes. This ensures that the messages received in the communication from process A to process B will match the order they were sent. Other types of ordering are not required by MPR's synchronization algorithm. Additionally, MPR operates without the need for globally synchronized clocks. Each process uses its own local clock, which does not need to be synchronized with other processes.

We implemented the *configuration protocol* as a heartbeat mechanism that periodically synchronizes the processes with Pipeline's global states for achieving global agreement. Moreover, we implemented our protocol in a bottom-up fashion. Therefore, the ones that start the communication are the Stage Processes, and they only communicate with the Stage Managers [2]. Then, the Stage Manager waits until it has acknowledged all processes and only then communicates with the Pipeline Manager. This way, MPR's Stage Processes can progress because the processes that fail to communicate may be detected and eventually dropped since their communication is isolated. If we had implemented the heartbeat mechanisms in a top-down fashion, every time a Pipeline Manager communicates with a faulty process, it would lead to a deadlock and halt the entire Pipeline.

When utilizing the *configuration protocol* for reconfiguration, specific requirements must be met for MPR to actively add or remove processes. Firstly, all processes must enter reconfiguration mode, which is accomplished in MPR by having the Pipeline Manager notify all Stage Managers, who then notify their respective Stage Processes. Secondly, all pending communication must be completed prior to any reconfiguration. To achieve this, any Stage Processes waiting to send a message must be matched by their respective receiving processes. Stream item messages are buffered rather than processed to minimize overhead. Finally, when a process is removed, all stream item messages must be computed before the process is effectively removed. Therefore, messages are

[2] Refer to Figure 5.2 to remind the hierarchy of MPR's Pipeline processes.

processed and sent forward in the Pipeline to ensure proper handling of all data without losing messages.

System programmers can leverage the *configuration protocol* synchronization algorithm already implemented in MPR for extending to new synchronization Actions. An example of a new feature could be supporting work-stealing. One could synchronize a group of processes and redistribute their workload in order to improve load balancing. Others may want to add a new feature to get additional performance metrics to debug the Pipeline Graph execution. Note that we already implemented different scopes of synchronizations. The most costly Actions are the ones for adding and removing processes since they require global synchronization. But other synchronizations enable employing simply point-to-point synchronizations between two processes.

5.4.3 Normal Execution Protocol

Figure 5.4 helps to illustrate the synchronization between processes in the *configuration protocol* during normal execution. The *data protocol* is not illustrated in the image for simplicity. Suppose a Pipeline processing a streaming application. After a period of time `T(n)`, the Pipeline execution halts, and processes enter into a synchronization stage. Then all Stage Processes (`P1` in the example) send `Action 1` (means the process is alive) to their respective Stage Manager (`SM1` in the example), which acknowledges and returns `Action 0` (do nothing). Once the Stage Manager acknowledges all Stage Processes of a given Stage, he starts communication with the Pipeline Manager (PM). The Stage Manager then sends the `Report` (status of all Stage Processes). Finally, the Pipeline Manager acknowledges the `Report` and sends `Action 0` to the Stage Manager. In short, the normal execution protocol is characterized by processes communicating, after `T(n)`, their status and receiving `Action 0` as a response. This happens periodically with a time interval `n` in `T(n)` set by the programmer. For example, in our experiments, we configured the processes to halt the Pipeline execution and enter into this reconfiguration protocol every 100 milliseconds.

Listing 5.1 showcases the normal execution algorithm in MPR. Throughout the Pipeline execution, data messages are received using MPI's probe function. We employ this function for two reasons: to leverage non-blocking operations and to bind the probed message to a specific message handler. Each time a message is received (condition in line 13), MPR leaves loop from line 1 to 13 and processes it accordingly. If messages are not received, eventually, after a period of time `checkInterval` (line 2), the Pipeline execution halts and the Stage Processes synchronize with their respective Stage Manager. Usually, this execution is fast since it represents a two-way handshake between a pair of processes, which adds negligible overhead to the application if the `checkinterval` is not

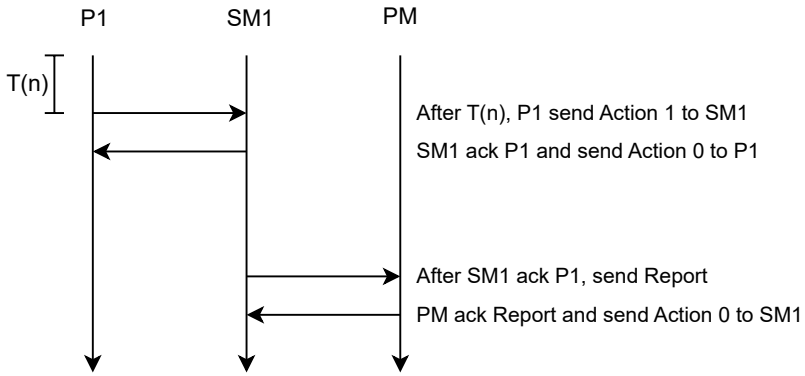


Figure 5.4: MPR configuration protocol during normal execution.

too frequent. Note that we cannot guarantee that synchronization will happen exactly at the moment of `checkInterval` because the Stage Process can be executing application code at that moment. However, once it has finished, before receiving another message, we guarantee that it will synchronize with the Stage Manager. As shown in Figure 5.4, Stage Processes send `Action 1` and receive `Action 0` when no further synchronization is required. The `Action 1` message will update information about the stage status containing the total number of items consumed and produced (line 3). Then line 4 sends the message while line 5 receives the response from the Stage Manager containing an Action to synchronize. During normal execution, the response is always Action 0, which means no synchronization is required and the process can resume processing the Pipeline. During reconfiguration, the `Action` code received can be `Action 3` (line 6 - `Action` to add processes) or `Action 4` (line 8 - `Action` to remove processes). Currently, MPR implements these two `Actions` (`Action` 3 and 4) to support adaptability. Also, it receives only one `Action` from Stage Manager at a time. In the future, this can be easily extended to support other features in MPR.

```
1  do {
2   if (waitingTime >= checkInterval) then
3    Update the stageStatus
4    Send Action 1 & stageStatus to the Stage Manager
5    Receive an Action from the Stage Manager
6    if (action == 3) then
7      Add processes
8    else if (action == 4) then
9      Remove processes
10   Reset the local clock
11  }
12  isDataMsgReady ← Probe new messages from the input or output communicators
13 } while (isDataMsgReady fails);
```

Listing 5.1: Example of MPR normal execution in Stage Process.

5.4.4 Adding Processes Protocol

Figure 5.5 illustrates the *configuration protocol* in case of adding new processes. As can be seen, periodically, the Stage Processes will send their status. Since no runtime system adaptation is required, the Stage Manager sends `Action 0` to P1. Periodically, the Pipeline Manager reads the `parameters.json` configuration file, this happens after a period of time `T(x)`, where `x` is configurable. It detects that new processes are needed via `Action 3`. So, the next time a Stage Manager reports the Stage Processes' status to the Pipeline Manager, it returns `Action 3`. The Pipeline Manager waits until notifying all Stage Managers of the new `Action` before entering into a configuration mode for `Action 3`. The same happens to the Stage Manager, which returns `Action 3` to all Stage Processes that send their status. Only after ensuring all Stage Processes received `Action 3` it enters into a configuration mode for `Action 3`. This protocol is important because MPI's spawn function is blocking and requires that all processes call the spawn function with exactly the same parameters for it to take effect. That is represented by the Spawn blocking call in Figure 5.5. With `Action 3`, the Stage Managers also send the new amount of processes for each stage. Then, the Stage Processes use this information to compute the number of new processes that must be spawned. Once the new processes are spawned (only P2 in our example), the Pipeline Manager sends `Action 6` (new process spawn information) and `Action 5` (configuration of a new process) to all new processes (P2). Then, the Pipeline Manager broadcasts the ranks that are executing in each group so that each process can recreate its inter-communicators. The inter-communicators that will be recreated were introduced in Figure 5.2, which depicts MPR's communication layout. It is worth noting that edge Stage Processes recreate two inter-communicators (with Stage Manager and either Input or Output) while middle ones recreate three inter-communicators (with Stage Manager and both Input and Output). In the end, a barrier ensures that all processes have finished configuring their new communicators before returning to the Pipeline execution. Also, our strategy puts the most computationally intensive parts of the strategy on the side of Stage Processes to preclude Stage Managers from becoming a bottleneck.

In the following, we present the synchronization algorithm MPR implements for the Stage Process of Compute type (middle stage) during the *configuration protocol* for adding new processes. We present the implementation of the synchronization protocol depicted in Figure 5.5. The algorithm is depicted in Listing 5.2. The algorithm implementation can differ slightly depending on the Stage Process type that executes it. For example, in the Compute process (the one we show in our examples), MPR must probe the input communicator checking for pending messages before recreating the inter-communicators. Instead, the Source process can skip such verification since it is the first Pipeline stage and does not receive input messages.

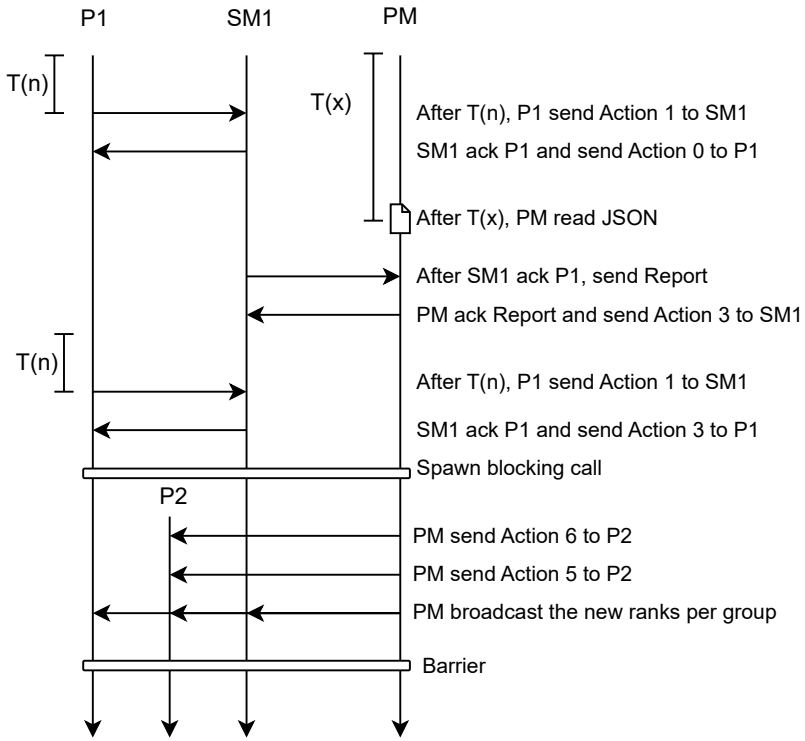


Figure 5.5: MPR configuration protocol when adding processes.

MPR is able to reconfigure the number of processes during execution time, and it ensures that no application data message is lost or duplicated. For example, MPR uses a `READY_MSG` to achieve a consistent global Pipeline state before processes enter into reconfiguration. The `READY_MSG` is generated by the first Stage Process and sent to all output ports. Stage Processes continue reading all input data messages until receiving the `READY_MSG`. Then, they send a `READY_MSG` to all their output ports notifying they are ready to reconfigure. Eventually, all processes will have received a `READY_MSG` from all its input ports and reconfiguration can be performed safely.

The first part of Listing 5.2 shows the mechanisms synchronizing messages between adjacent stages before *configuration protocol*. In line 2, we implement a logic that iterates an `importantEvents` buffer of received events to check for `READY_MSG` tags. MPR only starts the *configuration protocol* when all Stage Processes from the prior Stage are ready. Otherwise, MPR iterates the loop from line 3 to 15 to get all pending data messages and finishes when the `READY_MSG` tag is received (line 9). In this logic, line 7 probes incoming data messages from the input communicator, and line 13 buffers the messages for later processing. We do not process them at this point to avoid increasing the overhead of reconfiguration calls. After that, MPR ensures no pending messages in the intercommunicator. However, it is possible that there are unprocessed messages in our internal data buffer. This is not a concern when adding new processes, but it should be considered when removing processes.

```
1  if (action == 3) then
2    readyMsgFound ← iterate ImportantEvents buffer
3    while(true) then
4      if (readyMsgFound) then
5          break
6      end if
7      (isDataMsgReady, msgTag) ← Probe new messages from the input communicator
8      if (isDataMsgReady) then
9        if (msgTag == READY_MSG) then
10         Receive the configuration message
11         break
12       end if
13       Buffer the data message
14     end if
15   end while
16   for proc_id=0 to nextStageSize do
17     Send the ready signal to process proc_id
18   end for
19   for proc_rank=0 to amountNewProcesses do
20     Add 1 new process
21   end for
22   Receive and update stage list of ranks
23   if (stageSize < newStageSize) then
24     Update stage group of ranks
25     Recreate the Stage Manager communicator
26   end if
27   Recreate the input communicator
28   Recreate the output communicator
29   Wait in Barrier
30   Update new stage sizes
31 end if
```

Listing 5.2: Example of MPR's adding process protocol.

Between lines 16 and 18, MPR complements the aforementioned logic and sends the READY_MSG tag to notify the next Stage Processes that it is ready to start *configuration protocol*. After sending READY_MSG to all processes from the next Pipeline stage, MPR implements a logic to calculate the number of new processes that will be spawned (amountNewProcesses). Then, MPR adds new processes employing MPI's Spawn routine in line 20. Note that MPR spawns one process at a time, and the newly spawned processes help in the spawning of the next ones. We chose this design because MPI's spawn function creates a shared intra-communicator for each new spawn invocation. If processes are spawned together, they share the same intra-communicator and cannot be removed due to this dependency. Therefore, we spawn each process independently to avoid such limitations.

Finally, the last part of Listing 5.2 finishes the logic for adding new processes. All processes from the Pipeline receive an updated list of ranks that belong to each Pipeline

stage (line 22). For stages that have added new processes, MPR needs to update their inter-communicators. In a Compute Stage Process, three inter-communicators must be recreated (lines 25, 27, and 28). In the end, all processes wait in a barrier to ensure all inter-communicators are already recreated before using them (line 29). Line 30 finishes updating the new stage sizes and the Pipeline restarts the normal execution.

5.4.5 Removing Processes Protocol

The beginning of MPR *configuration protocol* when removing processes is similar to the one previously described, as can be seen in Figure 5.6. In a nutshell, all processes are notified with `Action 0` when in normal execution. After a period of time `T(x)`, the Pipeline Manager reads the `parameters.json` file, finds out that it was modified, and that processes need to be removed. The synchronization between processes happens after all of them received the configuration `Action 4`. Again, the Pipeline Manager and Stage Manager only enter into the configuration mode after they have sent the new `Action` to all processes that are under their responsibility. Then, the Pipeline Manager decides the processes that will be removed and broadcasts the removed ranks. Stage Processes use this list to compare against for checking if they can continue or are banned. Also, the Pipeline Manager broadcasts the new list of ranks per group to all processes. With that, the processes can update their local list of ranks per group. Finally, the ones that continue need to recreate their inter-communicator using the updated group information and wait in the barrier.

This protocol is challenging because MPI does not provide a function to remove processes. Although we were able to minimize the number of messages in the protocol, internally the routine to remove processes requires significant coordination between all processes. Even the removed processes are used to destroy current communicators. Moreover, once the new communicators are recreated, the processes receive new ranks, and the protocol needs to ensure they match the previous ranks of the Pipeline. For example, a process from Stage 1 cannot become a process from Stage 3. Usually, the processes have an internal state that cannot be lost. Therefore, the processes cannot simply swap Pipeline Jobs.

In both situations, special routines are activated before recreating the new communicators, either because new processes entered the Pipeline execution or were removed from the execution. That happens because messages could be already sent but were not received when the Pipeline reconfiguration started. Therefore, before destroying the communicator, the strategy checks if there are any stream items left in the buffer and computes them.

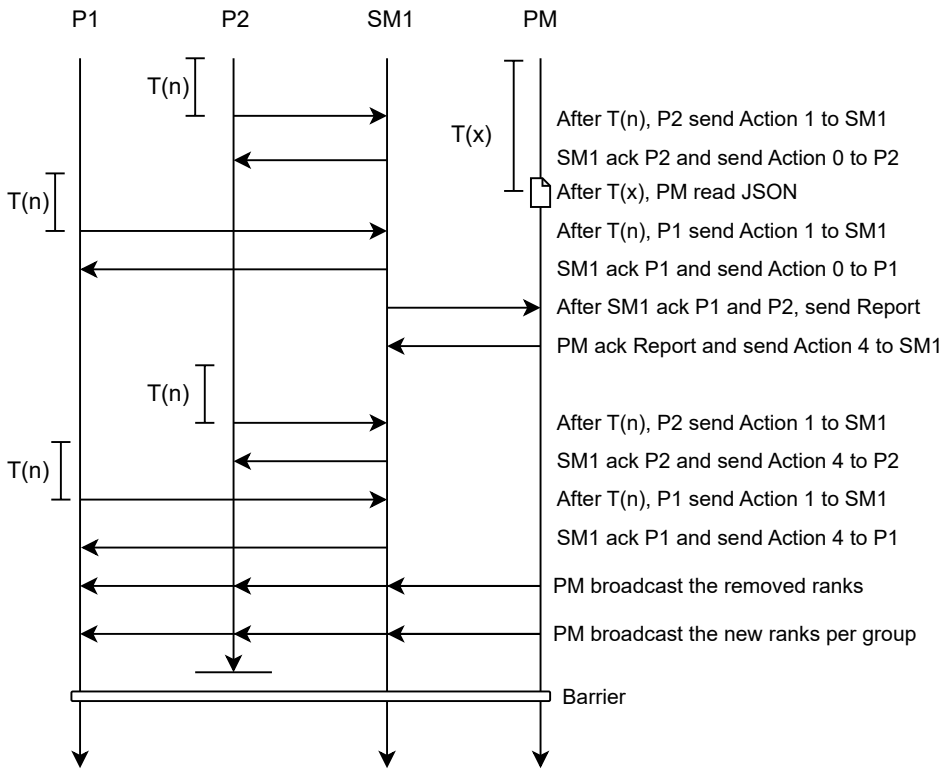


Figure 5.6: MPR configuration protocol when removing processes.

Listing 5.3 depicts the synchronization algorithm MPR implements during the *configuration protocol* for removing processes. We showcase the same Compute process, complementing the adding protocol from the previous Section 5.4.4. The first part of Listing 5.3 is similar to the algorithm to add processes. MPR employs a synchronization strategy to ensure that no application data message is lost or duplicated. In the beginning, each Stage Process iterates the `importantEvents` buffer to check if `READY_MSG` was already received (line 2). If not found, it enters in a loop (lines 3 to 15) to receive and buffer all incoming data messages (line 13) until receiving `READY_MSG` (line 9). After that, the Stage Process signalizes that it is ready to reconfigure by sending `READY_MSG` to all processes from the next Pipeline Stage (lines 16 to 18). Then, each process receives a list of ranks that are to be removed from their current communicators (line 19) and an updated list of ranks that continue residing in each Pipeline stage (line 20).

```
1  if (action == 4) then
2    readyMsgFound ← iterate ImportantEvents buffer
3    while(true) then
4      if (readyMsgFound) then
5          break
6      end if
7      (isDataMsgReady, msgTag) ← Probe new messages from the input communicator
8      if (isDataMsgReady) then
9        if (msgTag == READY_MSG) then
10         Receive the configuration message
11         break
12       end if
13       Buffer the data message
14     end if
15   end while
16   for proc_id=0 to nextStageSize do
17     Send the ready signal to process proc_id
18   end for
19   Receive a list of ranks to remove
20   Receive and update stage list of ranks
21   while(true) then
22     (isDataMsgReady, msgTag) ← Probe new messages from the output communicator
23     if (isDataMsgReady) then
24       if (dataBuffer is empty) then
25         if (msgTag == REQUEST_MSG) then
26           Receive the configuration message
27           Send the end_signal to the process
28           break
29         end if
30       else then
31         Process a data message from the buffer and send it
32       end if
33     end if
34   end while
35   for ban_rank in listOfRanksToBan do
36     if (myRank == ban_rank) then
37       Finalize the process
38     end if
39   end for
40   Update stage group of ranks
41   Recreate the Stage Manager communicator
42   Recreate the input communicator
43   Recreate the output communicator
44   Wait in Barrier
45   Update new stage sizes
46 end if
```

Listing 5.3: Example of MPR's removing process protocol.

