PUCRS

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

RICARDO LUIS PIEPER

HIGH-LEVEL PROGRAMMING ABSTRACTIONS FOR DISTRIBUTED STREAM PROCESSING

Porto Alegre
2020

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

HIGH-LEVEL PROGRAMMING ABSTRACTIONS FOR DISTRIBUTED STREAM PROCESSING

RICARDO LUIS PIEPER

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

Porto Alegre
2020

Ficha Catalográfica

P614h Pieper, Ricardo Luis

High-level programming abstractions for distributed stream processing / Ricardo Luis Pieper. – 2020.
171 f.
Dissertação (Mestrado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

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

1. Parallel Programming. 2. Stream Parallelism. 3. Parallel Code Generation. 4. Domain-Specific Language. 5. Skeleton Library. I. Fernandes, Luiz Gustavo Leão. 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

Ricardo Luis Pieper

High-Level Programming Abstractions for Distributed Stream Processing

This Master Thesis has been submitted in partial fulfillment of the requirements for the degree of Master of Computer Science, of the Graduate Program in Computer Science, School of Technology of the Pontifícia Universidade Católica do Rio Grande do Sul.

Sanctioned on 25th September, 2020.

COMMITTEE MEMBERS:

Prof. Dr. Marco Danelluto (University of Pisa)

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

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

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

ACKNOWLEDGMENTS

I would like to thank the people that made this work possible. First, I would like to thank my family, peers and colleagues for supporting me and providing feedback through this work. Second, I would like to thank Ubots (my day job) for supporting my decision to pursue a Master's degree. I would like to thank my advisor Luiz Gustavo for trusting me from the beginning, and supporting me through the projects we participated together. Last, but not least, I would like to thank my co-advisor Dalvan. Dalvan has infinite patience and infinite energy, and always pushes his students to make the best work possible. Thank you for the moral, technical and scientific support, and for the friendship we have been building for half a decade. This work was partially supported by funding from Hewlett Packard Brasil LTDA under Brazilian Informatics Law (Law nº 8.2.48 of 1991).

ABSTRAÇÕES DE PROGRAMAÇÃO DE ALTO NÍVEL PARA PROCESSAMENTO DE FLUXO CONTÍNUO DE DADOS DISTRIBUÍDOS

RESUMO

Aplicações de processamento de fluxo contínuo de dados (*stream processing*) representam uma parte significativa dos softwares atuais. Uma quantidade maior de dados é gerada todos os dias e de fontes variadas (através de dispositivos computacionais e aplicações), que requerem processamento em tempo hábil. Arquiteturas de memória compartilhada não conseguem atender demandas de processamento em larga escala. No contexto de computação de alta performance, a interface de programação MPI (*Message Passing Interface*) é considerada o estado da arte para implementar programas paralelos em C/C++. No entanto, a exploração de paralelismo de fluxo contínuo de dados usando MPI é difícil e sujeita a erros aos desenvolvedores de aplicação, pois ele expõe os detalhes de baixo nível das arquiteturas de computadores e sistemas operacionais. Os programadores precisam lidar com mecanismos de serialização de dados, comunicação entre processos, sincronização, tolerância a falhas, balanceamento de carga, escalonamento de trabalhos e estratégias de paralelismo. Este trabalho aborda um subconjunto desses problemas e desafios, oferecendo duas abstrações de alto-nível para processamento de fluxo contínuo de dados em arquiteturas de memória distribuída. Primeiramente, foi criada uma biblioteca de paralelismo de fluxo contínuo de dados nomeada de DSPARLIB. A biblioteca foi construída como uma *biblioteca de esqueletos* equipada com os padrões paralelos *Farm* e *Pipeline*, provendo abstrações acima do MPI. Depois, a linguagem SPAR foi estendida para suportar arquiteturas de memória distribuída, uma vez que ela é uma linguagem de domínio específico para expressar paralelismo de fluxo contínuo de dados usando anotações do C++11, que se mostram produtivas em arquiteturas de memória compartilhada. Este trabalho conseguiu estender o compilador e a linguagem SPAR sem alterar significativamente a sintaxe e a semântica, gerando código que usa DSPARLIB como ambiente de execução paralelo.

Os experimentos foram conduzidos usando aplicações reais de processamento de fluxo contínuo de dados em diferentes configurações de *cluster*. Este trabalho demonstra que a DSPARLIB provê uma interface mais simples que o MPI e apresenta um desempenho competitivo. O compilador da linguagem SPAR foi capaz de gerar código paralelo automaticamente, sem perdas de desempenho em comparação com código escrito à mão com a DSPARLIB. Por último, com todas essas abstrações de alto nível, a SPAR se torna a primeira linguagem baseada em anotações para expressar paralelismo de fluxo contínuo de dados em C++ a suportar arquiteturas de memória distribuída, evitando refatorações de código significativas para permitir a execução paralela em *clusters*.