The subsequent part of Listing 5.3 showcases MPR's implementation to clear the data buffer before removing the processes. In this step, all Stage Processes that will be removed must finish executing the data items from their internal buffer before exiting. However, since all Stage Processes must wait in this step MPR allows all of them to continue executing to minimize overhead. For example, if a single process is removed, all other processes become idle until it finishes, so MPR allows them to process as well during this step (line 31). Since our *data protocol* implements on-demand scheduling, the Stage Processes expect to receive a `REQUEST_MSG` in order to process the data item and send it to the requester. When the buffer is empty (line 24), the Stage Process receives the last `REQUEST_MSG` (line 25) and replies to the requester with a `END_MSG` (line 27) indicating that all buffered data was sent.

The next part of the algorithm presents the mechanisms MPR implements to finalize the Stage Processes that will be removed. For the removing protocol, each Stage Process checks if it is one of the removed processes (line 36). If true, the removed processes disconnect from the Pipeline and invoke MPI's finalize function (line 37). For the processes that were not removed, they start updating the Pipeline mechanisms. First, the Stage Processes update their information about which ranks are in each Pipeline stage (line 40). After, the new rank list is used to recreate the inter-communicators: with Stage Manager (line 41), with prior Pipeline stage (line 42), and with next Pipeline stage (line 43). Subsequently, a barrier ensures that all processes have recreated the inter-communicators before using them (line 44). Finally, line 45 updates the stage sizes, and the Stage Process can resume the Pipeline normal execution.

5.4.6 MPR Data Communication Protocols

In the previous sections, we presented the synchronization protocols employed by MPR for the *configuration protocol*, which is one of two communication protocols used by the framework. The second protocol, the *data protocol*, is used to exchange stream data items between the Pipeline Graph processes. In the following sections, we extend the discussion to the *data protocol* to complement our earlier discussion of the *configuration protocol*. To explain MPR's communication design between the Pipeline stage Processes, we employed the concept of MPI's intra- and inter-communicators. Please refer to Figure 5.2, which was used in the previous section to explain MPR's low-level communication layer implementation. As previously explained, each Pipeline Job is implemented with its own MPI intra-communicator. The following sections cover MPR's data communication as follows: Section 5.4.7 presents the initialization step that creates the processes and assigns them to a specific Pipeline Job. Section 5.4.8 briefly introduces MPR's data transfer

and scheduling. Finally, Section 5.4.9 discusses MPR's strategy to communicate stream data items between the different Pipeline Jobs.

In the upcoming Listings, we have included multiple code snippets that offer insight into the implementation of MPR using MPI. As previously mentioned, we have chosen to present these code segments without excessive abstraction to assist those who may be grappling with the complexities of MPI implementations. We hope that readers can draw inspiration from our code to create their own MPI implementations, especially given the paucity of examples showcasing less-common MPI interfaces. For those readers who are not MPI developers, we suggest that they skim over the code snippets and focus on the explanations provided in the text.

5.4.7 Process Creation and Job Assignment

In this section, we describe how MPR creates processes and assigns them to Pipeline Jobs. The code logic executed by the first processes created by MPI when the user calls `mpirun` is depicted in Listing 5.4. The number of processes to be spawned is obtained in lines 1 to 4. This number equals the sum of: 1 Pipeline Manager + 1 Stage Manager for each Pipeline stage + the number of Stage Processes specified by the user in `parameters.json`. The first extra MPI process is created in lines 5 to 7. The loop from line 8 to 17 creates the remaining processes, with each process helping to create the next ones. MPR implements a logic to inform each new spawned process of how many processes it must help to spawn, as described by `Action 6`, which is sent between lines 9 and 13. When all processes have been created, the first MPI process communicates to all others by sending `Action 0` (lines 18 to 23). We will explain the purpose of this last message when we show MPR's logic to set Pipeline Jobs later.

```
1  int n_procs = pipeInfo.GetNumStages();
2  for(int stageID = 1; stageID <= pipeInfo.GetNumStages(); stageID++) {
3   n_procs += pipeInfo.GetStageSize(stageID);
4  }
5  MPI_Comm interComm;
6  MPI_Comm_spawn(argv[0], argv+1, 1, MPI_INFO_NULL, 0, MPI_COMM_WORLD, &interComm,
       MPI_ERRCODES_IGNORE);
7  MPI_Intercomm_merge(interComm, 0, globalComm);
8  for(int processRank=1; processRank<=n_procs; processRank++){
9   bufConfigAux[0] = 6;
10  bufConfigAux[1] = n_procs-processRank;
11  MPI_Request requestReconfig;
12  MPI_Isend(bufConfigAux, pipeInfo.GetNumStages()+1, MPI_INT, processRank,
      RECONFIG_MSG, *globalComm, &requestReconfig);
13  MPI_Wait(&requestReconfig, MPI_STATUS_IGNORE);
14  if(processRank == n_procs) break;
```

```
15  MPI_Comm_spawn(argv[0], argv+1, 1, MPI_INFO_NULL, 0, *globalComm, &interComm,
        MPI_ERRCODES_IGNORE);
16  MPI_Intercomm_merge(interComm, 0, globalComm);
17 }
18 for(int processRank=1; processRank<=n_procs; processRank++){
19  bufConfig[0] = 0;
20  MPI_Request requestReconfig;
21  MPI_Isend(bufConfig, pipeInfo.GetNumStages()+1, MPI_INT, processRank,
        RECONFIG_MSG, *globalComm, &requestReconfig);
22  MPI_Wait(&requestReconfig, MPI_STATUS_IGNORE);
23 }
```

Listing 5.4: First process created by MPI.

Listing 5.5 complements the previous Listing 5.4. In MPI, processes that are spawned during execution time (child processes) hold an additional communicator that links them with their parent processes. To obtain the parent communicator, we use the `MPI_Comm_get_parent` function, as shown in line 1. This is how we differentiate the first processes from the others, since the MPI processes created statically do not have a parent, and their communicator returns `MPI_COMM_NULL`. Once we have obtained the parent communicator, we merge it with the local communicator of the new processes to create a global intra-communicator (line 2). All processes that belong to the Pipeline are part of this global communicator. The global communicator is important because it allows the Pipeline Manager to have a direct connection with each process of the Pipeline during execution time. Line 3 receives the number of processes that the new process should help spawn via `Action 6`. The loop from lines 6 to 8 then spawns the new processes and includes them in the global communicator. Finally, all processes receive a *configuration protocol* `Action` message (line 9). This message is used in MPR's logic to set processes' Pipeline Jobs.

```
1  MPI_Comm_get_parent(&parentComm);
2  MPI_Intercomm_merge(parentComm, 1, globalComm);
3  MPI_Irecv(bufConfigAux, pipeInfo.GetNumStages()+1, MPI_INT, 0, RECONFIG_MSG, *
       globalComm, &requestReconfig);
4  MPI_Wait(&requestReconfig, MPI_STATUS_IGNORE);
5  for(int processRank=1; processRank<=bufConfigAux[1]; processRank++){
6   MPI_Comm_spawn(argv[0], argv+1, 1, MPI_INFO_NULL, 0, *globalComm, &interComm,
        MPI_ERRCODES_IGNORE);
7   MPI_Intercomm_merge(interComm, 0, globalComm);
8  }
9  MPI_Irecv(bufConfig, pipeInfo.GetNumStages()+1, MPI_INT, 0, RECONFIG_MSG, *
       globalComm, &requestReconfig);
10 MPI_Wait(&requestReconfig, MPI_STATUS_IGNORE);
```

Listing 5.5: Processes spawned by MPI during execution time.

Listing 5.6 demonstrates MPR's logic for assigning processes to specific Pipeline Jobs. This logic follows the dynamic process spawning logic that we discussed in the pre-

vious listings. For the sake of brevity, we show only a part of the code, but we summarize the remaining strategy. The first two lines (1 and 2) abstract the implemented strategy for creating `MPI_Groups`, which are used to organize the processes' ranks.

To illustrate the process assignment to Pipeline Jobs, let us consider a Pipeline with three stages and a configuration file `parameters.json` that specifies `stage1:1, stage2:3, stage3:1`. In this case, the total number of spawned processes is 9. Rank (0) is the Pipeline Manager. Ranks 1 to `pipeline_size` are the Stage Managers, being ranks are (1), (2), and (3). The remaining process ranks are assigned to Stage Processes. Although our strategy supports the arbitrary assignment of process ranks to Stage Processes, in the current MPR version, we assign the first rank after the last Stage Manager to the Source process and the subsequent rank to the Sink process. Therefore, Stage Processes 1, 2, and 3 are assigned to ranks (4), (6, 7, 8), and (5), respectively. This design choice is based on our experimental tests, where we aimed to have a fair comparison with DSParLib and MPI's handwritten static versions. We will discuss this topic further in the experiments section, where we provide more details about MPI's process allocation strategies.

```
1  pipeInfo.SetStageManagerGroups();
2  pipeInfo.SetStageProcessGroups();
3  for(int managerID = 0; managerID <= pipeInfo.GetNumStages(); managerID++) {
4   MPI_Group * stageManagerGroup = pipeInfo.GetStageManagerGroup(managerID);
5   MPI_Group_rank(*stageManagerGroup, &localRank);
6   if(localRank != MPI_UNDEFINED) {
7    procInfo.SetProcJob(managerID);
8    int groupTag = 2000+managerID;
9    MPI_Comm * pipelineComm = pipeInfo.GetPipelineComm();
10   MPI_Comm_create_group(*globalComm, *stageManagerGroup, groupTag,pipelineComm);
11   break;
12  }
13 }
14 for(int stageID = 1; stageID <= pipeInfo.GetNumStages(); stageID++) {
15  MPI_Group * stageProcessGroup = pipeInfo.GetStageProcessGroup(stageID);
16  MPI_Group_rank(*stageProcessGroup, &localRank);
17  if(localRank != MPI_UNDEFINED) {
18   procInfo.SetProcJob(pipeInfo.GetNumStages() + 1 + stageID - 1);
19   procInfo.SetStageID(stageID);
20   int groupTag = 2000 + pipeInfo.GetNumStages() + 1 + stageID;
21   MPI_Comm * pipelineComm = pipeInfo.GetPipelineComm();
22   MPI_Comm_create_group(*globalComm, *stageProcessGroup, groupTag,pipelineComm);
23   break;
24  }
25 }
```

Listing 5.6: `PipelineGraph`'s SetJobs() function.

Once all available MPI process ranks are assigned to `MPI_Groups`, MPR uses them to create the local intra-communicators. Each Pipeline Job has its own local communicator to communicate with the other Pipeline Jobs. Lines 3 to 13 create the Stage Managers' intra-communicators, while lines 14 to 25 create the Stage Processes' intra-communicators. Line 4 gets the `MPI_Group` reference created in Line 1, and line 5 retrieves the local rank of the current calling process in that group. Only processes belonging to the group receive valid ranks, and others receive `MPI_UNDEFINED`. If a process belongs to a given group, it sets its Pipeline Job in line 7 and creates the intra-communicator in line 10. To organize the communication, we use `pipelineComm` as the intra-communicator, and each Pipeline process overwrites `pipelineComm`'s reference accordingly.

Similarly, in line 15, MPR gets the Stage Processes `MPI_Group` reference that was created in Line 2. Then, in line 16 tries to get the local rank of the current calling process in that group. If a process belongs to a given group, it sets its Pipeline Job in line 18 and assigns the stage identifier in line 19. Finally, the process creates the `pipelineComm`. After finishing Listing 5.6 each process knows its Pipeline Job and shares an intra-communicator with all processes with the same Job. Later, this intra-communicator is used to create the required Pipeline inter-communicators with other Pipeline Jobs. In MPR's current version, we implement the Pipeline pattern, which enables processes to communicate only with adjacent Pipeline stages. However, future work can investigate MPR's flexibility toward the dataflow paradigm, where inter-communicators are employed for communicating non-consecutive DAG stages.

5.4.8 Data Transfer and Scheduling

In this section, we elaborate on MPR's data scheduling and transfer design between different Stage Processes, which optimizes load balancing and enables back pressure during Pipeline execution. MPR uses an all-to-all communication model, which means that each Stage Process from a given Pipeline stage communicates with all the Stage Processes of its neighbors. The communication is performed using an on-demand approach. Therefore, each Stage Process requests data from the previous Pipeline stage using the `REQUEST_MSG` tag. When the previous stage receives a `REQUEST_MSG`, it sends the data to the requester. The identification of the Stage Process to request or send data is straightforward in MPR. Due to our design choice, MPR uses the Pipeline stage intra-communicators and enables using their local rank to address a target Stage Process. MPI is able to translate a local process rank to its global rank. For example, if a given Pipeline stage has three Stage Processes with global ranks {4,8,17}, then their local ranks are {0,1,2}, respectively. MPR leverages this last representation since the ranks can be easily accessed using a loop construction from 0 to *pipeline_stage_size*.

In applications with unbalanced workloads, the on-demand data transfer strategy used by MPR can effectively improve performance, despite requiring more messages to communicate. This is because a Stage Process only requests new data when it has completed processing its previous one. However, in applications with balanced workloads, the on-demand mechanism can potentially slow down the Pipeline execution, as a Stage Process becomes idle from the moment it requests new data until it receives it. To minimize this idle time, MPR includes an internal data buffer with a configurable size that can be adjusted during execution. As a result, each Stage Process can buffer a variable amount of input data until the internal data buffer structure is full, thus reducing idle time and improving overall performance.

In addition to load-balancing, the on-demand mechanism also handles back pressure. Back pressure is a key feature of DSPS that enables it to cope with the varying speeds of senders and receivers [49]. To prevent the system from crashing in case of overloading, MPR Stage Processes only compute and send data when they receive a `REQUEST_MSG`. However, if a Stage Process does not receive any `REQUEST_MSG`, it can become idle, resulting in performance degradation. To optimize performance, whenever a Stage Process receives a `REQUEST_MSG`, it first checks its internal data buffer for available data to process. If data is found, it is computed and sent to the requesting Stage Process. If the buffer is empty, the requesting Stage Process rank is stored for future use. The next time data is received, it is immediately processed and sent to the stored ranks without buffering. This way, both request messages and data messages can be internally buffered, and each time a request message matches a data message, it is computed and sent to the requester, improving overall system performance.

5.4.9 Publishing and Receiving Data

In this section, we describe MPR's approach for exchanging data messages in the application. Each data stream item is sent in a multi-message fashion. MPR uses a `Header` message to establish communication, followed by one or more payload messages. The `Header` message consists of two fields: the message ID and the number of payload messages that will follow. Listing 5.7 shows how MPR implements the `Publish()` interface. In line 2, a pointer to the `Ctx` object is cast to access context variables needed for executing the Pipeline Graph. For the `Publish` function, MPR obtains the message header from the application context (line 4) and updates the number of messages that will follow the current header (line 5). The header message is then sent in line 7, and the application payload is sent in line 9 using `MPI_BYTE` to simplify data management. Finally, MPR increments the number of produced items (line 11).

```
1 void Publish(void * _ctx, void * dataOut, int size){
```

```
2  Ctx * ctx = static_cast<Ctx*>(_ctx);
3  MPI_Comm * outputComm = stageInfo->GetOutputComm();
4  Header * header = ctx->GetHeader();
5  header->incomingMsgs = 1;
6  int targetRank = ctx->GetTargetProc();
7  MPI_Isend(header, sizeof(Header), MPI_BYTE, targetRank, HEADER_MSG, *outputComm
       , &requestSend);
8  MPI_Wait(&requestSend, MPI_STATUS_IGNORE);
9  MPI_Isend(dataOut, size, MPI_BYTE, targetRank, DATA_MSG, *outputComm, &
       requestDataSend);
10 MPI_Wait(&requestDataSend, MPI_STATUS_IGNORE);
11 stageInfo->IncrementItemsProduced();
12 };
```

Listing 5.7: MPR's interface for publishing data.

In addition to the `Publish` interface, MPR implements another interface called `PublishMulti`. Note that `Publish` can send only static and contiguous data types. It cannot deal with data located at different memory locations since it only accepts a single reference to data. For that, we provide the `PublishMulti` interface presented in Listing 5.8. The abstracted code is identical to the `Publish` function, but instead, it iterates a list of multiple pointers and sizes in loop from lines 4 to 8. This interface is important because it allows MPR's runtime system to support dynamically allocated data. More details are given in the following when we present the receiving side of these interfaces.

```
1  void PublishMulti(void * _ctx, vector<void *>& dataOutList, vector<int>& size){
2   /* Abstracted code logic identical to Publish() */
3   header->incomingMsgs = dataOutList.size();
4   for(long unsigned int dataID=1; dataID<dataOutList.size(); dataID++){
5    int targetRank = ctx->GetTargetProc();
6    MPI_Isend(dataOutList[dataID], size[dataID], MPI_BYTE, targetRank, DATA_MSG, *
       outputComm, &requestDataSend2);
7    MPI_Wait(&requestDataSend2, MPI_STATUS_IGNORE);
8   }
9   stageInfo->IncrementItemsProduced();
10 };
```

Listing 5.8: MPR's interface for publishing multiple data.

Upon receiving a message, the first step for Stage Processes is to determine the action required. It does so by checking the tag associated with the message. Listing 5.9 shows the message tags available in MPR. Briefly, `STOP_MSG` acknowledges the stop signal; `REQUEST_MSG` either buffers the requester's rank or computes a data from its internal buffer, as discussed before; `END_MSG` and `READY_MSG` are configuration messages unintentionally captured during application execution, so MPR buffers them in the `importantEvents` buffer; finally `HEADER_MSG` is the function we are interested in this section.

```
1 bool ReceiveAndBufferMsg(Ctx * ctx){
2  if (status->MPI_TAG == STOP_MSG) { ... }
3  else if (status->MPI_TAG == HEADER_MSG){ ... }
4  else if (status->MPI_TAG == REQUEST_MSG) { ... }
5  else if(status->MPI_TAG == END_MSG){ ... }
6  else if(status->MPI_TAG == READY_MSG){ ... }
7  return false;
8 }
```

Listing 5.9: MPR's interface when receiving MPI messages.

When messages containing the HEADER_MSG are received, MPR executes the unmarshalling logic described in Listing 5.10. It starts by receiving the Header, which is the first message each data stream item sends (line 3). Subsequently, MPR probes the next payload message (line 7) and receives it in line 11. By default, the protocol ends after decrementing the number of incoming messages (line 13) because it fails the verification in the loop from line 15. However, PublishMulti enables sending multiple data from different memory pointers. For each extra payload, MPR probes the message (line 17), then gets its dynamic size (line 19) and allocates a new memory space (line 20). Finally, it receives the data in line 21 and overwrites the pointer in line 23. This last part is a low-level C++ memory manipulation that enables manually replacing the pointer of the user's struct by computing its location. Note that MPR configures a static pointer size of 8 bytes (64 bits). In the future, we plan to extend the experiments to different architectures and compilers to assess if this strategy is generic or needs to be adapted. For example, an alternative could be replacing the static 8 value with `sizeof(void*)` or using a compiler variable to get this information.

```
1 MPI_Get_count(status, MPI_BYTE, &size);
2 Header * header = new Header();
3 MPI_Imrecv(header, size, MPI_BYTE, probeMsg, &requestRecvData);
4 MPI_Wait(&requestRecvData, MPI_STATUS_IGNORE);
5 int incomingMsgs = header->incomingMsgs;
6 do{
7     MPI_Improbe(status->MPI_SOURCE, DATA_MSG, inputComm, &isDataMsgReady, &
    probeDataMsg, &statusRecv);
8 }while(!isDataMsgReady);
9 MPI_Get_count(&statusRecv, MPI_BYTE, &staticSize);
10 char * buffer = new char[staticSize];
11 MPI_Imrecv(buffer, staticSize, MPI_BYTE, &probeDataMsg, &requestRecvData);
12 MPI_Wait(&requestRecvData, MPI_STATUS_IGNORE);
13 incomingMsgs--;
14 TDataIn * payload = static_cast<TDataIn*>((void*)buffer);
15 while(incomingMsgs > 0){
16     do{
17         MPI_Improbe(status->MPI_SOURCE, DATA_MSG, inputComm, &isDataMsgReady, &
    probeDataMsg, &statusRecv);
```

```
18      }while(!isDataMsgReady);
19      MPI_Get_count(&statusRecv, MPI_BYTE, &size);
20      char * buffer = new char[size];
21      MPI_Imrecv(buffer, size, MPI_BYTE, &probeDataMsg, &requestRecvData);
22      MPI_Wait(&requestRecvData, MPI_STATUS_IGNORE);
23      *((char**)payload+(sizeof(TDataIn)/8)-incomingMsgs) = buffer;
24      incomingMsgs--;
25  }
26  /* Abstracted code logic for immediate computation */
```

Listing 5.10: MPR's interface when receiving application data messages.

The data communication strategy we just described is how MPR enables zero-copy serialization. Note that the strategy does not involve the CPU, and data is transferred from its original memory location without intermediate copies to obtain contiguous data. This way, MPR implements communication following the data serialization design goal described in Section 5.1. In addition, the `PublishMulti` is an extra abstraction we provide for efficient data transfer. To implement communication in streaming applications, programmers can combine their preferred serialization library with the `Publish` interface. Available options include Cereal [34], Boost [19], and Protocol Buffers [33].

5.5 Adaptability Support

In this section, we describe how self-adaptive algorithms can work together with MPR to monitor application execution and adjust the number of parallel processes as needed. While the low-level mechanisms that enable MPR to support adaptability were already discussed during the processing engine section, here we focus on the interfaces that programmers can use. These interfaces are simple Json files that can be read from or written to. This design principle makes it possible for self-adaptive algorithms to be written in any programming language that supports Json files, thus enhancing MPR's portability. Furthermore, this modular interface separates the concerns of programmers and researchers from MPR's low-level runtime system mechanisms, allowing them to focus on the adaptability model. This section is brief, as the monitoring interface is presented in Section 5.5.1, and the adaptation decision is explained in Section 5.5.2.

5.5.1 Monitoring

MPR provides Pipeline execution statistics in the form of Json files, which can be used by self-adaptive programs to monitor application execution and adjust the number of processes by scaling up or down. Each Pipeline stage has its own statistics file,

which allows parallel stages to write metrics simultaneously for a given timestamp. This is important since the Stage Managers continuously accumulate metrics from the running processes during execution time, and race conditions may happen if written in a single file.

Currently, MPR monitors the number of items consumed and produced by each Pipeline stage, and also reports a composed metric, which is the average throughput of each Pipeline stage obtained when dividing the number of items produced by the time. This initial set of metrics serves as a basis for self-adaptive algorithms to make informed decisions about the system's performance and to trigger adaptations.

MPR collects the number of items consumed and produced from all the active Stage Processes. The Stage Processes periodically communicate with the Stage Managers via the *configuration protocol* and report the updated metrics. While these metrics are the ones used in MPR's self-adaptive algorithm, there are many other metrics that can be derived from the application execution, such as CPU utilization, latency, memory consumption, and total bytes sent or received. To include additional metrics, programmers can either replace or include them in MPR's current synchronization algorithm. This flexibility allows programmers to tailor their monitoring needs while abstracting them from MPR's runtime system.

An example of monitoring reported in a Json file is depicted in Listing 5.11. The Stage Manager (`stage2` in this example) reports the number of items consumed and produced for each Stage Process (`Compute0` and `Compute1`). Note that for each timestamp, MPR regulates the number of current running Stage Processes and reports accordingly. In this example, the first timestamp (line 2) converted to a human-readable date is February 4, 2023 11:25:04. This timestamp ensures that each sample is unique in the statistics report. Moreover, this report format also enables algorithms to detect load-balancing issues in the Pipeline execution. For instance, the first timestamp (line 2) reports a single Stage Process while the second (line 9) reports two Stage Processes. This means that in between the first and second timestamps, MPR spawned a new Stage Processes for `stage2`. Note that in the second timestamp, `Compute0` processes almost 25 thousand more items than `Compute1`.

```
1  "stage2": {
2      "1675563904635": {
3          "Compute0": {
4              "ItemsConsumed": 81347,
5              "ItemsProduced": 81348,
6              "averageItemsProduced": 40674.53009081341
7          }
8      },
9      "1675563909814": {
10         "Compute0": {
11             "ItemsConsumed": 74916,
```

```
            "ItemsProduced": 74916,
            "averageItemsProduced": 37459.205175008094
        },
        "Compute1": {
            "ItemsConsumed": 50588,
            "ItemsProduced": 50587,
            "averageItemsProduced": 25294.31379395769
        }
    },
}
```

Listing 5.11: Example of Json format MPR uses to report application statistics.

5.5.2 Adaptation Decision

MPR decides on the number of processes in each Pipeline stage by periodically checking the `parameters.json` configuration file. The Pipeline Manager is responsible for performing the read. When the Pipeline Graph execution starts, it reads the file to decide how many processes will be spawned. Then, during the Pipeline execution, the Pipeline Manager regularly reads the file. For each read, it compares the number of current processes with the new value read from the `parameters.json` file. An example of such file is showcased in Listing 5.12. If the read number of processes of a given stage is higher than the current amount, then an `Action 3` takes place. Every time a Stage Manager communicates with the Pipeline Manager it will receive `Action 3` as a response. The same happens if the read number of processes is smaller than the current amount, in which `Action 4` takes place. To summarize, each time the values differ from a previous read, the Pipeline Manager analyzes if processes must be added or removed in order to adapt the runtime system to the specified amount of processes. In this initial version, MPR supports changes in one stage at a time.

```
{
    "stage1": 1,
    "stage2": 64,
    "stage3": 1
}
```

Listing 5.12: Example of `parameters.json` to set the number of processes in MPR.

MPR's self-adaptive interface offers flexibility and portability, allowing programs to read the `parameters.json` configuration file and set an arbitrary number of processes based on the adaptation decision. This means that there are no restrictions on the programming language used to write the self-adaptive algorithm. MPR's API is designed to be easy to use by programmers and researchers alike. While C++ may achieve higher performance than other high-level programming languages, it is not as popular for data science

and machine learning algorithms. Python, on the other hand, is well-suited for designing efficient self-adaptive algorithms due to its wide range of libraries. Furthermore, the program responsible for deciding the number of processes does not need to achieve maximum performance. In fact, the self-adaptive algorithm used to test MPR's API is written in Python, and we describe it in Section 7 during the experimental evaluation.

In future work, we aim to expand the range of configurable parameters beyond just the number of processes. While MPR's current implementation allows for self-adaptive adjustments on the number of processes during execution time, there are other metrics within the runtime system that could be dynamically configured as well. For example, the buffer size and various time intervals are hard-wired into the current implementation but could potentially be adjusted during runtime to improve system performance. In this study, we explore different configurations for some of these parameters and report our findings in the experimental section. By expanding the range of configurable parameters, we hope to provide users with more flexibility to fine-tune the runtime system to their specific use cases and requirements.

5.6 High-level API for Distributed Stream Processing

In this section, we introduce MPR's high-level API designed for implementing distributed stream processing applications. This interface focuses on application programmers that want to use MPR for implementing streaming applications. The primary building block of MPR's Pipeline Graph is the Stage Process, which is responsible for executing the Pipeline streaming application. The Managers are the processes that coordinate the Pipeline execution. Programmers are required to "wrap" the streaming application code in terms of Stage Processes, and append these already-implemented Stage Processes to produce the final Pipeline Graph topology. The main benefit of this approach is that it abstracts many of the parallelism complexities from the programmer. With MPR, they don't have to implement everything from scratch each time a new streaming application is developed, which would be the default behavior for implementing low-level MPI distributed applications. Typically, many of the parallelism and communication complexities, such as schedulers, message-passing protocols, and process synchronization, need to be manually implemented.

MPR's Pipeline Stages can be implemented by extending MPR's Stage Process. By inheriting the functions provided within the Stage Process, programmers are equipped with all the interfaces they need to implement communication between the Pipeline stages. However, before using these interfaces, programmers must implement the Stage Process virtual functions with the application code that will be executed. For this purpose, MPR exposes four virtual interfaces:

- **`OnInit`**: This interface is executed once at the beginning of the Stage Process' lifetime, and programmers can write the application code for this stage here.
- **`OnEnd`**: This interface is executed once at the end of the Stage Process' lifetime, and programmers can write the application code for cleaning up resources and finalizing the stage here.
- **`OnInput`**: This interface is executed each time a new input data message is received, and programmers can write the application code that processes this data here.
- **`OnProduce`**: This interface is executed for generating new data stream items, and programmers can write the application code that produces new data or reads from another source here.

Programmers can write the code logic for each of these functions according to the requirements of their application. In case nothing needs to be executed in any of these steps, the corresponding functions can be left empty.

Listing 5.13 provides an example of MPR's API for implementing a streaming application. In this example, the stream processing application processes the numbers from `0` to `totalNum` and calculates the total number of primes in this interval.

```
class Stage1: public StageProcess <void*, int> {
public:
  void OnProduce(void * ctx) {
   for(int i=0; i<totalNum; i++){
    Produce(ctx, &i, sizeof(i));
} }; };

class Stage2: public StageProcess <int, bool> {
public:
  void OnInput(void * ctx, void * dataIn) {
   int i = Unpack(dataIn);
    bool isPrime = true;
    for (int j = 2; j < i; j++) {
     if (i % j == 0) {
      isPrime = false;
      break;
     }
    }
    void * dataOut = Pack(isPrime);
    Publish(ctx, dataOut, sizeof(isPrime));
}; };

class Stage3: public StageProcess <bool, void*> {
public:
  int primes;
```

```
26    void OnInput(void * ctx, void * dataIn) {
27      bool * isPrime = static_cast<bool*>(data);
28      if(*isPrime) {
29        primes++;
30  } }; };
```

Listing 5.13: Example of MPR's API for implementing streaming applications.

The Stage1 is responsible for producing the stream data items (lines 1 to 6). Each new number generated in the loop (line 4) is sent to the next stage via the Produce interface, which receives as a parameter the execution context (ctx), the data reference, and its size.

Next, the Stage2 represents the computational stage that receives the data items, processes them, and emits a boolean indicating if the number is a prime number or not (lines 8 to 21). The OnInput function callback is activated each time a stream item is received by the Stage Process. The data is received in a reference, which must be cast to the original data type. MPR provides an interface to help the user with this called Unpack (line 11). Also, MPR provides another helper interface called Pack (line 19), which returns a pointer to the object. The results from Stage2 can be published to the next stage using Publish. This interface also receives as a parameter the execution context (ctx), the data reference, and its size.

Finally, the Stage3 accumulates the results (lines 23 to 30). In line 27, an alternative way for manually casting the input stream data item is shown. This is similar to what is done in Unpack. More experienced programmers may prefer not to use Pack and Unpack as they may observe a minor performance improvement.

MPR's interface abstracts many low-level parallelism and distributed complexities from the programmer. To send data, programmers only need to provide the correct data reference and size, which can easily be obtained using the sizeof function in C++. When a new stream data item arrives, the OnInput function is called, providing the execution context and data payload. The payload is then cast to the desired data type. Currently, the execution context is not used in MPR's API, but we plan to provide lower-level interfaces for experienced programmers and researchers to extend MPR's flexibility. For instance, we can allow programmers to manually set the target process to which the application should send the message via the ctx parameter. This feature can be quickly implemented with minor modifications to MPR's code since it already uses the ctx parameter internally to set the target Stage Process. However, we do not expose this functionality to the user.

```
1 mpr::PipelineGraph pipe(&argc, &argv, 3);
2
3 Stage1 source;
4 Stage2 compute;
5 Stage3 sink;
6
```

```
7  pipe.AddPipelineStage(&source);
8  pipe.AddPipelineStage(&compute);
9  pipe.AddPipelineStage(&sink);
10
11 pipe.Run();
```

Listing 5.14: Example for assembling Stage Processes in a Pipeline Graph using MPR.

Once the Stage Processes are properly implemented to execute the application code, they can be assembled in a Pipeline fashion to be executed by MPR's runtime system. Listing 5.14 showcases how this can be done in MPR. Initially, a `PipelineGraph` object is created (line 1). The parameters expected are the application's `argc` and `argv`, and the pipeline size. After creating the `PipelineGraph`, programmers instantiate the Pipeline stage operators (lines 3 to 5). In this example, we use the Stage Processes implemented in the previous Listing 5.13. Each Stage Process operator becomes a Pipeline stage when it is appended to the `PipelineGraph` via the `AddPipelineStage` function (lines 7 to 9). Finally, MPR starts executing the Pipeline when the program calls the `Run()` function (line 11).

While MPR currently exclusively supports the Pipeline pattern, our plans for future versions involve exploring support for different patterns. With the MPR runtime system now implemented, we believe its flexibility allows for such exploration. For example, we could implement data parallelism via the Map pattern by modifying the available Pipeline protocol to send a single message containing all the data that needs to be processed. However, this would require extending the protocol to support resending data that is internal to one Stage Process to other Stage Processes, in order to maintain MPR's adaptability.

MPR also provides additional abstractions programmers can enable in their stream applications. The abstraction we provide in this version regards the ordering of stream data items. This feature can be enabled by calling `EnableOrdering` in the Stage Process ordering is required. Since parallelism can lead to non-deterministic program execution, some applications require re-ordering of stream data to ensure output integrity, such as in video processing where frame ordering is crucial. MPR supports the ordering of data after parallel Pipeline stages, using an algorithm based on the mechanisms proposed by Griebler et al. [40]. Specifically, MPR employs a `std::priority_queue` to buffer stream data items that are out of order. The decision on whether a data item is out of order is based on the unique message ID that each item carries in its header.

6. MPR PERFORMANCE EVALUATION

This chapter presents our evaluation of MPR's performance and compares its overhead against static distributed stream processing implementations using DSParLib and handwritten MPI versions. In the next chapter (Chapter 7), we conduct additional experiments to evaluate and characterize MPR's API for supporting an autonomic management module. MPR leverages dynamic process management, which we expect to result in additional overhead since the runtime system needs to make extra function calls to check the current Pipeline stage size each time it sends a new message. To better understand this overhead, we identify the main features that we believe contribute the most to it and present them in the following.

- **MPR spawns extra processes:** In additional to Stage Processes, those responsible to execute the Pipeline application code, MPR also spawns Manager Processes for coordinating the Pipeline Graph execution. These Manager Processes are always active and they periodically exchange messages with the Stage Processes. Therefore, MPR cannot disable them to have a fair comparison with DSParLib and MPI. Nonetheless, we can adjust the synchronization period to significantly reduce their synchronization and the number of messages exchanged by them. Although, this does not isolate them from the application since extra communication messages are still active in the runtime system, this can help in a fairer comparison.
- **MPR dynamic process management:** The static DSParLib and MPI versions use a single global intra-communicator to communicate between the Pipeline processes. MPI relies on the `MPI_COMM_WORLD` creates with size equal to the number of processes specified in the `mpirun` command. Differently, DSParLib creates the processes during execution time. However, it creates them only once at the beginning via a static number of processes and merges them in a single global intra-communicator, where process management becomes similar to MPI. Instead, MPR uses multiple groups of processes and inter-communicators to dynamically manage the processes. During execution time, Stage Processes execute additional functions to receive and send data, such as checking the current Pipeline stage sizes and which processes are assigned to each Pipeline Job. Also, to communicate with different Pipeline stages, inter-communicators are created to communicate the local and remote group of processes.
- **MPR uses on-demand communication:** MPR differs from DSParLib and MPI communication because it uses an on-demand approach for exchanging data stream messages between Pipeline stages. An on-demand communication can help balancing the Pipeline workload between parallel Stage Processes and enabling back pressure at the same time. However, it can also introduce extra overhead when Stage

Processes have to wait for a request message before sending data stream items to the requester. If no request messages are received, the Stage Process can become idle, reducing overall performance.

- **MPR communicates more messages:** MPR sends and receives more messages than DSParLib and MPI. Most of these messages are due to the on-demand communication protocol between each Pipeline stage, which improves load-balancing but each data stream item is associated with an extra request message. Other messages are due to MPR's runtime system coordination strategies. When network bandwidth is sufficient to support data communication without congestion, the overhead is usually negligible. However, when the network becomes a bottleneck, sending more messages can introduce significant overhead to a distributed computing system.

6.1 Application Parallelization

In this section, we briefly described how the stream processing applications were parallelized using MPR, DSParLib, and MPI. We focus on implementation details that are more relevant to understand the differences between the APIs. We selected four stream processing applications from different domains to simulate varied computational characteristics. All the streaming applications described here were implemented using a traditional stream processing Pipeline. The application generates stream items in the Source (first Pipeline stage). Then, items are processed by a computational stage that can be replicated to increase the degree of parallelism. Finally, the results are accumulated in the Sink. Some applications generate an output file (i.e., an image or a video), while others report their numeric result. The only version that does not use a Pipeline is Bzip2's MPI version. This is an existing Bzip2 implementation obtained from [38] that uses a master-worker pattern.

6.1.1 Mandelbrot Set

The Mandelbrot Set application is a mathematical program that computes a fractal within the complex plane, and it is known to have an unbalanced workload [35]. For instance, the first and last stream items of the workload have almost zero computation, while intermediate stream items require significantly more computational power to process. We selected this application because it exhibits a cyclic workload imbalance. For instance, the first stream items produced by the source have almost no computational power, which starts to increase until reaching its peak in the middle. Then, the computa-

tional intensity starts to decrease until reaching again almost no computational power at the end.

In this section, we provide a detailed explanation of the parallel Pipeline graph implementation for Mandelbrot Set, as the parallelism strategy is consistent among all stream processing applications covered in this evaluation. In the subsequent sections, we will focus only on the differences between the APIs, such as the type of data exchanged between the processes.

Figure 6.1 depicts the Pipeline Graph utilized in the parallelizations. The first Pipeline stage is always responsible for generating the data. Some applications have data generators, while others read from a file. Then, the second Pipeline stage processes the stream items. This stage can be replicated in all applications since it is stateless. Subsequently, the final stage accumulates the computed results. Some applications accumulate a numeric result value while others write in files.

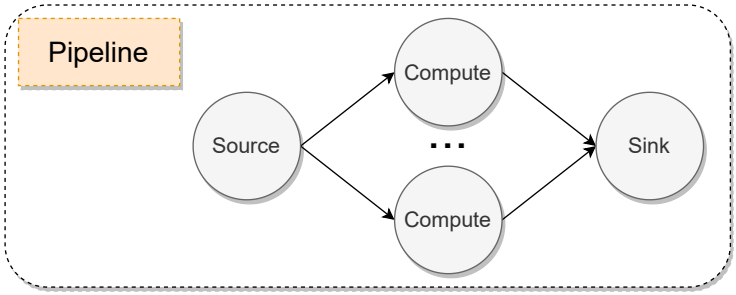


Figure 6.1: Parallel Pipeline Graph implemented in the applications.