Palavras-Chave: Programação Paralela, Paralelismo de Fluxo, Geração de Código Paralelo, Linguagem de Domínio Específico, Bibliotecas de Esqueleto, Esqueletos Algorítmicos, Padrões Paralelos, Processamento Paralelo, Arquiteturas de Memória Distribuída, Programação Distribuída, C++, Pipeline, Farm.

HIGH-LEVEL PROGRAMMING ABSTRACTIONS FOR DISTRIBUTED STREAM PROCESSING

ABSTRACT

Stream processing applications represent a significant part of today's software. An increased amount of streaming data is generated every day from various sources (computing devices and applications), which requires to be processed on time. Shared-memory architectures cannot cope with these large-scale processing demands. In High-Performance Computing (HPC), Message Passing Interface (MPI) is the state-of-the-art parallel API (Application Programming Interface) for implementing parallel C/C++ programs. However, the stream parallelism exploitation using MPI is difficult and error-prone to application developers because it exposes low-level details to them, regarding computer architectures and operating systems. Programmers have to deal with implementation mechanisms for data serialization, process communication and synchronization, fault tolerance, work scheduling, load balancing, and parallelism strategies. Our research work addresses a subset of these challenges and problems providing two high-level programming abstractions for distributed stream processing. First, we created a distributed stream parallelism library called DSPARLIB. It was built as a skeleton library equipped with Farm and Pipeline parallel patterns to provide programming abstractions on top of MPI. Second, we extend the SPAR language and compiler roles to support distributed memory architectures since it is a Domain-Specific Language (DSL) for expressing stream parallelism using C++11 annotation that has been proved to be productive on shared-memory architectures. We managed to make it work without significantly changing the easy of use language syntax and semantics, generating automatic parallel code with SPAR's compiler using DSPARLIB as the parallel runtime. The experiments were conducted using real-world stream processing applications and testing different cluster configurations. We demonstrated that DSPARLIB provides a simpler API than MPI and a competitive performance. Also, the SPAR's compiler was able to generate

parallel code automatically without performance penalties compared to handwritten codes in DSParLib. Finally, with all these high-level programming abstractions implemented, SPar becomes the first annotation-based language for expressing stream parallelism in C++ programs to support distributed-memory architectures, avoiding significant sequential code refactoring to enable parallel execution on clusters.

Keywords: Parallel Programming, Stream Parallelism, Parallel Code Generation, Domain-Specific Language, Skeleton Library, Algorithmic Skeletons, Parallel Patterns, Parallel Processing, Distributed Memory Architectures, Distributed Programming, C++, Pipeline, Farm.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