DSParLib and MPI handwritten Mandelbrot Set implementations were obtained from [64]. Parallelizing this traditional stream processing parallel pattern with MPI is difficult and error prone. The programmer has to deal with low-level parallelism aspects and manually design and implement all the communication, scheduling, load-balancing, ordering, and others. Listing 6.1 abstract the Mandelbrot Set MPI implementation to showcase how it is implemented in MPI. Programmers first decide on how many processes will execute, then these processes are manually assigned to the Pipeline stages. The Source and Sink receive one process each, while the remaining processes are assigned to the Compute stage. After the split, the Source process generates the stream items in the loop (line 2). New stream data items are only generated and sent ahead (line 4) when a `DEMAND_MSG` is received (line 3). When there are no more data items to generate, the Source process sends a message indicating the end of the stream (line 6). Compute processes are the ones requesting data items from the Source (line 9). They probe each received message to receive its metadata before effectively receiving the payload (line 10). Each message has its tag checked (lines 11 and 12). If it is a `STOP_MSG`, propagate the end of the stream and finish. Otherwise, `DATA_MSG` is received (line 13) and processed (line 14). The results are sent using three messages to ship two integers and the mandelbrot set result (lines 15 to 17). Finally, the Sink process probes any incoming message (line 22). If the tag is

STOP_MSG, acknowledge and wait until receiving all stop messages from the Compute processes. In case of a DATA_MSG, it receives the two integers (lines 25 and 26) and the data (line 28). Other optimizations that are required in the other applications, such as ordering, must be manually implemented by programmers when using MPI.

```
1  if (isSource) {
2   for (int line = 0; line < DIM; line++) {
3    MPI_Recv(&demand, 1, MPI_INT, MPI_ANY_SOURCE, DEMAND_MSG, comm, &status);
4    MPI_Send(&line, 1, MPI_INT, status.MPI_SOURCE, DATA_MSG, comm);
5   };
6   /* abstracted application logic to send EOS */
7  } else if (isCompute) {
8   while (true) {
9    MPI_Send(&demand, 1, MPI_INT, sourceRank, DEMAND_MSG, comm);
10   MPI_Probe(sourceRank, MPI_ANY_TAG, comm, &status);
11   if (status.MPI_TAG == STOP_MSG) { ... }
12   else if (status.MPI_TAG == DATA_MSG) {
13    MPI_Recv(&line, 1, MPI_INT, sourceRank, DATA_MSG, comm, &status);
14    /* abstracted application logic to process the data */
15    MPI_Send(&line.line, 1, MPI_INT, rank, DATA_MSG, comm);
16    MPI_Send(&line.size, 1, MPI_INT, rank, DATA_MSG, comm);
17    MPI_Send(&line.M, line.size, MPI_BYTE, rank, DATA_MSG, comm);
18   }
19  }
20 } else if (isSink) {
21  while (true) {
22   MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &status);
23   if (status.MPI_TAG == STOP_MSG) { ... }
24   else if (status.MPI_TAG == DATA_MSG) {
25    MPI_Recv(&line, 1, MPI_INT, rank, DATA_MSG, comm, &status);
26    MPI_Recv(&size, 1, MPI_INT, rank, DATA_MSG, comm, &status);
27    char *data = new char[size];
28    MPI_Recv(data, size, MPI_BYTE, rank, DATA_MSG, comm, &status);
29   }
30  }
31 }
```

Listing 6.1: Example of a Pipeline implementation using MPI.

In this application. the stream item message communicated between the stages is a struct with two integers and a *char* pointer. The MPI version calls three consecutive MPI_Send for sending data and three MPI_Recv for receiving. The first two messages use MPI_INT while the last message uses MPI_BYTE for sending the data pointer by the char. DSParLib implements programming abstractions to hide some MPI communication complexities. It prevents programmers from dealing with low-level parameters such as target ranks, tags, and data types while still providing zero-copy serialization. However, DSParLib's implementation requires considerable code implementation because it is verbose.

Listing 6.2 presents the implementation of SenderReceiver abstraction to send data stream items in Mandelbrot Set with DSParLib. In this example, programmers can extend SenderReceiver (line 1) and implement both Send and Receive functions using the correct order that messages are sent and received. For sending, DSParLib's version first sends the two integers (lines 3 and 4) and then sends the dynamically allocated pointer of chars (line 6). For receiving, the same order must be respected. Then, in lines 28 to 35 we abstracted DSParLib's code to only show how serialization is employed by the Farm pattern. The Farm stages are initialized in lines 29 to 31, while the data communication is initialized in line 32. Consequently, the communication object can be placed between the Farm stages (line 33). Now, each time a Farm stage has to send data by calling Emit (line 25), it will use the template provided for Send in line 2, and for receiving, vice-versa.

```
1  class LineToRenderSerializer : public SenderReceiver<LineToRender> {
2   void Send(MPISender &sender, MessageHeader &msg, LineToRender &data) {
3    sender.SendTo(msg, data.line);
4    sender.SendTo(msg, data.size);
5    if (data.size > 0) {
6     sender.SendTo(msg, data.M, data.size);
7     delete[] data.M;
8    }
9   };
10  LineToRender Receive(MPIReceiver &receiver, MessageHeader &msg) {
11   LineToRender data;
12   memset(&data, 0, sizeof(LineToRender));
13   receiver.Receive(msg, &data.line);
14   receiver.Receive(msg, &data.size);
15   if (data.size > 0) {
16    data.M = new char[data.size];
17    receiver.Receive(msg, data.M, data.size);
18   }
19   return data;
20  };
21 };
22 class Compute : public Wrapper<LineToRender, LineToRender> {
23  void Process(LineToRender &lineToCalculate) override {
24   /* abstracted application code */
25   Emit(lineToCalculate);
26  };
27 };
28 int main(int argc, char **argv) {
29  Source s1(...);
30  Compute s2(...);
31  Sink s3(...);
32  LineToRenderSerializer comm;
33  auto farm = dspar::Farm(s1, comm, s2, comm, s3);
34  farm.Start(...);
```

```
35 }
```

Listing 6.2: Example of data serialization in Mandelbrot Set using DSParLib.

In MPR, we use different design principles to restrict the serialization step to the scope the application is sending data. Also, MPR assumes that each data block is a pair of data pointer + size. Listing 6.3 showcases MPR's implementation for zero-copy data serialization. Therefore, programmers that want to send data must create a vector of `void` pointers (line 5) and a vector of sizes (line 6). Then, programmers append to these vectors the first (lines 7 and 8) and second (lines 9 and 10) pairs of data+size. Finally, these vectors containing the data pointers and size are passed to MPR's runtime using `PublishMulti` (line 11). The counterpart of sending is succinct in MPR since no extra code implementation is required for receiving data. The Pipeline stage will receive a pointer to data (line 2), which can be cast to the desired data type (line 3). This can be done using plain C++, as shown in this example, or using MPR's auxiliary function named Unpack. To execute the Pipeline, programmers first instantiate a `PipelineGraph` (line 15). Then, they instantiate the Pipeline stages (lines 16 to 18) and append them to the Pipeline in the correct order (lines 19 to 21). The Pipeline will start executing when Run is called (line 22).

```
1  class Compute : public mpr::StageProcess<LineToRender, LineToRender> {
2   void OnInput(void * ctx, void * data) {
3    LineToRender * dataIn = static_cast<LineToRender*>(data);
4    /* abstracted application code */
5    std::vector<void *> dataOutList;
6    std::vector<int> sizeList;
7    dataOutList.push_back(dataIn);
8    sizeList.push_back(sizeof(LineToRender));
9    dataOutList.push_back(M);
10   sizeList.push_back(dataIn->size);
11   PublishMulti(ctx, dataOutList, sizeList);
12  };
13 };
14 int main(int argc, char **argv) {
15  mpr::PipelineGraph pipe(&argc, &argv, 3);
16  Source s1(...);
17  Compute s2(...);
18  Sink s3(...);
19  pipe.AddPipelineStage(&s1);
20  pipe.AddPipelineStage(&s2);
21  pipe.AddPipelineStage(&s3);
22  pipe.Run();
23 }
```

Listing 6.3: Example of data serialization in Mandelbrot Set using MPR.

The regulation to using MPR's serialization abstraction is simply paying attention when creating the data struct. Listing 6.4 shows the data struct used to exchange data

between the processes. To use MPR's serialization abstraction, the user must declare the dynamic-sized objects at the end. Each time MPR receives an extra payload message, it will append the received data to the last struct pointers.

```
1 struct LineToRender{
2   int size;
3   int line;
4   char *M;
5 };
```

Listing 6.4: Mandelbrot Set data struct.

6.1.2 Lane Detection

The Lane Detection program utilizes the OpenCV (computer vision library) to detect the boundaries of road lanes in a video stream for autonomous vehicles. The pre-existing MPI application was obtained from [77] while the DSParLib version was obtained from [64]. These versions were based on OpenCV 2, which is relatively old at the time of this dissertation. We have revised and reimplemented all versions of the Lane Detection application to be compatible with OpenCV 4.

This application requires that the input video frames are kept in order when writing the output video. In MPI's parallel version, the ordering strategy was manually implemented, while in DSParLib, ordering is enabled using `SetCollectorIsOrdered(true)` in the Farm pattern. In MPR, ordering is enabled using `EnableOrdering()` in the last Pipeline stage.

The data communication in this application is performed through the use of the OpenCV object, namely `Mat`. This data type can be confusing at first impression since it separates the object interface reference from the data reference. In this application, the `Mat` serialization is implemented by assembling the data size, then communicating its individual parts through the network. Listing 6.5 shows how serialization is achieved DSParLib. The strategy mixes DSParLib's abstractions with MPI native calls. As can be seen, each `Mat` is sent using two messages (lines 10 and 11). In order to later recreate the `Mat` object, the application must know the number of rows, columns, and type. These values are assigned to the `MPIFrame` struct between lines 5 and 8, in addition to the data size assigned in line 9. Then, the first message sends the struct containing these metadata (line 10), and the second sends the actual data (line 11). The receiver obtains the messages in the exact same order. Then it uses the metadata and the data content to recreate the `Mat` object (line 19). The MPI version is similar to this strategy, except it uses an extra message to send the data size separately while DSParLib takes advantage of the `MPIFrame` struct.

```
1  class MatSerializer : public dspar::SenderReceiver<Frame> {
2   void Send(dspar::MPISender &sender, dspar::MessageHeader &msg, Frame &frame) {
3    Mat data = frame.image;
4    size_t size = frame.rows*frame.cols*frame.channels();
5    MPIFrame mpiFrame;
6    mpiFrame.rows = data.rows;
7    mpiFrame.cols = data.cols;
8    mpiFrame.type = data.type();
9    mpiFrame.byteCount = size;
10   MPI_Send(&mpiFrame, 1, MPI_MAT, msg.target, MPI_DSPAR_STREAM_MESSAGE, sender.
       GetComm());
11   MPI_Send(data.data, size, MPI_BYTE, msg.target, MPI_DSPAR_STREAM_MESSAGE,
       sender.GetComm());
12  };
13  Frame Receive(dspar::MPIReceiver &receiver, dspar::MessageHeader &msg) {
14   MPIFrame mpiFrame;
15   MPI_Status status;
16   MPI_Recv(&mpiFrame, 1, MPI_MAT, msg.sender, MPI_DSPAR_STREAM_MESSAGE, receiver
       .GetComm(), &status);
17   unsigned char *data = new unsigned char[mpiFrame.byteCount];
18   MPI_Recv(data, mpiFrame.byteCount, MPI_BYTE, msg.sender,
       MPI_DSPAR_STREAM_MESSAGE, receiver.GetComm(), &status);
19   Mat mat(mpiFrame.rows, mpiFrame.cols, mpiFrame.type, data);
20  };
21 };
```

Listing 6.5: Example of data serialization in Lane Detection using DSParLib.

In MPR, the serialization step can be simplified using its programming abstractions. For sending data, first we set the metadata required to recreate the `Mat` object (lines 6 to 9). Then we create vectors to append the data pointers and sizes (lines 10 and 11). The metadata is appended in lines 12 and 13, while the actual data is appended in the following two lines. Finally, the programmer can call `PublishMulti` and pass the data pointers and their sizes. Complementary, when data is received, MPR passes its pointer to the application (`dataIn` in line 2), which the programmer can cast the expected data type (line 3). Finally, the `Mat` object is recreated using the metadata and the real data (line 4).

```
1  class Compute : public mpr::StageProcess<MatStruct, MatStruct> {
2   void OnInput(void * ctx, void * dataIn) {
3    MatStruct * matIn = static_cast<MatStruct*>(dataIn);
4    Mat image(matIn->rows, matIn->cols, matIn->type, matIn->data);
5    /* abstracted application logic */
6    MatStruct mat;
7    mat.rows = image.rows;
8    mat.cols = image.cols;
9    mat.type = image.type();
10   std::vector<void *> dataOutList;
11   std::vector<int> sizeList;
```

```
12    dataOutList.push_back(&mat);
13    sizeList.push_back(sizeof(mat));
14    dataOutList.push_back(image.data);
15    sizeList.push_back(image.rows*image.cols*image.channels());
16    PublishMulti(ctx, dataOutList, sizeList);
17   };
18 };
```

Listing 6.6: Example of data serialization in Lane Detection using MPR.

6.1.3 Prime numbers

Prime Numbers is a synthetic application that accumulates the number of primes within a given range [36]. We selected this application due to its highly unbalanced workload. For example, while Mandelbrot Set slowly increases and decreases its computational intensity, the Prime Numbers has peaks from one stream data item to another since even numbers are much easier to process than odd numbers. The algorithm uses brute force and assumes that a given number n is a prime number if it fails any division between 2 and $n-1$. Note that even numbers will fail right at the beginning when the division by 2 is performed. We obtained a pre-existing handwritten MPI version [38] and manually inspected it to ensure that its parallelization is similar to ours. DSParlib's version was obtained from [64]. All versions use a Source to generate the numbers, a Compute to processes prime numbers, and a Sink to accumulate those that return true. In this application, the message passing is straightforward, as it only communicates integers between the Source and Compute, and booleans between Compute and Sink.

6.1.4 Bzip2

Bzip2 is a widely used compression library in many Linux distributions. MPIBzip2 is an existing parallel version of this library that uses OpenMPI [38]. DSParLib's version was obtained from [64]. In this application, the parallel version we obtain is slightly different from our parallel activity graph. The parallelism strategy of MPIBzip2 follows a master-worker pattern, where the master task divides the input file into smaller blocks and collects them while the workers perform the compression. The difference between ours is that MPIBzip2 combines the Source and Sink into a single Master. In general, the strategies are equivalent, but results could show differences if any resource used by the master becomes a bottleneck, such as a network connection or disk.

In this application, the Source is responsible to read an input file from a distributed filesystem and split it into smaller blocks. We enable Bzip2's maximum block

size allowed, which is 900Kb per block. These blocks are then sent to the Compute processes, that process bzip2's compression algorithm. Finally, compressed blocks are sent to the Sink, which writes them in an output file. Note that the size of compressed blocks can vary since it depends on the type of data and how much can be compressed. Bzip2 demands that stream data items are ordered to preserve the correctness of the output file.

Data serialization in DSParLib is similar to other applications. Programmers extend the `SenderReceiver` and implement its `Send` and `Receive` functions. Listing 6.7 shows such implementation. Lines 4 and 5 send the data block, while lines 8 and 10 receive the data. After implementing this serialization structure in DSParLib, it can then be initialized in the main program and passed as a parameter to be used in the Pipeline communication between adjacent stages. Ordering can be enabled when instantiating the Farm parallel pattern.

```
1  class BlockOutputSerializer : public SenderReceiver<BZipBlockOutput> {
2  public:
3   void Send(MPISender &sender, MessageHeader &msg, BZipBlockOutput &block) {
4    sender.SendTo(msg, block.buffSize);
5    sender.SendTo(msg, block.in, block.buffSize);
6   };
7   BZipBlockOutput Receive(MPIReceiver &receiver, MessageHeader &msg) {
8    receiver.Receive(msg, &buffSize);
9    char *data = new char[buffSize];
10   receiver.Receive(msg, data, buffSize);
11   return BZipBlockOutput(data, buffSize);
12  };
13 };
```

Listing 6.7: Example of data serialization in Bzip2 using DSParLib.

MPR strategy for zero-copy serialization is similar to the other applications previously shown. Bzip2's data struct is presented in Listing 6.8. In the MPR version, we append references to Bzip2's buffer size and buffer payload to vectors and invoke `PublishMulti`. Ordering can be enabled in the Sink process, therefore, programmers do not need to implement extra logic for that.

```
1 struct bzip2Block {
2     int bufferSize;
3     char * buffer;
4 };
```

Listing 6.8: Bzip2 data struct.

6.2 Methodology and Environment

In this section, we describe the methodology adopted to execute the performance experiments. Moreover, we describe the environment used in the experiments. The tests were executed on LAD (Laboratório de Alto Desempenho) at PUCRS. We used their cluster that has newer software installed since we need OpenMPI newer releases due to bug fixes in `MPI_Comm_spawn` and in Slurm integration. The cluster name is Pantanal, and it has four nodes. We allocated all of them for the experiments. Each node has two Intel Xeon Gold 5118 @ 2.30GHz (12 cores, 24 threads) with 192GB of RAM memory. The nodes are interconnected using four Gigabit-Ethernet networks. Also, each node uses a single Gigabit-Ethernet network to access data from the distributed file system. InfiniBand is not available in this cluster. The applications are compiled using GCC version 9.4.0 with the optimization flag -O3 enabled. For the computer vision application, we use OpenCV version 4.7.0. OpenMPI is version 4.1.1, and Slurm is version 19.05.5.

In our experiments, we compare parallelizations using MPR framework with respect to DSParLib and MPI versions. We measured the average throughput of the four applications described in Section 6.1. The throughput was computed via the relation between the total number of stream data items and the total time execution, namely `num_items/time`. We check the result correctness of all applications using MD5 hash and compare them against a sequential version, except for prime numbers, which result can be inspected since it returns the total number of primes.

The result graphs showcase the mean throughput of three execution. The standard deviation is reported using error bars and may not be visible when the value is negligible. The x-axis shows the number of Compute processes. Note that the graphs do not depict the total number of active processes running in the cluster. For example, we do not consider the Source and Sink to plot the graphs. This decision extends to all versions: MPR, DSParLib, and MPI. Moreover, in MPR, we do not consider the Manager Processes to plot the graphs.

OpenMPI enables us to configure the process allocation strategy. We tested varied configurations using custom `hostfiles` with different mapping fashions and selected the best one to report on the results. Therefore, we reserved two nodes out of four from our cluster to isolate the Source and Sink processes. For MPR, we have mapped all the Manager Processes (four processes in our experiments) within the same node as the Source. In the remaining two nodes, we allocate all the processes that will compute the application. We configured each node to allow only processes equal to its total number of physical cores. This means that each node can allocate up to 24 Compute processes, totalizing 48 Compute processes in two nodes. In addition to that, we configured OpenMPI to map by slots and to bind processes to cores. This configuration choice was also made

by DSParLib's work [64], but their results show better scalability since they were able to run in a cluster with InfiniBand. However, the OpenMPI version they used was OpenMPI 1.4.5, which release dates from 2011.

We report the results in the following order: (1) In Section 6.3, we compare MPR's performance against DSParLib and handwritten MPI. The goal of this experiment is to position MPR against state-of-the-art solutions. We configured MPR to avoid many of its communication messages between Stage Processes and Stage Managers. However, some of the communications continue to be active during the Pipeline execution. We expect MPR to introduce higher overhead because it is the only dynamic runtime, while the other two are static. (2) In Section 6.4 we executed the same MPR version using different rates for Stage Processes to communicate with Stage Managers. The goal is to investigate the overhead impact of the configuration messages exchanged between Stage Processes and their Managers. (3) Finally, in Section 6.5 we report results with different internal data buffer size configurations. The goal is to assess the impact of the buffer size on the performance.

6.3 MPR Performance Evaluation

This section presents the results of our performance evaluation. Figure 6.2 shows the throughput achieved by the Mandelbrot Set streaming application using MPR, DSParLib, and handwritten MPI implementations. The workload used in this application consisted of 6,000 stream data items with a computational intensity of 10,000 iterations. Our results indicate that the parallel versions of the application perform similarly across all three implementations. Specifically, the application scales up to 48 processes, and MPR's performance is comparable to that of DSParLib and MPI for applications with similar characteristics to the Mandelbrot Set.

Figure 6.3 showcases a different situation. As can be seen, the application scales only up to eight Compute processes in all versions. Moreover, the MPR version performs slightly worse, up to 6.1% less throughput, compared to the others. In Lane Detection, the Source process read an input video and publishes in the Pipeline each frame as a new stream data item. Due to a large amount of data being transferred through the network, we attribute the lack of scaling as a consequence of network congestion. DSParLib's work [64] did not observe such behavior in their experiments. However, their tests were executed in a different cluster equipped with InfiniBand. The cluster we use in our experiments does not provide an InfiniBand network. Since the network is already congested, the extra messages MPR uses for communication are the reason for achieving lower throughput compared with others. DSParLib and MPI perform similarly, although, in some degrees of parallelism, DSParLib achieves up to 1.1% less throughput than MPI. MPR and DSParLib use extra messages to communicate data between stages. Their com-

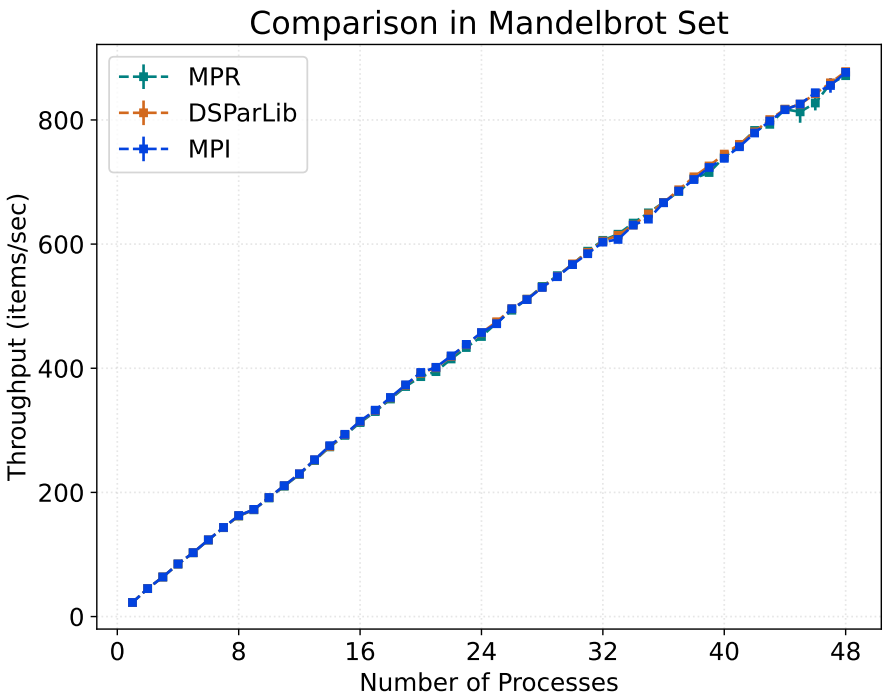


Figure 6.2: Mandelbrot Set throughput comparison.

munication starts with a header message followed by the payload data messages. This explains the reason MPI achieves better throughput in Lane Detection.

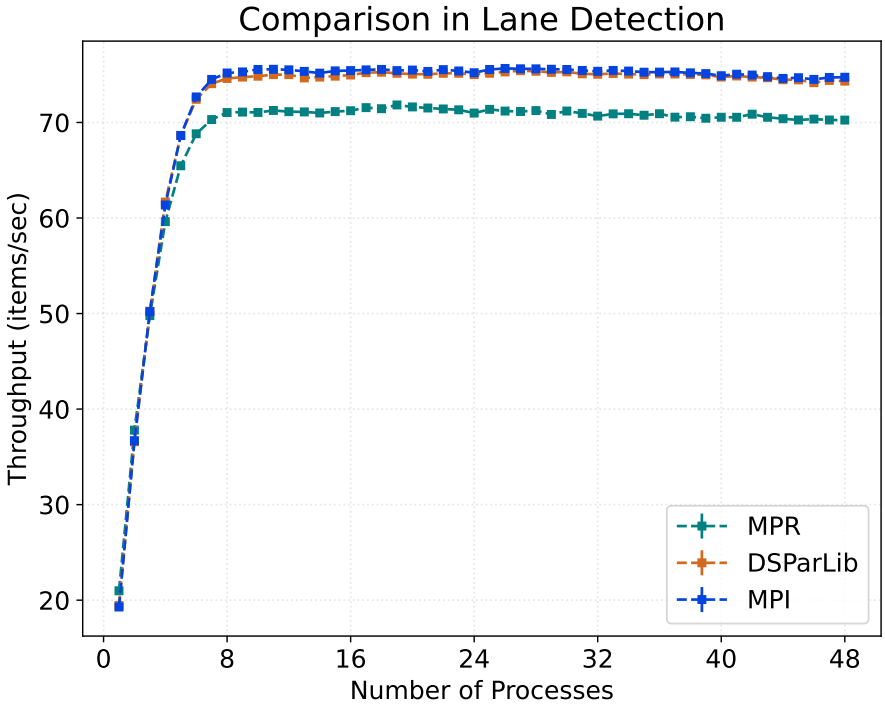


Figure 6.3: Lane Detection throughput comparison.

Figure 6.4 presents the performance results obtained in the Prime Numbers application. At this point, we have already noted that the network bandwidth has a significant impact on performance due to our cluster configuration. The same explanation we gave

before about the number of messages applies here. In fact, Prime Numbers exhibits better the impact of both MPR and DSParLib using an extra header message. In Prime Numbers, the data exchanged between processes is very small (only an integer or boolean). Therefore, adding the header message that carries extra data can increase the overhead. For instance, MPI sends a single integer (4 bytes) while DSParLib sends an extra header (20 bytes) in addition to the data integer (4 bytes). MPR's header uses 12 bytes in addition to the data integer. However, MPR shows a higher overhead, which results in a throughput 10.1% lower than DSParLib and 33.4% lower than handwritten MPI at the maximum degree of parallelism (48 Compute processes). One of the differences MPR has with respect to the other versions is that it implements an on-demand protocol in the Sink while others do not. Therefore, additional idle time exists between the time the Compute processes finished their processing and a new request is sent. For instance, in Prime Numbers, some stream data items(i.e., even numbers) process almost instantly. Therefore, the Compute process could start processing another data item, however, since we implemented on-demand, it must first wait for a request message. These results are interesting and give us new insights regarding MPR optimizations. We already have two ideas that will be tested in the future: (1) The first alternative is removing the on-demand protocol from the Sink. In our definition, the Sink must always be available to accumulate the results. If any computational logic is needed, it must be moved from the Sink stage into an extra Compute stage that precedes the Sink. (2) Implement a second data buffer for the output connections. Currently, MPR implements a buffer only in the input connection in such a way that the input raw messages are stored until request messages ask them to be computed. By introducing an additional data buffer in the output connection, a Stage process can already process a number of messages that equals the output data buffer size and keep their results buffered until a request message arrives.

Our main focus at this stage is to assess the impact of the on-demand algorithm and to design the mechanisms that can be applied when MPR is extended to support multiple Pipeline stages. Our plan is to relocate the mechanism currently implemented between the Compute and Sink stages to be utilized between two Compute stages. However, we have also noticed that the on-demand protocol could become a bottleneck when it is employed by sequential stages. For example, since there is only one Sink process to receive messages, Stage Processes will always send data to this single process which does not improve load-balancing. We will take this into account for future extensions of MPR. By doing so, we expect to enhance MPR's capability to handle more complex Pipeline graphs and improve the performance of various stream processing applications.

Figure 6.5 presents the results for Bzip2 compression application. In this application, we observed the same behavior as Lane Detection and Prime Numbers, that the network impacts the performance. In Bzip2, the Sink application code is responsible for writing the compressed data blocks on the disk. In our cluster configuration, writing to a

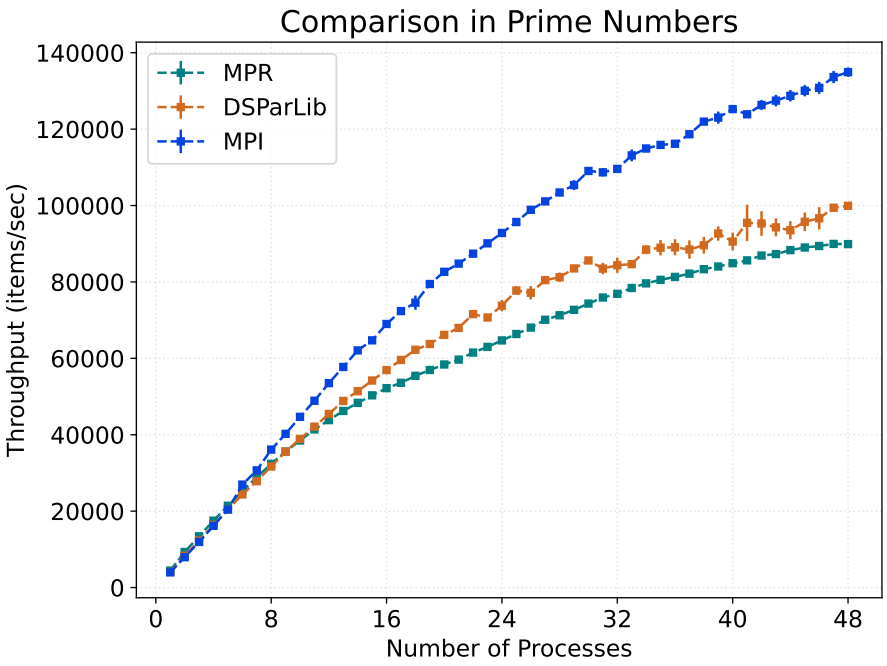


Figure 6.4: Prime Numbers throughput comparison.

distributed file system transfers data using a single Gigabit-Ethernet network. As can be seen in the graph of Figure 6.5, some degrees of parallelism show higher values for the standard deviation due to the variation in network congestion and disk usage. We can observe that MPI is the version with higher variation compared to the other two versions. Both MPI and DSParLib achieve similar performance in most parallelism degrees. MPR is the version that achieves the worse throughput. At the maximum degree of parallelism (48 Compute processes), MPR achieves a throughput 22.4% lower than MPI and 25.4% lower than DSParLib. These results reveal that maintaining the on-demand strategy currently implemented for the Sink shows significant overhead in some situations. For example, in Bzip2 the Sink stage writes the compressed blocks in the disk. Meanwhile, it cannot send enough requests to avoid Compute processes from becoming idle. There is a gap between the moment the Sink receives the data and the moment it sends a new request to the Compute processes, making them becoming idle in the meantime.

6.4 MPR Management Evaluation

In addition to the comparison results between MPR and other parallelizations, we tested some MPR mechanisms and their impact on the application execution. Figure 6.6 showcases the results for different applications when changing the communication rate of Stage Processes. In the previous experiments, we increased the period to a value that would make Stage Processes never communicate with their Stage Managers. In Figure 6.6, this is represented by the *Never* label. Additionally, we configured the processes to com-

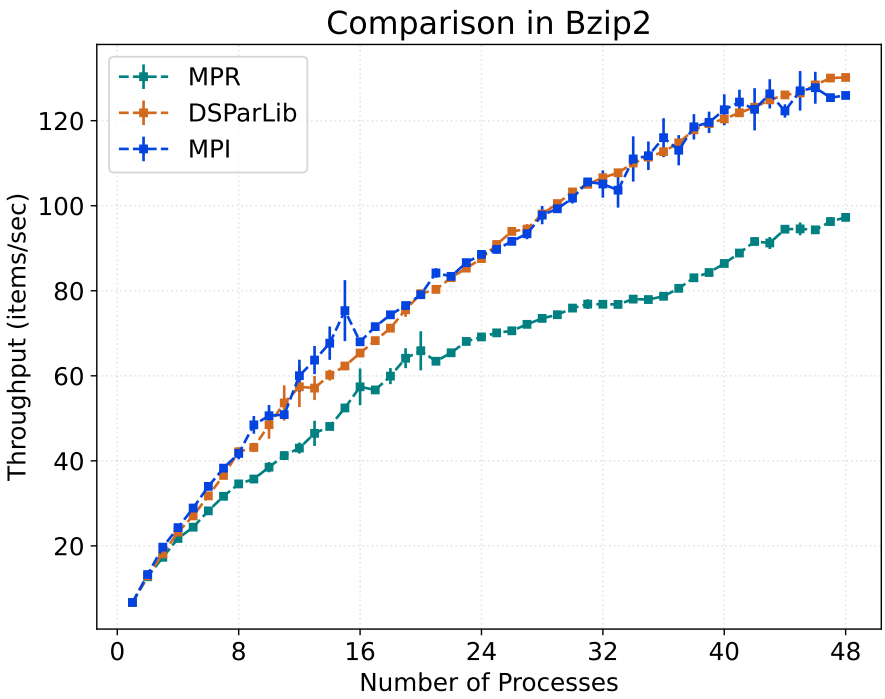
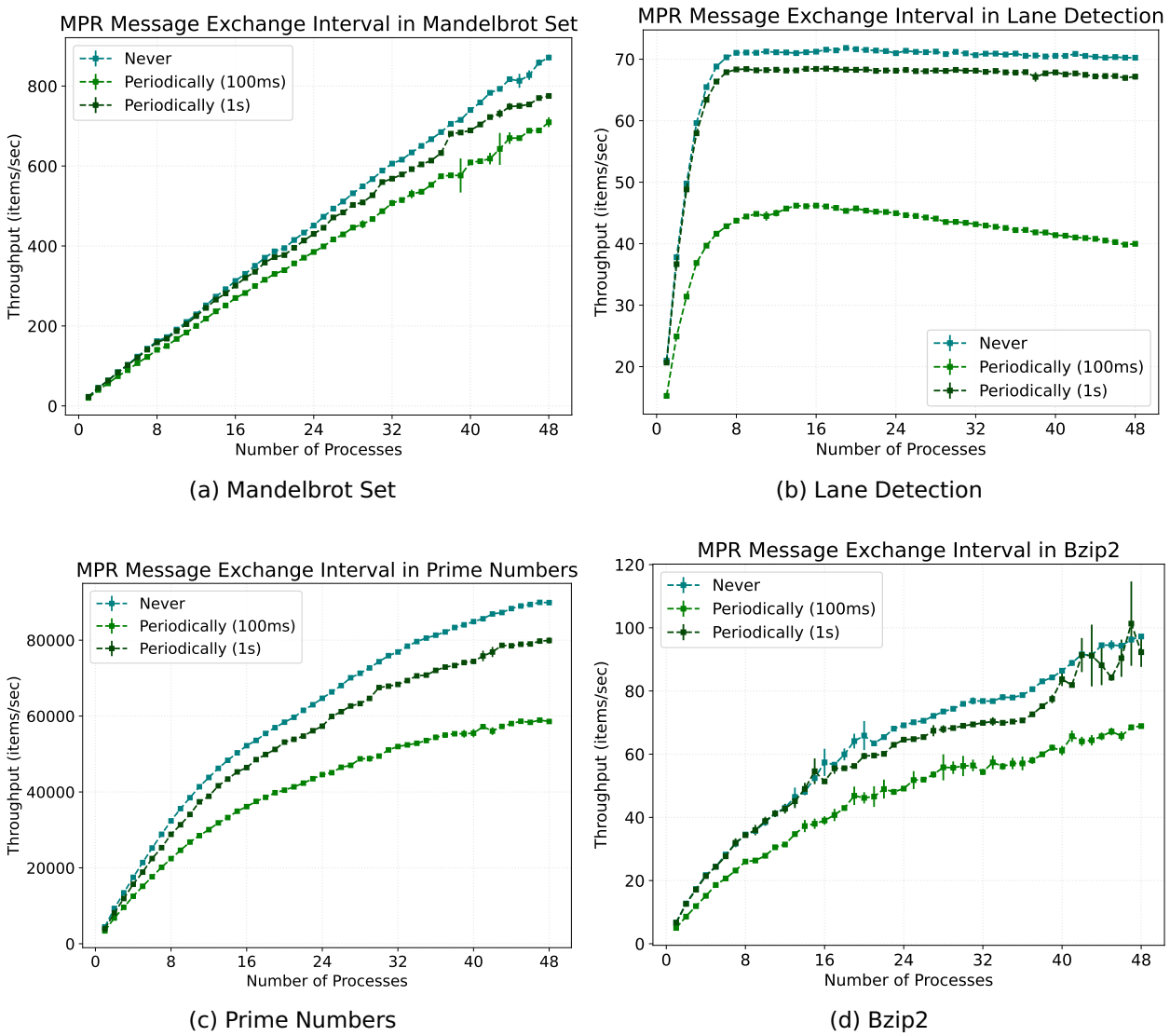


Figure 6.5: Bzip2 throughput comparison.

municate each 0.1 seconds (*Periodically (100ms)*) and each 1 second (*Periodically (1s)*). It is worth noting that it is not guaranteed that Stage Processes will communicate exactly every 100 milliseconds, for example. The reason is that processing a certain stream data item can take longer than the specified period. In that case, MPR cannot stop the application execution at any point and synchronize with the Managers. Instead, it finishes executing, and after that, it checks if the period between the last message exchanged (with the Manager) and the current clock is greater than the specific period (i.e., 100ms). MPR ensures that after a given interval period, based on the local Stage Process clock, Stage Processes will make an effort to synchronize. However, there is no guarantee that they will communicate at the exact same time. For example, if the period is set to 100 milliseconds, some processes will communicate at this exact moment while others communicate at 150 milliseconds. No process blocks another during normal execution, they simply communicate their status and continue the Pipeline execution. If synchronization is required (i.e., adding or removing processes), then processes are blocked until all processes arrive.

Figure 6.6 shows that the communication rate can have a significant impact on performance, especially when the network becomes a bottleneck. Figures 6.6a and 6.6d are applications that showed less impact due to network congestion. To be precise, Bzip2's performance has deteriorated due to congestion in the single network that connects the nodes to the distributed file system. Remember that our cluster uses four network links, so the other three are available. In both applications, the configuration rate of 1 second showed similar trends to the version that never communicates. This happens until the



(a) Mandelbrot Set

(b) Lane Detection

(c) Prime Numbers

(d) Bzip2

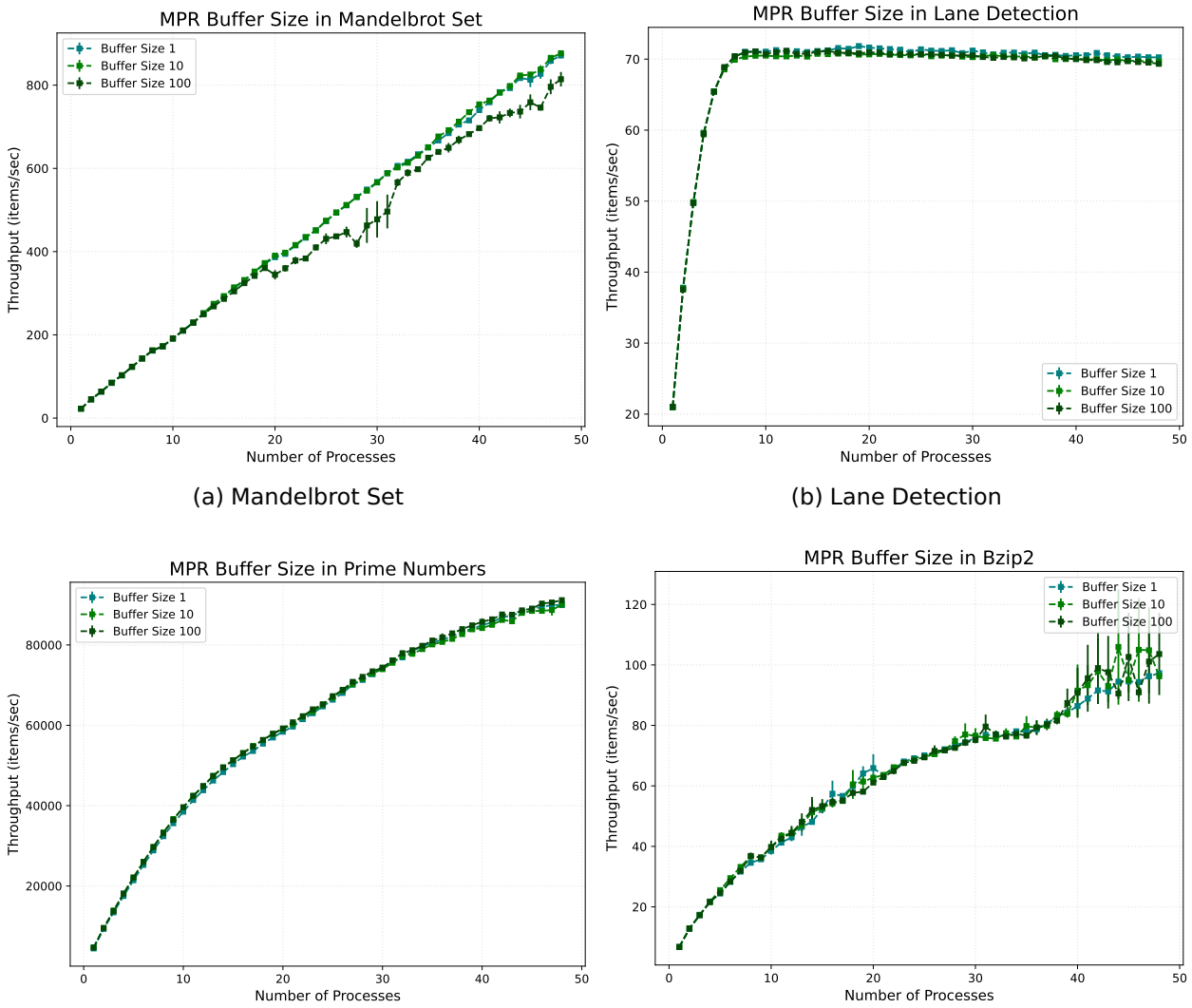
Figure 6.6: Evaluating the impact of different communication rates.

degree of parallelism with 16 Compute processes. After that, the throughput starts to decrease. For example, in Figure 6.6a the throughput at the maximum degree of parallelism with 1-second periods is 11.1% lower than the version that never communicates. When the rate is increased 10 times (100ms), the throughput is 18.6% slower than the version that never communicates. In Bzip2, since most Stage Processes are idle when they do not receive requests from the Sink, extra communications during the idle time do not decrease the throughput. However, when the communication becomes too frequent, we can observe that it can impact on performance up to 29.2 lower throughput than the version that never communicates. In the other two Figures 6.6b and 6.6c, we can observe that the exchange message period has a higher impact since network congestion is a problem. In Lane detection, the throughput of the most frequent version is up to 44.3% lower than the best MPR version. In Prime Numbers, the difference is smaller but still significant,

achieving a peak of 34.8% worse throughput compared to MPR best version that never communicates.