IOT – *Internet Of Things*

TPL – *Microsoft Task Parallelism Library*

API – *Application Programming Interface*

DSL – *Domain Specific Language*

HPC – *High-Performance Computing*

MPI – *Message Passing Interface*

TBB – *Thread Building Blocks*

CINCLE – *Compiler Infrastructure for New C/C++ Language Extensions*

SPAR – *Stream Parallelism*

LINQ – Language-Integrated Query

GPU – Graphics Processing Unit

TPU – Tensor Processing Unit

DAG – *Directed Acyclic Graph*

JVM – *Java Virtual Machine*

CPU – *Central Processing Unit*

SQL – Structured Query Language

TCP – Transmission Control Protocol

POJO – *Plain Old Java Object*

RPC – Remote Procedure Call

RDD – *Resilient Distributed Dataset*

HDFS – *Hadoop Distributed File System*

RAM – Random Access Memory

MPI – *Message Passing Interface*

HPX – High Performance ParalleX

LCO – Local Control Objects

AGAS – Active Global Address Space

ULFM – User-Level Fault Mitigation

GC – Garbage Colletor

REPARA – Reengineering and Enabling Performance And poweR of Applications

CUDA – *Compute Unified Device Architecture*

GRPPI – *Generic Reusable Parallel Pattern Interface*

PICO – Pipeline Composition

GAM – Global Asynchronous Memory

DIA – Distributed Immutable Array

POD – Plain Old Datatype

OS – Operating System

XML – Extensible Markup Language

POSIX – Portable Operating System Interface

EDM – eSkel Data Model

MIT – Massachusetts Institute of Technology

FIFO – *First In First Out*

CQL – Continuous Query Language

SPL – Streams Processing Language

SFINAE – Substitution Failure Is Not An Error

STL – Standard Template Library

RTTI – Run-Time Type Information

GCC – GNU Compiler Collection

MD5 – Message Digest 5

PBZIP2 – Parallel Bzip2

FPS – Frames Per Second

MP4 – MPEG-4 AVC

MPEG – Moving Picture Experts Group

AVC – Advanced Video Coding

SPMD – Single Program Multiple Data

NFS – Network File System

RGB – Red, Green, Blue

HSV – Hue, Saturation, Value

ELF – Extensible Linking Format

VLC – VideoLAN Client

FPGAS – Field-Programmable Gate Array

CONTENTS

1. INTRODUCTION

The hardware industry has made significant progress in the last decades. From big, expensive, and slow machines of the 1980s, to cheaper, smaller, and faster machines of 2020 [TS06]. In previous decades, processor clock speeds increased at a steady pace, and single-threaded programs would become faster with each new processor technology advancement. Increasing frequencies also increases energy consumption and heat production [MRR12], which limits the improvement of single-core execution speed. Instead of increasing clock frequencies, the industry has shifted to multi-core processors. In 2020, high-end 64-core 128-thread processors have become available[1], whereas earlier processor releases have been focused in 4, 8, or 16 cores.

Programmers need to use parallel programming techniques to extract the full potential of a multicore processor. There have been initiatives for parallelizing code automatically, but they have not been successful [MRR12, MSM04]. Therefore, the programmer must explicitly use parallelism techniques in the code. However, the main programming paradigm is still sequential programming rather than parallel programming. Sequential code cannot use more than one processor core, which limits the performance scalability of the code. If new processors with more cores become available, sequential code cannot use the full potential of the new processor. In general, programmers can no longer rely on single-core execution to benefit from continued hardware improvements.

Given that new processor technologies are focused on multicore, it is important to write efficient parallel code. The challenges of writing parallel code are well documented in the literature [MRR12, Col89, Gri16, ADKT17, MSM04]. Parallel programming is often error-prone and can lead to poor performance if not done properly. For instance, modifying shared state must be done in a synchronized manner, but the synchronization itself might incur significant overheads or poor scaling. Fortunately, this problem has been studied for a long time. Nowadays, programming languages have built-in abstractions for parallelism exploitation such as the Stream API available in Java 8 [Ora20]. For Microsoft .NET, C# has language-integrated queries (LINQ) that can exploit parallelism [Mic20]. Other languages such as Rust and C++ have libraries that implement parallel abstractions such as Rayon [Ray20] or TBB [Rei07]. Application programmers nowadays have a wide range of options for parallelism exploitation.

Programmers looking for ways to exploit parallelism in their languages might find concepts like *map* and *reduce*. These abstractions are called *parallel patterns* and come from concepts that have been studied for decades. Algorithmic skeletons [Col89] became the theoretical framework for other studies in the area of parallelism abstractions. [MSM04] and [MRR12] give a practical introduction to parallel design patterns and structured parallel

[1] https://www.amd.com/en/products/cpu/amd-ryzen-threadripper-3990x

programming, which are based on algorithmic skeletons. In the context of stream processing, FastFlow [ADKT17], GrPPI [dRADFG17], and Thread Building Blocks [Rei07] allow the programmer to implement *structured stream parallelism* using the well known *pipeline* and *farm* patterns. Most of these libraries for C++ only support shared-memory architectures and cannot use other resources in an HPC cluster. Furthermore, the programmer might have to introduce significant code refactoring to existing code bases to exploit stream parallelism.

Moreover, distributed stream processing applications are becoming commonplace to further scale computation. In general, these applications process a possibly infinite, continuous, live stream of data. The data might be audio, video, sensor data, social media posts, stocks, and other types. *Big Data* frameworks play a big role in the field of distributed stream processing. Some of them are Apache Flink, Storm, Spark, Akka Streams, and others. The vast majority of such frameworks run in the Java Virtual Machine. There is space for C++ to become an important language in this field. The works in PiCo [Mis17] and Thrill [BAJ+16] describe ways in which C++ could become a mainstream *Big Data* language due to its expressiveness (with features brought by C++11 and C++14) and performance.