6.5 MPR Buffering Evaluation

Finally, we conducted a supplementary experiment to investigate the impact of buffer size on application execution. In the MPR version used in the comparison with DSParLib and MPI, we configured the buffer size to be 1. In the graphs of Figure 6.7, we added two additional buffer size configurations: buffer size 10 and buffer size 100. These sizes stand for the number of stream items that can be buffered at a time. As can be seen in the Lane Detection and Prime Numbers applications (Figures 6.7b and 6.7c), the different buffer size configurations produced comparable throughput measures. In the Prime Numbers application, we expected the higher buffer size configuration to decrease throughput since the workload is highly unbalanced. For example, if a process fills its 100-sized buffer with computationally intensive prime numbers, the workload will become unbalanced because all other processes must wait for this one. However, we believe that network congestion mitigated the issue of unbalanced workload. In the Mandelbrot Set application shown in Figure 6.7a, both buffer sizes 1 and 10 show similar performance, while the buffer size of 100 decreases application performance. In this application, some processes buffered too many computationally intensive stream data items, which unbalanced the workload. The larger the buffer size, the greater the likelihood of the application becoming unbalanced. However, buffering items reduces idle time as data is always ready to be processed. Finally, Figure 6.7d shows the results for Bzip2, where both buffer sizes (10 and 100) showed comparable performance until the end, when both versions introduced load-balancing issues to the application execution. The problem with MPR in Bzip2 was that requests were not sent rapidly enough by the Sink, causing Compute processes to become idle. We selected a workload for Bzip2, an ISO file of size 704 MB, which, after compression, becomes 658 MB. Using Bzip2's default block size of 900 KB, we have a total of 783 stream data items. For example, if processes are quick enough, only 8 processes of buffer size 100 can take all the workload. This explains the behavior of Bzip2. The results demonstrate that adding more processes does not significantly improve application performance since the main bottleneck is the Sink. In fact, having fewer processes computing makes them less idle as the Sink has fewer processes to request data for.



(a) Mandelbrot Set

(b) Lane Detection

(c) Prime Numbers

(d) Bzip2

Figure 6.7: Evaluating the impact when varying internal data buffer sizes.

6.6 Final Remarks

In this section, we presented a performance evaluation to position MPR with respect to state-of-the-art solutions. Moreover, we evaluated two MPR features, one is the rate at which processes synchronize, and another is the data buffer size.

First, we presented a performance evaluation of MPR by comparing its parallelization results with equivalent DSParLib and manually implemented MPI versions. The results showed that MPR achieves comparable throughput with respect to the others in Mandelbrot Set. However, we observed that in the remaining applications, the network becomes a bottleneck due to our cluster configuration. Therefore, MPR's performance measure is worsened due to the additional messages it uses to operate the Pipeline. In these cases, MPR's overhead is higher than the other versions. Furthermore, the experiments with

Bzip2 revealed that additional application code in the Sink can introduce idle time in MPR's Pipeline execution due to the on-demand scheduler. These results provided new insights that will aid in enhancing MPR in future versions. With these experiments, we can conclude that MPR offers negligible overhead in applications that are not communication-intensive, such as Mandelbrot Set. But it can introduce extra overhead when communication is intensive, and the network bandwidth is a bottleneck. Further experiments in a cluster with more nodes and higher network bandwidth will be conducted to refine our analysis and obtain more accurate conclusions.

In MPR's features evaluation, the experiments revealed that configuring different values can impact the application's performance. The experiments, when changing the rate at which processes halt the Pipeline normal execution and communicate with the managers, showed that the impact is significant. These results suggest that a self-adaptive algorithm can exploit this feature and dynamically set the best rate on-the-fly. For example, when the Pipeline execution is balanced and stable, the synchronization rate can be lower. However, when the Pipeline execution is experiencing workload variation, the rate can be increased, trading-off between throughput and responsiveness. In our experiments with Mandelbrot Set, the throughput trade-off is 11% for a 1 second rate compared to the version that never communicates. In other applications, since the network is a bottleneck, we obtained throughput measures up to 44.3% lower than the version that never communicates. In addition, the experimental evaluation using different buffer size configurations revealed that it can impact the application execution performance. Our experiments showed that increasing the buffer size can negatively impact throughput, as larger buffer sizes tend to introduce load-balancing issues. In our experiments, we cannot observe throughput gains since the source is always available to send new items as the input is already loaded in memory. Therefore, buffering them has almost no impact on performance. Nonetheless, in real-world applications, sources can delay sending items, and we expect the buffer size to have a greater impact in this situation.

MPR is an ongoing project that we plan to continue contributing to in our research group. Future versions of MPR will address the insights we have gained from our experiments. Overall, our experiments have shown that MPR's dynamic runtime system can achieve performance comparable to a static MPI implementation. The performance degradation we observed in our results is mostly due to extra messages MPR is sending to coordinate the Pipeline execution. There is the possibility of reducing the number of messages sent by removing some mechanisms, such as the on-demand feature. However, we are not convinced this is the best approach. Our experiments were conducted on a four-node cluster with limited network bandwidth, which limits the expressiveness of our findings. A second set of experiments in a different cluster is mandatory to improve our understanding of MPR's runtime system behavior. And from that point, we will take proper actions to optimize MPR's runtime system.

7. MPR ADAPTABILITY EVALUATION

In this chapter, we present the evaluation of MPR's external autonomic management module. Our goal with this experimental implementation is to investigate if the current MPR interface shows flexibility and portability for programmers implementing self-adaptive algorithms. We have implemented a self-adaptive algorithm based on a Monitor-Analysis-Planning-Executing (MAPE) feedback loop [79].

In the Monitoring stage, the prediction models are fed with the current state of the streaming application, including the environment information and the user goals. The analysis stage compares the gathered application statistics with the user-specified goals. Depending on the parameters, these comparisons implement complex models; for instance, the goal can be either a simple throughput measure or a quality attribute such as maximum efficiency. Planning determines the runtime system adaptations necessary to satisfy the goals. Creating a plan is challenging when the self-adaptive runtime system's parameters do not exhibit a clear output effect. For example, increasing the number of parallel replicas executing a given application does not guarantee performance improvement. Some complex models may be required for this step, such as machine learning techniques. The execution step, finally, synchronizes with the runtime system and applies the actions decided by the self-adaptive algorithm.

In this work, we designed MPR to ease the synchronization steps of the first and last MAPE stage activities: Monitoring and Execution. MPR provides Pipeline execution metrics for all stages and reads configuration metrics that automatically reconfigure the runtime system from Json files. Any MAPE self-adaptive system implementation can interact with these files to adapt the Pipeline Graph execution.

7.1 Self-adaptive Algorithm

Listings 7.1 and 7.2 present the algorithm, split into two parts. The first part, depicted in Listing 7.1, represents the monitoring step. It reads from the output statistic files generated by MPR. The first lines of code (lines 1 to 7) provide synchronization so that the self-adaptive algorithm and the Pipeline execution can run concomitantly. The implemented strategy periodically tries to open the statistics file (line 2). If an exception is thrown, it sleeps for 1 second (line 6) and tries again. The algorithm will successfully open the statistic file when the Pipeline Graph starts executing and MPR starts generating output to this file.

The Pipeline statistics are organized in different output files. There is one file for each Pipeline stage. The files are periodically updated by the Stage Managers, each

one is responsible for its own Pipeline stage statistics file. These files contain the number of items consumed and produced in a time interval, which interval can be configured. However, MPR does not guarantee that the files are updated exactly at the time interval specified by the programmer. In addition to the time interval, each Stage Manager only updates its statistics when all Stage Processes have reported their consumed and produced items. Due to this characteristic, we adjusted the self-adaptive algorithm. As shown in Listing 7.1, after MPR starts generating the statistics, the self-adaptive algorithm will start reading them. Since statistics are not generated regularly, we designed the algorithm to accumulate all the timestamp results between its last timestamp read and the current timestamp. Timestamps are generated in milliseconds with the format: `1676593622000`. So, there is a chronological order between them. The first time our algorithm reads the stats file, the `lastTimestamp` fails in the condition of line 9 and skips the computation, moving to the next loop iteration.

```
1  while (true) do
2   try
3    fileIn = open("stats_stage2.json")
4    break
5   exception
6    sleep(1)
7  end while
8  itemsProduced = 0
9  if lastTimestamp then
10  startCounting = false
11  for currentTimestamp in json["stage2"] do
12   if startCounting then
13    for compute in json["stage2"][currentTimestamp] do
14     itemsProduced += json["stage2"][currentTimestamp][compute]["ItemsProduced"]
15    end for
16   end if
17   if (currentTimestamp == lastTimestamp) then
18    startCounting = true
19    initTimestamp = currentTimestamp
20   end if
21  end for
22 end if
23 lastTimestamp = max(json["stage2"])
24 duration = (lastTimestamp - initTimestamp) / 1000
25 throughputApplication = itemsProduced/duration
```

Listing 7.1: Implementation of MPR's adaptability algorithm (Part 1).

In the next iteration, the execution has stored its `lastTimestamp` and executes the logic between lines 9 and 22. For that, the self-adaptive algorithm iterates all timestamps generated by MPR. When it reaches the `lastTimestamp`, it enters the condition of line 17. Then, `startCounting` is set to true (line 18) and `initTimestamp` is set. From this

point on, we start counting/accumulating all the produced items between the `initTimestamp` and the maximum timestamp generated by MPR. The counting considers all Compute processes (line 13) since the number of active processes can change. After accumulating the total number of items produced, it updates the `lastTimestamp` (line 23) and computes the time interval between the initial and last timestamps (line 24). We use the total number of items and the duration to compute the application throughput (line 25).

After we have processed the current application throughput in the monitoring step, we compare this metric with the user-specified goals. Currently, our self-adaptive algorithm tries to scale up or scale down the application in order to reach a throughput goal. Listing 7.2 showcases the second part of the algorithm, which implements the analysis and planning steps. The comparison analysis between the application and goal throughputs is done in lines 3 and 10. For each situation, we implemented proper actions. For example, if the application throughput is higher than the goal (line 3), then the application has to scale down, meaning that processes should be removed. We have included a static model to our self-adaptive algorithm, which uses four scale factors (lines 5 to 8). If the application has a throughput 100% higher than the goal, we remove 4 processes (line 5). If the throughput is 50% or is 25% higher than the goal, we remove 2 and 1 process, respectively. Otherwise, we keep the same amount (line 8). After, we use the values read from the `parameters.json` file (lines 1 and 2) and update them accordingly (line 9).

```
1  fileIn = open("parameters.json")
2  json = fileIn.read()
3  if (throughputApplication > throughputGoal) then
4   currentValue = json["stage2"]
5   if (throughputApplication/throughputGoal > 2) then factor = 4
6   else if (throughputApplication/throughputGoal > 1.5) then factor = 2
7   else if (throughputApplication/throughputGoal > 1.25) then factor = 1
8   else then factor = 0
9   if (currentValue > factor) then json["stage2"] = currentValue - factor
10 else if (throughputApplication < throughputGoal) then
11  currentValue = json["stage2"]
12  if (throughputApplication/throughputGoal < 0.5) then factor = 4
13  else if (throughputApplication/throughputGoal < 0.75) then factor = 2
14  else if (throughputApplication/throughputGoal < 0.9375) then factor = 1
15  else then factor = 0
16  if (currentValue+factor < 48) then json["stage2"] = currentValue + factor
17 end if
18 fileOut = open("parameters.json")
19 fileOut.write(json)
```

Listing 7.2: Implementation of MPR's adaptability algorithm (Part 2).

When the application throughput is lower than the goal, our self-adaptive algorithm adds more processes (lines 10 to 17). The logic is equivalent to the previously explained part. However, we use different configuration thresholds. If the application

throughput is 100% lower than the goal, we add 4 processes (line 12). If the application throughput is 25% lower, or 6.25% lower, we add 2 and 1 processes, respectively. Otherwise, no new process is spawned by MPR (line 15). We set 48 as the maximum number of allowed processes because it represents the number of available physical cores in our cluster (line 16). Finally, the updated values are written back to the `parameters.json` file. Eventually, it will be read by MPR, which completes the MAPE loop by executing the adaptation modifications. Note that we use different values to decide the adaptation factor for adding or removing processes. In MPR, the synchronization required for removing processes is heavier than the one for adding processes. Therefore, we configure the deletion thresholds to be higher than the addition thresholds. Some complexities arrive when removing processes: (1) their data buffer needs to be emptied before they can be removed. In MPR's current version, each Stage Process must empty its buffer by processing all data. In the future, an optimization can be implemented enabling data to be sent to another active process instead of processing it. (2) Processes must destroy all communicators and recreate them with fewer processes. MPI does not offer any interface to shrink communicators.

7.2 Methodology and Environment

In this section, we describe the methodology adopted to assess MPR's adaptability support. Moreover, we describe the environment used in the experiments, which is the same as the previous performance experiments. It is a cluster with four nodes, each node has two Intel Xeon Gold 5118 @ 2.30GHz (12 cores, 24 threads) with 192GB of RAM memory. The nodes are interconnected using four Gigabit-Ethernet networks. Also, each node uses a single Gigabit-Ethernet network to access data from the distributed file system. InfiniBand is not available in this cluster. The applications are compiled using GCC version 9.4.0 with -O3 enabled. OpenMPI is version 4.1.1, and Slurm is version 19.05.5. OpenCV is version 4.7.0.

In our experiments, we characterize MPR's adaptability by running the application's Pipeline Graph we previously presented (Mandelbrot Set, Lane Detection, Prime Numbers, and Bzip2) along with our self-adaptive algorithm. MPR is written in C++, while the self-adaptive algorithm is written in Python. We ran experiments for each streaming application multiple times and selected the two most representative results. Differences exist between the executions because distributed environments are unpredictable. Therefore, the self-adaptive algorithm must be ready to deal with this dynamic environment and make optimal decisions to achieve the system's goal.

The result graphs showcase the mean throughput variation when the number of processes is adapted. The x-axis represents the timestamps reported by MPR during the

Pipeline execution. The number of timestamps varies depending on the application's total execution time. Then, the right y-axis shows the number of processes. This number refers to the amount of active Compute processes, we do account for the Source, Sink, and Managers. The number of processes is depicted using a step curve, where each point refers to a timestamp. The left y-axis showcases three information: the mean throughput (gray columns), and the throughput goal and its trend (lines). The mean throughput is calculated for each timestamp reported by MPR. The throughput trends are obtained using a function that fits the data within a 4th-degree polynomial function using the least squares method. In addition, the throughput goal is the configured value for each application in the self-adaptive algorithm.

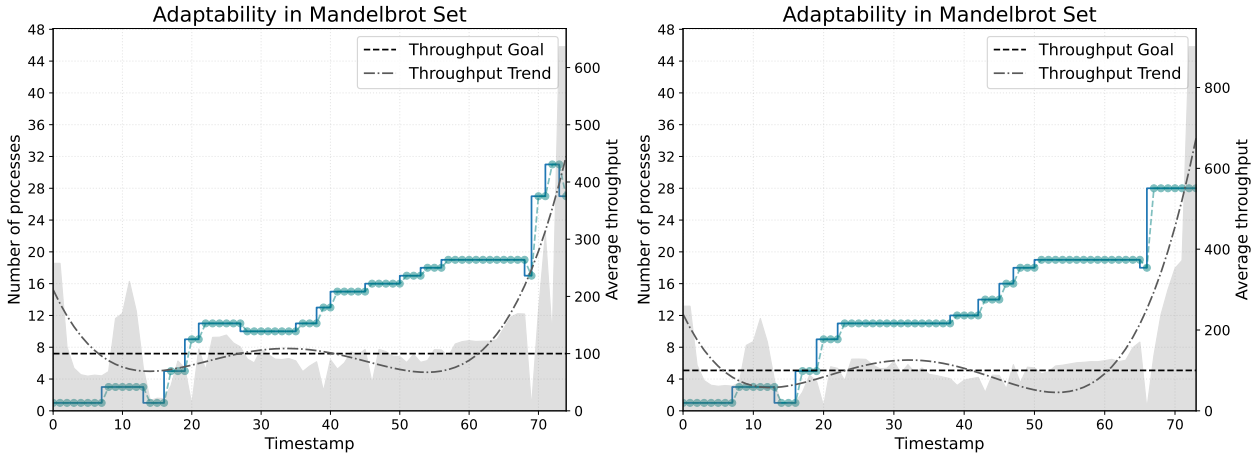
We configured MPR with an intrusive reconfiguration strategy that makes the runtime system frequently adapt the number of processes only for experimental reasons. A practical distributed application would configure less frequent self-adaptations. We used a time interval of 100ms for the Stage Processes to communicate with their Stage Managers. Then, we set each Stage Manager with a time interval of 300ms to write the statistics in the Json file. In addition, we configured the Pipeline Manager to read from the `parameters.json` each 500ms to check the number of processes and reconfigure the runtime accordingly. Finally, we configured the self-adaptive algorithm to read the statistics with a time interval of 1 second. MPR runtime systems can potentially adapt each 1 second, which is a relatively elevated number of reconfigurations for a distributed system. For example, each reconfiguration halts the Pipeline application execution until the number of processes is adapted. The data internal buffer size is set to 1.

7.3 Adaptive Strategy Characterization

In this section, we report the self-adaptive strategy characterization results. We aim to conduct a first experimental evaluation to investigate if MPR is flexible and can self-adapt the runtime system when integrated with self-adaptive algorithms. We do not perform analysis on performance because this is the first time a self-adaptive algorithm is implemented on MPR. We have considered comparing with the self-adaptive mechanisms proposed in [47], a state-of-the-art strategy for adapting the number of processes. However, it was not feasible due to the short time of the master's studies. Therefore, we aim to evaluate MPR's self-adaptive interface and its capability to automatically reconfigure the runtime system by adding and removing processes on-the-fly.

7.3.1 Mandelbrot Set

Figure 7.1 depicts the results for two executions in Mandelbrot Set. The mandelbrot pattern makes the first and last stream items less computationally intensive. This explains why the throughput beginning and end tend to be high. Initially, the throughput is not expressive because we start the application execution with a single Compute process. When the throughput falls below a certain threshold, the self-adaptive algorithm adds two more processes (before timestamp 10). The application scales well, as these processes are quickly removed (after timestamp 10). Since the synchronization to remove is heavier, the overall application throughput falls (around timestamp 15). After that, the self-adaptive algorithm starts adding more processes until the application stabilizes. More processes are added after timestamp 40 since the computational intensity is slowly increasing. But in the end (after timestamp 70), the workload is very small, and the overall throughput increases. The execution from Figure 7.1a catches the pattern and starts decreasing the number of processes, while the second execution from Figure 7.1b did not. After that, the application execution finishes.



(a) Execution 1

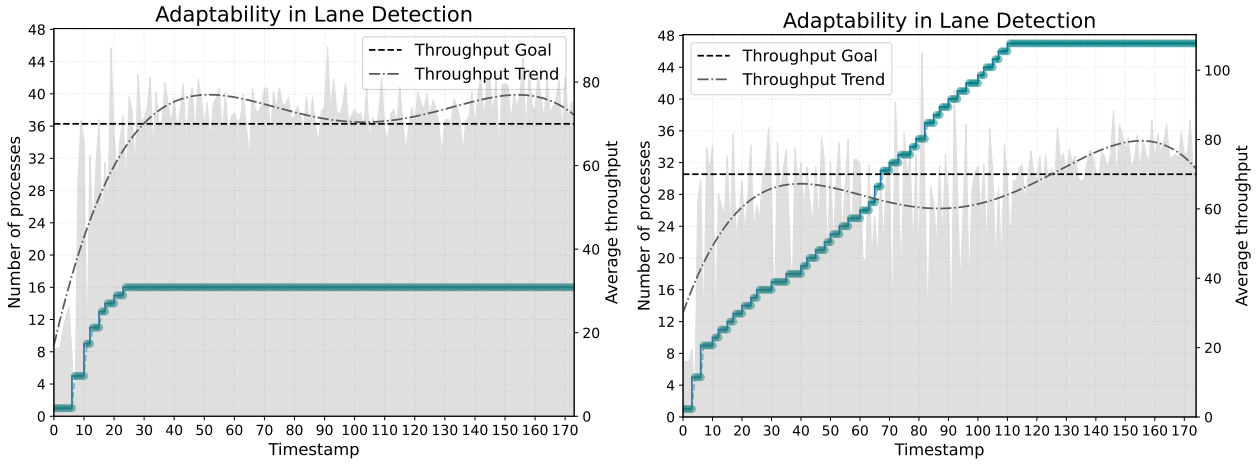
(b) Execution 2

Figure 7.1: Adaptive strategy characterization on Mandelbrot Set.

7.3.2 Lane Detection

Figures 7.2 depicts the two execution results for the Lane Detection application. In this application, we present two different self-adaptive characterizations the self-adaptive algorithm provides. In our adaptability strategy, we assume that adding new processes always increases throughput while removing processes decreases it. However, this is not the case in Lane Detection. The previous performance comparison results revealed

that Lane Detection does not scale until the maximum degree of parallelism. In fact, it stops scaling early, around 8 Compute processes. Therefore, the first execution depicted in Figure 7.2a shows the self-adaptive algorithm stabilizing after setting 16 processes. In the first two decisions (before timestamp 10), it adds 8 more processes to increase the throughput. Since the reconfiguration adds extra overhead, which reduces the overall throughput of the application, the algorithm adds more processes until 16. The second execution results are illustrated in Figure 7.2b. As can be seen, in this case, the self-adaptive algorithm keeps adding more processes until it reaches the maximum threshold. First, its decision is not optimized since it considers adding more processes to increase performance. Secondly, by making the runtime repeatedly self-adapt, it introduces configuration overhead that decreases the overall application throughput. Therefore, it never achieves its goal during adaptation. However, the application throughput stabilizes after timestamp 110.



(a) Execution 1

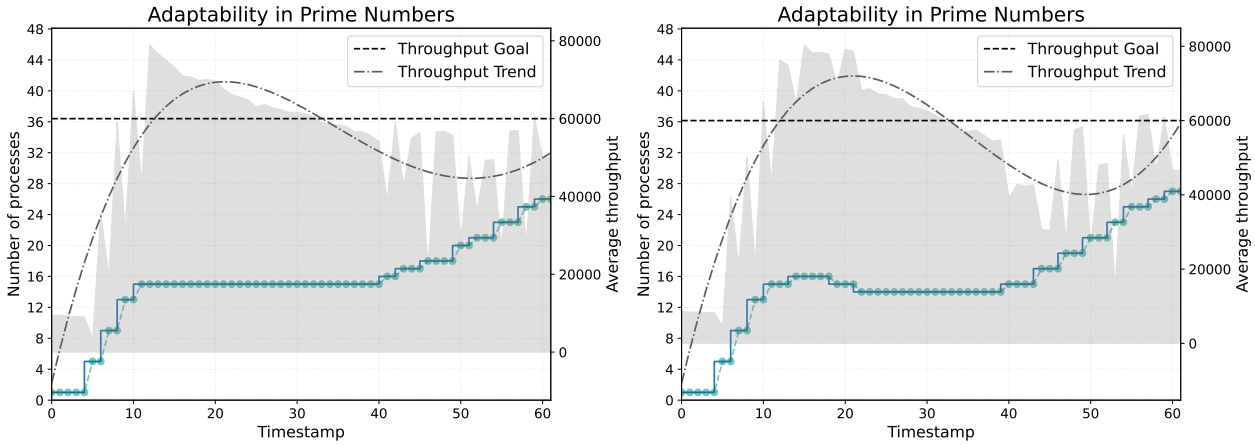
(b) Execution 2

Figure 7.2: Adaptive strategy characterization on Lane Detection.

7.3.3 Prime Numbers

The Prime Numbers results characterization are depicted in Figures 7.3. This streaming application has an unbalanced workload. Moreover, the computational intensity increasingly grows for each new stream item. The workload increase can be seen in the constant parts of the execution. For example, between timestamps 10 and 40 in Figure 7.3a and between timestamps 20 and 40 in Figure 7.3b. Both executions are similar since they attempt to follow the workload increase by adding new processes. Between timestamps 10 and 20, the executions differ in decisions. The first maintains the number of processes, while the second removes some processes. Each time processes are removed, we see a bigger decrease in throughput, even if just a single process is removed.

The gaps after timestamp 40 are due to Prime numbers' unbalanced workload. Some time periods that get more computationally intensive stream data items to process will produce fewer items. However, our self-adaptive algorithm accumulates all items produced in between the time intervals. Therefore, the peaks' interference is reduced during the monitoring.



(a) Execution 1

(b) Execution 2

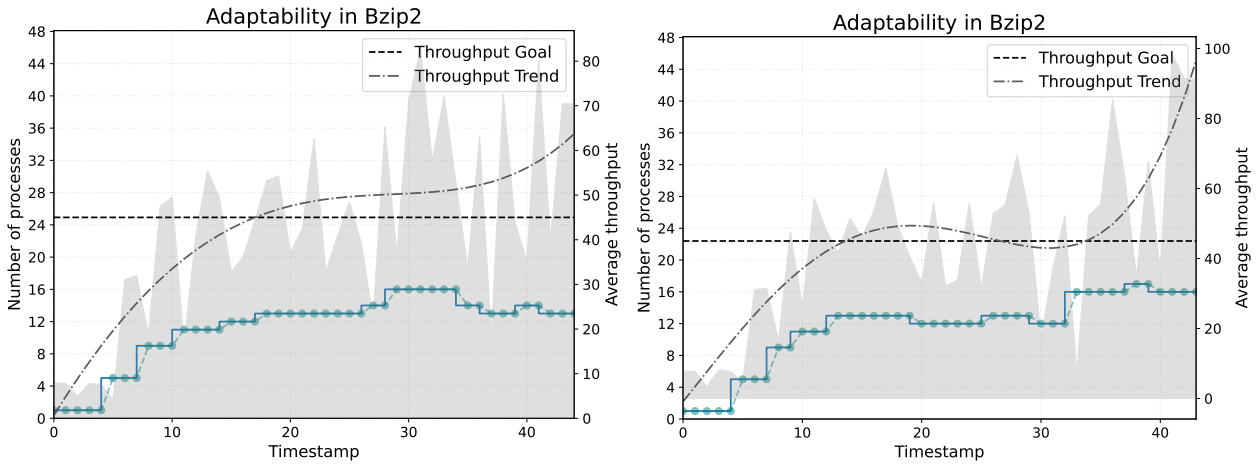
Figure 7.3: Adaptive strategy characterization on Prime Numbers.

7.3.4 Bzip2

The final application is Bzip2, and we showcase its two execution results in Figures 7.4. As can be seen, this application also has an unbalanced workload. There is a significant variation between the throughput MPR measures. However, the self-adaptive algorithm follows these variations by adding and removing processes. The two execution are similar until timestamp 20, except at the end when one execution adds another process while another removes it. The first execution, which adds a new process (Figure 7.4a), has a smoother transition to 16 processes around the timestamp 30. The second execution removes one process two times (Figure 7.4b) and suddenly detects it should add 4 processes at once to compensate for the throughput loss (after timestamp 30).

7.3.5 Adaptability Evaluation

In addition to the characterization graphs, we measured the time spent in the reconfiguration phases. This is shown in Figure 7.5. We split the measured time by separately measuring the time spent adding and banning processes. As can be seen, the time during initialization is minimal since we start the application with a single Compute



(a) Execution 1 (b) Execution 2

Figure 7.4: Adaptive strategy characterization on Bzip2.

process and scale up until achieving the desired throughput goal. In the characterizations graphs previously explained, we can observe many adaptation decisions were toward adding more processes. Still, the time spent in this phase is acceptable compared to the computation time. In Prime Numbers, it represented 10.38% while in Lane Detection, it is 7.38%. These two applications do not show expressive computational time when reconfiguring to remove processes. Lane Detection does not remove processes, while Prime numbers rarely do so (0.33% of execution time). Still, ban time is lower than others since, during our experiments, only small workload items (almost instant computation) were in the buffer when reconfiguration is needed. For other applications, ban time is significantly higher since stream items are more computationally intensive, and everyone waits for the buffers to be empty. This can be seen in Mandelbrot Set and Bzip2, where ban time is 12.46% and 2.96%, respectively.

7.4 Final Remarks

In this section, we reported the self-adaptive algorithm characterization results. Since we do not measure throughput for performance comparison, we configured the self-adaptive strategy to frequently change the number of processes with the goal of stressing MPR's self-adaptive capabilities. Ideally, distributed systems would use a less frequent adaptation rate to minimize reconfiguration overhead. For example, our experiments showed that in Mandelbrot Set there was a situation in which a second reconfiguration immediately succeeded a previous reconfiguration (stack of eight processes being added in a single timestamp). This means that the Pipeline did not compute any stream data item between two consecutive self-adaptations and only reconfigured.

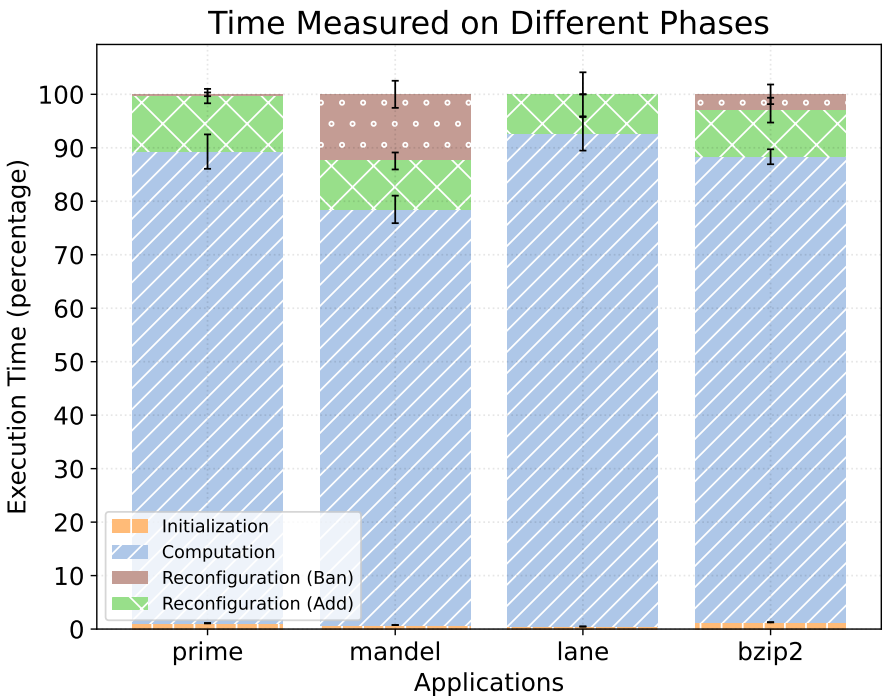


Figure 7.5: Evaluation on the execution time measured during adaptability.

In the experiments, we showed that our self-adaptive algorithm (based on a MAPE feedback loop) was able to cope with MPR and read the Pipeline application execution metrics (Monitoring stage) while applying the decision actions (Execution stage). Internally, the self-adaptive algorithm implements the analysis stage and adopts a static strategy to plan the adaptations (Analysis and Planning stages). The results showed that the self-adaptive algorithm was able to adjust the number of parallel processes without user intervention. Also, the execution time evaluation revealed that the overhead for constantly changing the number of processes represents an acceptable percentage of the application's total execution time. In all applications, the execution time spent reconfiguring to add processes is near 10%, while removing processes varied between 3% and 12%. However, the characterization results from the self-adaptive algorithm showed optimization opportunities. Our primary goal in conducting these experiments was not to implement an optimal self-adaptive algorithm but to test the self-adaptive capabilities of MPR. Going forward, future research could focus on developing a more sophisticated planning stage that incorporates machine learning techniques to accurately predict workload variation and anticipate adaptation decisions.

With these experiments, we concluded that MPR self-adaptive interface is flexible, expressive, and portable among different programming languages. Moreover, although our self-adaptive algorithm does not make optimal decisions, it was enough for us to conclude that MPR's runtime system supports self-adaptive distributed stream processing. In this thesis, we have laid the foundation for further research by developing a self-adaptive runtime system on top of MPI. MPR provides other researchers with the tools

necessary to implement their own self-adaptive algorithms. The runtime system is flexible to support other monitoring metrics beyond the number of items sent and received we implemented in this version. Also, MPR's runtime system implements a synchronization protocol that enables processes' to agree on a global Pipeline state. This opens the possibility of extending MPR via additional parameter configurations to optimize the Pipeline execution. For example, programmers can leverage the built-in synchronization to add new functionalities other than adding or removing processes on-the-fly.

8. CONCLUSION

In this work, we have tackled the issue of adaptability in distributed streaming systems and introduced MPR, a new framework aimed at simplifying the development of self-adaptive distributed stream processing applications. The framework incorporates higher-level programming abstractions targeting application programmers. The abstractions are dynamic process management, scheduling, data communication, serialization, load balancing, ordering, and back pressure. Thanks to these programming abstractions and an easy-to-use API, MPR becomes a viable solution for implementing distributed streaming systems in C++. Some features we included are not unique and are already tackled by solutions from our related work. Still, other features that MPR presents are novel to distributed stream processing in C++, namely dynamic process management, serialization abstractions, and back pressure. The last two features are available in MPR's API for application programmers, but more experiments will be conducted to conclude and consolidate these MPR features. Dynamic process management is rarely used by programmers to implement MPI programs, as revealed by recent surveys on MPI usability [50, 44]. In this work, we fully employ MPI's dynamic process creation, and in addition to that, we also introduced a new strategy to deal with dynamic Pipelines based on recent MPI specifications. The strategy was integrated with MPR's runtime system to support orchestrating and executing dynamic computations that may encounter variations during execution time. We have designed a novel runtime system that includes algorithms to handle dynamic process creation or removal, Pipeline job assignment, data management, and a leader-based synchronization protocol to coordinate MPI's processes.

To expose the self-adaptive capabilities, we have implemented a second API in MPR that allows self-adaptive algorithms to access Pipeline metrics and configure Pipeline execution parameters on-the-fly. Our focus was on making this interface portable, and we achieved this by using external Json files to exploit adaptability in MPR. The MPR runtime system can automatically reconfigure itself by reading configuration parameters from these files, and it can also report Pipeline execution metrics on them. The use of external Json files allows self-adaptive programs written in any programming language to communicate with MPR's runtime system. As a result of our work, other researchers and programmers can quickly test and evaluate novel self-adaptive algorithms, strategies, and optimizations using MPR. We believe that MPR's runtime system on top of MPI can be a valuable benchmark for comparing and testing new distributed optimizations and self-adaptive algorithms.

We conducted a performance evaluation on MPR parallelizations using stream processing applications from different domains. The performance experiment's goal was twofold, we aimed to compare MPR's performance with respect to DSParLib and handwritten MPI versions and evaluate two MPR features. The results demonstrated that MPR's

overhead is negligible in applications that are not communication-intensive. But the results also showed that MPR can introduce extra overhead when communication is intensive and the network bandwith becomes a bottleneck. Furthermore, the experiments evaluating MPR's features have revealed that they can impact the application's performance. The results indicate that self-adaptive algorithms can exploit these parameters by changing their values during execution time to improve the Pipeline execution performance or responsiveness.

In addition, we also conducted experiments to evaluate and characterize MPR's adaptability capabilities. We showed that MPR supports integration with feedback loop autonomic management systems. For that, we implemented a self-adaptive algorithm in Python. It decides the number of processes necessary for achieving the throughput goal specified by the application programmer. Our designed strategy uses a dynamic scaling factor based on how far the application throughput was from the user-specified goal. The characterization results revealed that MPR can effectively adapt the runtime system by adding or removing processes during execution time. Thanks to this work, others can use and extend MPR's runtime system to implement and test their own self-adaptive algorithms written in any programming language.

8.1 Future Work

This thesis revealed many interesting research directions and possibilities for future work. In addition, we plan to continue the work in this thesis by extending the experiments and optimizing MPR's runtime system. We will make MPR publicly available to enable other researchers to use it to develop stream processing applications and evaluate their own self-adaptive algorithms.

- Investigating fault-tolerance capabilities. While we have initiated a study on this topic, our focus has been on adaptability due to time constraints. Previous research works have explored fault-tolerance in MPI [41, 18, 51, 73, 72, 56]. However, these works mainly address the static aspects of MPI, and their proposed recovery solutions are not suitable for stream processing. As such, developing fault-tolerance solutions that target the dynamic runtime of MPR or other dynamic workloads is an open research challenge that requires further investigation.

- Extending MPR to support other parallel programming paradigms. Currently, MPR supports stream processing via the standard Pipeline parallel pattern. In addition, MPR could also support other programming paradigms, such as dataflow and data-parallel. For dataflow, MPR employs groups of processes that can be leveraged to communicate in an all-to-all fashion, enabling stages to communicate in a non-

consecutive fashion. For example, in a Pipeline, three stages will communicate in a *stage1 to stage2 to stage 3* fashion. By extending MPR for dataflow, arbitrary communications such as *stage1 to (stage 2 and stage 3)* should be allowed. To extend MPR for the data-parallel paradigm, supporting batches would enable some classes of applications to be implemented in a traditional batch-processing fashion.

- Improving the experimental evaluation. In this work, our experiments were conducted in a small cluster with limited network bandwidth, which was the infrastructure we had available to execute our tests. Unfortunately, many of our explanations targeted the experimental environment aspects rather than providing valuable insights about MPR's runtime system. To address this, we intend to acquire a new cluster through collaborations with the university, which will enable us to extend our experimental evaluation. Our plans involve repeating all the experiments in a second cluster and comparing the results with the reports presented in this thesis to gain a better understanding of the MPR runtime system. Furthermore, we plan to implement new streaming applications with different characteristics than the ones selected in this work. For instance, developing an application using a Pipeline with more than three stages to demonstrate how the MPR runtime system handles more complex applications. Lastly, we also plan to run experiments with an additional self-adaptive algorithm implemented by an expert in self-adaptability to compare with the one presented in this work.

- Extending MPR's adaptability interface to support a wider range of configurable parameters. In the current MPR version, we support changing the number of processes during execution time. However, there are other metrics within the runtime system that could be dynamically configured as well. For example, the buffer size and various time intervals are hard-coded in the current implementation. However, by simply adding a new interface in the configuration Json file, MPR can enable them to be read from the file and self-adapt the runtime system on-the-fly. In addition, other optimizations could be included in the runtime system, such as batches, which enable the application to trade off latency and throughput. By expanding the range of configurable parameters, MPR can become more flexible and expressive, which enables programmers to fine-tune the runtime system to their specific use cases and requirements.

- Extending MPR to enable expert programmers to exploit lower-level features. Currently, we provide an execution context variable that is received by input and sent as output for each message. This variable carries some information that is used to send the messages. A possible extension is adding some interface in this context variable to enable manual adjust some parameters, such as setting the target process or modifying the message header. By doing so, MPR's flexibility is improved

and programmers can implement new load-balancing mechanisms and replacement ordering algorithms.

- Implementing an additional data buffer to store output messages. Currently, MPR implements a data buffer to store incoming messages with the input stream data items. However, experiments revealed that MPR could benefit from an output data buffer in order to reduce idle time. By introducing an additional data buffer in the output connection, a Stage process can already process the stream items and keep their results buffered until a request message arrives.
- Optimizing MPR's reconfiguration mechanisms when removing processes. In the current version, MPR's configuration protocol requires that all processes empty their buffers to allow them to be effectively removed. However, some applications have intensive computational stages, which increases the reconfiguration overhead. Instead of processing the data, an optimization strategy could enable the stream data items stored in a process' internal buffer to be dispatched to another active process, reducing the overhead.