Unfortunately, distributed computing is not trivial due to well-known challenges such as network communication, reliability, and serialization [AGT14, TS06]. Specifically, the programmer must be aware of the costs of sending data back and forth through the network. In addition to that, data must be encoded (serialized) in a way that can be sent to other nodes in the network. The serialization process is complex but practical in languages such as Java, Python, C#, and others because their dynamic runtimes keep metadata about classes and types available during execution. Even in these languages, serialization is an important topic due to the potential of becoming a bottleneck [AGT14], and existing *Big data* frameworks provide multiple ways to configure and customize serialization [Lig19, CKE+15, Apa19, ZCF+10]. In C and C++, serialization is significantly more difficult since there is no metadata about types available during runtime.

Therefore, our work is motivated by the following problems:

- Programmability abstractions: Refactoring effort to introduce parallelism in existing C++ codebases or the necessity of rewriting existing code in different languages;
- Distributed-memory parallelism exploitation: Challenges inherent to distributed programming such as serialization, network communication, reliability, and performance.

The first problem is being addressed by SPAR [GDTF17] for shared-memory architectures. SPAR is a DSL (domain-specific language) compatible with C++11, which allows programmers to take existing C++ code and parallelize it using simple C++ annotations. Application programmers only need to learn the annotations and how to use them to parallelize existing code. SPAR uses the annotations to generate code that uses FastFlow for parallelism. This method provides increased productivity with negligible performance

losses [GHDF18a]. **The question is, can we also provide the same advantages regarding programming abstractions and performance for distributed-memory architectures in SPAR?**

Although SPAR supports multicore code generation to FastFlow, the original thesis could not make it work for the FastFlow's API (Application Programming Interface) that is for distributed-memory architectures [Gri16]. As MPI is ubiquitous in HPC, [GF17] studied whether SPAR can be integrated with MPI. However, no automatic code generation for cluster has been implemented. The prior work also highlighted the need for a pattern-based library for implementing stream processing without exposing the MPI low-level routines.

The long-term vision for SPAR is that it should be able to express stream parallelism with increased productivity and higher-level abstractions while scaling to any problem size. Support for distributed-memory architectures is necessary to achieve this vision. In Figure 1.1, we show the layered research framework proposed by [Gri16]. It is possible to see that multicore exploitation is done by generating FastFlow code using SPAR compiler, which was created by using CINCLE (Compiler Infrastructure for New C/C++ Languages Extensions). In this work, we developed DSPARLIB and extended SPAR to support clusters. GMaVis is a high-level description language for geospatial visualization in the application layer. It supports parallelism with SPAR for performance-critical sections of the code [LGMF17, LGMF15].

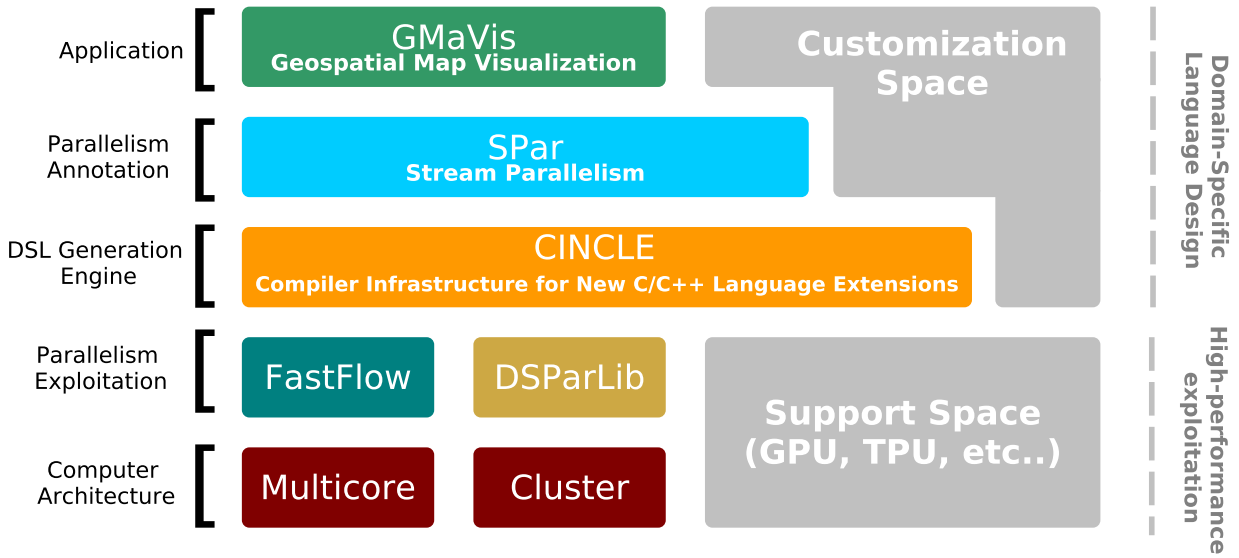

Figure 1.1 – Global research framework. Adapted from [Gri16].