8.2 List of Published Papers

This section presents the published papers during the master's studies. A partial contribution of this thesis was published in a journal paper [55]. In addition, other papers were published during the master's studies. They all aim at higher-level programming abstractions for stream processing and were inspirations in developing MPR, a C++ self-adaptive distributed stream processing framework. Except [57], which is a paper that originated from a work as an undergraduate and was revised and published during the master's studies.

- Journals
  - Júnior Löff; Renato Barreto Hoffmann; Ricardo Pieper; Dalvan Griebler; Luiz Gustavo Fernandes. DSParLib: A C++ Template Library for Distributed Stream Parallelism, International Journal of Parallel Programming, vol. 50, Oct 2022, pp. 454–485. [55]
  - Júnior Löff; Renato Barreto Hoffmann; Dalvan Griebler; Luiz Gustavo Fernandes. Combining Stream with Data Parallelism Abstractions for Multi-Cores, Journal of Computer Languages, vol. 73, Dec 2022, pp. 1–18. [54]
  - Renato Barreto Hoffmann; Júnior Löff; Dalvan Griebler; Luiz Gustavo Fernandes. OpenMP as Runtime for Providing High-level Stream Parallelism on Multi-Cores, The Journal of Supercomputing, vol. 78, Apr 2022, pp. 7655–7676. [43]

- Júnior Löff; Dalvan Griebler; Gabriele Mencagli; Gabriell Araujo; Massimo Torquati; Marco Danelutto; Luiz Gustavo Fernandes. The NAS parallel benchmarks for evaluating C++ parallel programming frameworks on shared-memory architectures, Future Generation Computer Systems, vol. 125, Jul 2021, pp. 743–757. [57]
- Ricardo Pieper; Júnior Löff; Renato Barreto Hoffmann; Dalvan Griebler; Luiz Gustavo Fernandes. High-level and Efficient Structured Stream Parallelism for Rust on Multi-cores, Journal of Computer Languages, vol. 65, Aug 2021, pp. 1–14. [65]

• Conferences

- Dinei Rockenbach; Júnior Löff; Gabriell Araujo; Dalvan Griebler; Luiz Gustavo Fernandes. High-Level Stream and Data Parallelism in C++ for GPUs. In: International Conference on Programming Languages, 2022, pp. 41–49. [69]
- Júnior Löff; Renato Barreto Hoffmann; Dalvan Griebler; Luiz Gustavo Fernandes. High-Level Stream and Data Parallelism in C++ for Multi-Cores. In: International Conference on Programming Languages, 2021, pp. 41–48. [58]

REFERENCES

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

[2] Aldinucci, M.; Campa, S.; Danelutto, M.; Kilpatrick, P.; Torquati, M. “Targeting Distributed Systems in FastFlow”. In: International Conference on Parallel Processing, 2012, pp. 47–56.

[3] Aldinucci, M.; Danelutto, M.; Kilpatrick, P.; Torquati, M. “FastFlow: High-Level and Efficient Streaming on Multi-Core”, *Programming multi-core and many-core computing systems, parallel and distributed computing*, vol. 24, Jan 2017, pp. 261–280.

[4] Aliaga, J. I.; Castillo, M.; Iserte, S.; Martín-Álvarez, I.; Mayo, R. “A Survey on Malleability Solutions for High-Performance Distributed Computing”, *Applied Sciences*, vol. 12, May 2022, pp. 1–32.

[5] Andrade, H. C. M.; Gedik, B.; Turaga, D. S. “Fundamentals of Stream Processing: Application Design, Systems, and Analytics”. Cambridge University Press, 2014, 558p.

[6] Apache Software Foundation. “Apache Beam”. Source: https://beam.apache.org, Dec 2022.

[7] Apache Software Foundation. “Apache Flink”. Source: https://flink.apache.org, Dec 2022.

[8] Apache Software Foundation. “Apache Hadoop”. Source: https://hadoop.apache.org, Dec 2022.

[9] Apache Software Foundation. “Apache Heron”. Source: https://heron.apache.org, Dec 2022.

[10] Apache Software Foundation. “Apache Kafka”. Source: https://kafka.apache.org, Dec 2022.

[11] Apache Software Foundation. “Apache Samza”. Source: https://samza.apache.org, Dec 2022.

[12] Apache Software Foundation. “Apache Spark”. Source: https://spark.apache.org, Dec 2022.

[13] Apache Software Foundation. “Apache Storm”. Source: https://storm.apache.org, Dec 2022.

[14] Benoit, A.; Cole, M.; Gilmore, S.; Hillston, J. “Flexible Skeletal Programming with eSkel”. In: International Conference on Parallel Processing, 2005, pp. 761–770.

[15] Biddiscombe, J.; Bikineev, A.; Heller, T.; Kaiser, H. “Zero Copy Serialization using RMA in the HPX Distributed Task-based Runtime”. In: International Conference on WWW/Internet and Applied Computing, 2017, pp. 151–158.

[16] Bienia, C.; Kumar, S.; Singh, J. P.; Li, K. “The PARSEC Benchmark Suite: Characterization and Architectural Implications”. In: International Conference on Parallel Architectures and Compilation Techniques, 2008, pp. 72–81.

[17] Bingmann, T.; Axtmann, M.; Jöbstl, E.; Lamm, S.; Nguyen, H. C.; Noe, A.; Schlag, S.; Stumpp, M.; Sturm, T.; Sanders, P. “Thrill: High-performance Algorithmic Distributed Batch Data Processing with C++”. In: International Conference on Big Data, 2016, pp. 172–183.

[18] Bland, W.; Bouteiller, A.; Herault, T.; Bosilca, G.; Dongarra, J. “Post-failure Recovery of MPI Communication Capability: Design and Rationale”, *The International Journal of High Performance Computing Applications*, vol. 27, Aug 2013, pp. 244–254.

[19] Boost committee. “Boost C++ Library: Serialization”. Source: https://www.boost.org/doc/libs/1_79_0/libs/serialization/doc/index.html, Dec 2022.

[20] Carbone, P.; Katsifodimos, A.; Ewen, S.; Markl, V.; Haridi, S.; Tzoumas, K. “Apache Flink: Stream and Batch Processing in a Single Engine”, *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, Dec 2015, pp. 28–38.

[21] Cardellini, V.; Presti, F. L.; Nardelli, M.; Russo, G. R. “Decentralized Self-Adaptation for Elastic Data Stream Processing”, *Future Generation Computer Systems*, vol. 87, Oct 2018, pp. 171–185.

[22] Chandy, K. M.; Lamport, L. “Distributed Snapshots: Determining Global States of Distributed Systems”, *ACM Transactions on Computer Systems*, vol. 3, Feb 1985, pp. 63–75.

[23] Cheng, D.; Zhou, X.; Wang, Y.; Jiang, C. “Adaptive Scheduling Parallel Jobs with Dynamic Batching in Spark Streaming”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, Jun 2018, pp. 2672–2685.

[24] Ciechanowicz, P.; Kuchen, H. “Enhancing Muesli’s Data Parallel Skeletons for Multi-Core Computer Architectures”. In: International Conference on High Performance Computing and Communications, 2010, pp. 108–113.

[25] Cruz, G. M. “Optimization Techniques for Adaptability in MPI Applications”, Doctoral thesis, Computer Science and Engineering Department, Universidad Carlos III de Madrid, 2015, 187p.

[26] Dagum, L.; Menon, R. “OpenMP: an Industry Standard API for Shared-Memory Programming”, *IEEE computational science and engineering*, vol. 5, Mar 1998, pp. 46–55.

[27] Deng, S.; Wang, B.; Huang, S.; Yue, C.; Zhou, J.; Wang, G. “Self-Adaptive Framework for Efficient Stream Data Classification on Storm”, *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 50, Oct 2017, pp. 123–136.

[28] Elnozahy, E. N.; Alvisi, L.; Wang, Y.-M.; Johnson, D. B. “A Survey of Rollback-Recovery Protocols in Message-Passing Systems”, *ACM Computing Surveys*, vol. 34, Sep 2002, pp. 375–408.

[29] Eskandari, L.; Huang, Z.; Eyers, D. “P-Scheduler: Adaptive Hierarchical Scheduling in Apache Storm”. In: International Conference of the Australasian Computer Science Week Multiconference, 2016, pp. 1–10.

[30] Falcou, J.; Sérot, J.; Chateau, T.; Lapresté, J.-T. “Quaff: Efficient C++ Design for Parallel Skeletons”, *Parallel Computing*, vol. 32, Sep 2006, pp. 604–615.

[31] Floratou, A.; Agrawal, A.; Graham, B.; Rao, S.; Ramasamy, K. “Dhalion: Self-Regulating Stream Processing in Heron”, *Proceedings of the VLDB Endowment*, vol. 10, Aug 2017, pp. 1825–1836.

[32] Fu, T. Z.; Ding, J.; Ma, R. T.; Winslett, M.; Yang, Y.; Zhang, Z. “DRS: Auto-Scaling for Real-Time Stream Analytics”, *IEEE/ACM Transactions on Networking*, vol. 25, Sep 2017, pp. 3338–3352.

[33] Google Inc. “Protocol Buffers”. Source: https://github.com/protocolbuffers/protobuf, Dec 2022.

[34] Grant, W. S.; Voorhies, R. “Cereal - A C++11 library for serialization”. Source: https://github.com/USCiLab/cereal, Dec 2022.

[35] Griebler, D. “Domain-Specific Language & Support Tool for High-Level Stream Parallelism”, Doctoral thesis, School of Technology, Pontifical Catholic University of Rio Grande do Sul, 2016, 243p.

[36] Griebler, D.; Danelutto, M.; Torquati, M.; Fernandes, L. G. “An Embedded C++ Domain-Specific Language for Stream Parallelism”. In: International Conference on Parallel Computing, 2015, pp. 317–326.

[37] Griebler, D.; Danelutto, M.; Torquati, M.; Fernandes, L. G. “SPar: A DSL for High-Level and Productive Stream Parallelism”, *Parallel Processing Letters*, vol. 27, Mar 2017, pp. 1–20.

[38] Griebler, D.; Fernandes, L. G. “Towards Distributed Parallel Programming Support for the SPar DSL”. In: International Conference on Parallel Computing, 2017, pp. 563–572.

[39] Griebler, D.; Hoffmann, R. B.; Danelutto, M.; Fernandes, L. G. “High-Level and Productive Stream Parallelism for Dedup, Ferret, and Bzip2”, *International Journal of Parallel Programming*, vol. 47, Feb 2018, pp. 253–271.

[40] Griebler, D.; Hoffmann, R. B.; Danelutto, M.; Fernandes, L. G. “Stream Parallelism with Ordered Data Constraints on Multi-Core Systems”, *Journal of Supercomputing*, vol. 75, Jul 2018, pp. 4042–4061.

[41] Gropp, W.; Lusk, E. “Fault Tolerance in Message Passing Interface Programs”, *The International Journal of High Performance Computing Applications*, vol. 18, Aug 2004, pp. 363–372.

[42] Gropp, W.; Lusk, E.; Skjellum, A.; Lusk, A. D. F. E. E. “Using MPI: Portable Parallel Programming with the Message-Passing Interface”. MIT press, 1999, 395p.

[43] 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, Apr 2022, pp. 7655–7676.

[44] Hori, A.; Jeannot, E.; Bosilca, G.; Ogura, T.; Gerofi, B.; Yin, J.; Ishikawa, Y. “An International Survey on MPI Users”, *Parallel Computing*, vol. 108, Dec 2021, pp. 1–13.

[45] Kaiser, H.; Brodowicz, M.; Sterling, T. “Parallex an Advanced Parallel Execution Model for Scaling-Impaired Applications”. In: International Conference on Parallel Processing Workshops, 2009, pp. 394–401.

[46] Kaiser, H.; Diehl, P.; Lemoine, A. S.; Lelbach, B. A.; Amini, P.; Berge, A.; Biddiscombe, J.; Brandt, S. R.; Gupta, N.; Heller, T.; et al.. “HPX: The C++ Standard Library for Parallelism and Concurrency”, *Journal of Open Source Software*, vol. 5, Sep 2020, pp. 1–9.

[47] Kalavri, V.; Liagouris, J.; Hoffmann, M.; Dimitrova, D.; Forshaw, M.; Roscoe, T. “Three Steps is All You Need: Fast, Accurate, Automatic Scaling Decisions for Distributed Streaming Dataflows”. In: International Conference on Operating Systems Design and Implementation, 2018, pp. 783–798.

[48] Kanev, S.; Darago, J. P.; Hazelwood, K.; Ranganathan, P.; Moseley, T.; Wei, G.-Y.; Brooks, D. “Profiling a Warehouse-Scale Computer”. In: International Conference on Computer Architecture, 2015, pp. 158–169.

[49] Kulkarni, S.; Bhagat, N.; Fu, M.; Kedigehalli, V.; Kellogg, C.; Mittal, S.; Patel, J. M.; Ramasamy, K.; Taneja, S. “Twitter Heron: Stream Processing at Scale”. In: International Conference on Management of Data, 2015, pp. 239–250.

[50] 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: International Conference for High Performance Computing, Networking, Storage and Analysis, 2019, pp. 1–14.

[51] Laguna, I.; Richards, D. F.; Gamblin, T.; Schulz, M.; de Supinski, B. R.; Mohror, K.; Pritchard, H. “Evaluating and Extending User-Level Fault Tolerance in MPI Applications”, *The International Journal of High Performance Computing Applications*, vol. 30, Aug 2016, pp. 305–319.

[52] Li, H.; Fang, H.; Dai, H.; Zhou, T.; Shi, W.; Wang, J.; Xu, C. “A Cost-Efficient Scheduling Algorithm for Streaming Processing Applications on Cloud”, *Cluster Computing*, vol. 25, Apr 2021, pp. 781–803.

[53] Liu, X.; Buyya, R. “Resource Management and Scheduling in Distributed Stream Processing Systems: A Taxonomy, Review, and Future Directions”, *ACM Computing Surveys*, vol. 53, May 2020, pp. 1–41.

[54] 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, Dec 2022, pp. 1–18.

[55] 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, Oct 2022, pp. 454–485.

[56] Losada, N.; González, P.; Martín, M. J.; Bosilca, G.; Bouteiller, A.; Teranishi, K. “Fault Tolerance of MPI Applications in Exascale Systems: The ULFM Solution”, *Future Generation Computer Systems*, vol. 106, May 2020, pp. 467–481.

[57] Löff, J.; Griebler, D.; Mencagli, G.; Araujo, G.; Torquati, M.; Danelutto, M.; Fernandes, L. G. “The NAS parallel benchmarks for evaluating C++ parallel programming frameworks on shared-memory architectures”, *Future Generation Computer Systems*, vol. 125, Jul 2021, pp. 743–757.

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

[59] Mancini, E. P.; Marsh, G.; Panda, D. K. “An MPI-Stream Hybrid Programming Model for Computational Clusters”. In: International Conference on Cluster, Cloud and Grid Computing, 2010, pp. 323–330.

[60] McCool, M.; Robison, A.; Reinders, J. “Structured Parallel Programming: Patterns for Efficient Computation”. Elsevier, 2012, 406p.

[61] Mencagli, G.; Torquati, M.; Cardaci, A.; Fais, A.; Rinaldi, L.; Danelutto, M. “WindFlow: High-Speed Continuous Stream Processing With Parallel Building Blocks”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, Apr 2021, pp. 2748–2763.

[62] Message Passing Interface Forum. “MPI: A Message-Passing Interface Standard”. Source: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf, Dec 2022.

[63] Morisawa, Y.; Suzuki, M.; Kitahara, T. “Flexible Executor Allocation without Latency Increase for Stream Processing in Apache Spark”. In: International Conference on Big Data, 2020, pp. 2198–2206.

[64] Pieper, R. “High-level Programming Abstractions for Distributed Stream Processing”, Master’s thesis, School of Technology, Pontifical Catholic University of Rio Grande do Sul, 2020, 170p.

[65] 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, Aug 2021, pp. 1–14.

[66] Raghavan, D.; Levis, P.; Zaharia, M.; Zhang, I. “Breakfast of Champions: Towards Zero-Copy Serialization with NIC Scatter-Gather”. In: International Conference on Hot Topics in Operating Systems, 2021, pp. 199–205.

[67] Reinders, J. “Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism”. O’Reilly Media, Inc., 2007, 336p.

[68] Rivas-Gomez, S.; Gioiosa, R.; Peng, I. B.; Kestor, G.; Narasimhamurthy, S.; Laure, E.; Markidis, S. “MPI Windows on Storage for HPC Applications”, *Parallel Computing*, vol. 77, Sep 2018, pp. 38–56.

[69] Rockenbach, D. A.; Löff, J.; Araujo, G.; Griebler, D.; Fernandes, L. G. “High-Level Stream and Data Parallelism in C++ for GPUs”. In: International Conference on Programming Languages, 2022, pp. 41–49.

[70] Silvestre, P. M. F. “Clonos: Consistent High-Availability for Distributed Stream Processing through Causal Logging”, Master’s thesis, Faculty of Sciences and Technology, NOVA University of Lisbon, 2020, 130p.

[71] Stein, C. M.; Rockenbach, D. A.; Griebler, D.; Torquati, M.; Mencagli, G.; Danelutto, M.; Fernandes, L. G. “Latency-Aware Adaptive Micro-Batching Techniques for Streamed Data Compression on Graphics Processing Units”, *Concurrency and Computation: Practice and Experience*, vol. 33, Jun 2020, pp. 1–19.

[72] Sultana, N.; Rüfenacht, M.; Skjellum, A.; Laguna, I.; Mohror, K. “Failure Recovery for Bulk Synchronous Applications with MPI Stages”, *Parallel Computing*, vol. 84, May 2019, pp. 1–14.

[73] Sultana, N.; Skjellum, A.; Laguna, I.; Farmer, M. S.; Mohror, K.; Emani, M. “MPI Stages: Checkpointing MPI State for Bulk Synchronous Applications”. In: International Conference of MPI Users’ Group Meeting, 2018, pp. 1–11.

[74] Tonci, N.; Torquati, M.; Mencagli, G.; Danelutto, M. “Distributed-Memory FastFlow Building Blocks”, *International Journal of Parallel Programming*, vol. 51, Feb 2022, pp. 1–21.

[75] Vogel, A.; Griebler, D.; Danelutto, M.; Fernandes, L. G. “Self-Adaptation on Parallel Stream Processing: A Systematic Review”, *Concurrency and Computation: Practice and Experience*, vol. 34, Mar 2021, pp. 1–30.

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

[77] Vogel, A.; Rista, C.; Justo, G.; Ewald, E.; Griebler, D.; Mencagli, G.; Fernandes, L. G. “Parallel Stream Processing with MPI for Video Analytics and Data Visualization”. In: International Conference on High Performance Computing Systems, 2020, pp. 102–116.

[78] Wagner, A.; Rostoker, C. “A Lightweight Stream-Processing Library using MPI”. In: International Conference on Parallel & Distributed Processing, 2009, pp. 1–8.

[79] Weyns, D.; Schmerl, B.; Kishida, M.; Leva, A.; Litoiu, M.; Ozay, N.; Paterson, C.; Tei, K. “Towards Better Adaptive Systems by Combining MAPE, Control Theory, and Machine Learning”. In: International Conference on Software Engineering for Adaptive and Self-Managing Systems, 2021, pp. 217–223.

[80] Wolnikowski, A.; Ibanez, S.; Stone, J.; Kim, C.; Manohar, R.; Soulé, R. “Zerializer: Towards Zero-Copy Serialization”. In: International Conference on Hot Topics in Operating Systems, 2021, pp. 206–212.

[81] Xu, L.; Peng, B.; Gupta, I. “Stela: Enabling Stream Processing Systems to Scale-in and Scale-out On-demand”. In: International Conference on Cloud Engineering, 2016, pp. 22–31.

[82] Zeuch, S.; Monte, B. D.; Karimov, J.; Lutz, C.; Renz, M.; Traub, J.; Breß, S.; Rabl, T.; Markl, V. “Analyzing Efficient Stream Processing on Modern Hardware”, *Proceedings of the VLDB Endowment*, vol. 12, Jan 2019, pp. 516–530.

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