The SPAR programming model has been proven generic enough to generate parallel code using different runtime libraries such as TBB [HGDF20] and OpenMP[Hof20], support self-adaptive parallelism degree abstractions [VGF21], and provide service level

objective via C++-11 attributes [GSV+18, GVS+19]. Also, the abstractions of SPAR were extended for supporting heterogeneous CPU and GPU computing along with a new structured parallel API called GSParLib [Roc20]. Similar extension were performed by [Lö20] for combining stream and data parallelism on multicores. Therefore, there are opportunities to improve the support space of SPAR for more computer architectures and runtime libraries.

Our goal is to design efficient and high-level programming abstractions for distributed stream parallelism. Therefore, the following scientific contributions were achieved:

- A new structured stream parallelism API (called DSPARLIB) for parallel and distributed stream processing, including efficient data serialization, scheduling, synchronization, and network communication.
- An extension to SPAR language to support parallel and distributed stream processing without changing the original semantics.
- New source-to-source code generation algorithms that avoid the redesign of stream pattern-based transformation rules in SPAR.

This document is organized as follows: Chapter 2 introduces the main concepts of stream processing and stream parallelism. We also explore tools and libraries for structured stream parallelism and parallel patterns. Chapter 4 contains the related work, in which we look into stream processing DSLs and recent developments in big data streaming for C++. Chapter 3 introduces the SPAR language, how to use it, and how the transformation rules work. Chapter 5 introduces DSPARLIB, our distributed streaming patterns library, where we present design and performance discussions. Chapter 6 introduces the changes to SPAR to support distributed streaming with DSPARLIB. Finally, Chapter 7 presents the conclusion and future works.

2. STREAM PROCESSING

Stream processing has become more important as the demand for data processing grows. Over the last few years, several companies developed tools for distributed, scalable stream processing, and *Big Data* tools are now widely available. Furthermore, research in the area of algorithmic skeletons explored parallel patterns for streaming applications.

In this chapter, we explore stream processing, stream parallelism patterns, and Big Data frameworks. Section 2.1 to 2.6 explore basic concepts found in stream processing such as operators, windowing, and back-pressure. Section 2.7 explores streaming in the field of algorithmic skeletons and parallel programming patterns. Section 2.8 explores frameworks for distributed streaming in the context of *Big Data*. Section 2.9 covers libraries for distributed programming in C++. Finally, 2.10 summarizes this chapter.

2.1 Streams

Stream Processing is a computing model to perform real-time processing in live data streams [AGT14]. Data sources often produce data on the scale of millions of data items per day. Streaming systems must be able to handle streams with varied throughput due to spikes or slowdowns. Streams also may vary in format and content, requiring suitable pre-processing steps. Streaming systems must process data quickly to provide valuable information in a timely manner.

A data stream is a potentially infinite sequence of tuples that share a schema, where each tuple is an atomic data item similar to a database record [AGT14]. Examples of sources of data streams are videos, audio, tweets from Twitter, events from sensors, or even a bounded stream of lines from a file or records from a database. A stream processing application may output data to one or more *data sinks*, which can be file systems, databases, message queues, and others. Figure 2.1 provides an abstract representation of a stream.

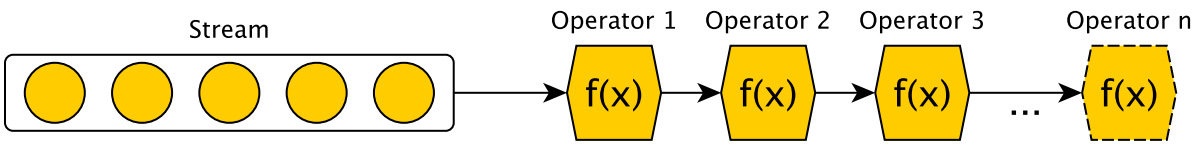

Figure 2.1 – Abstract stream. Adapted from [Gri16].

Therefore, a data stream is a channel continually transmitting data items or data to be processed by a sequence of operators.

2.2 Stream Operators

To manipulate a data stream, the programmer may define *operators*, which are the functions applied to every item of the stream. Operators are used to adapt data items to a more appropriate format (for instance, parsing the data), perform a computation, aggregation (reducing a stream to a reduced form, for instance, summing a stream of numbers), splitting (usually for parallelism), merging (combining multiple streams into one), and many others [AGT14]. Figure 2.2 offers an overview of operators working together.

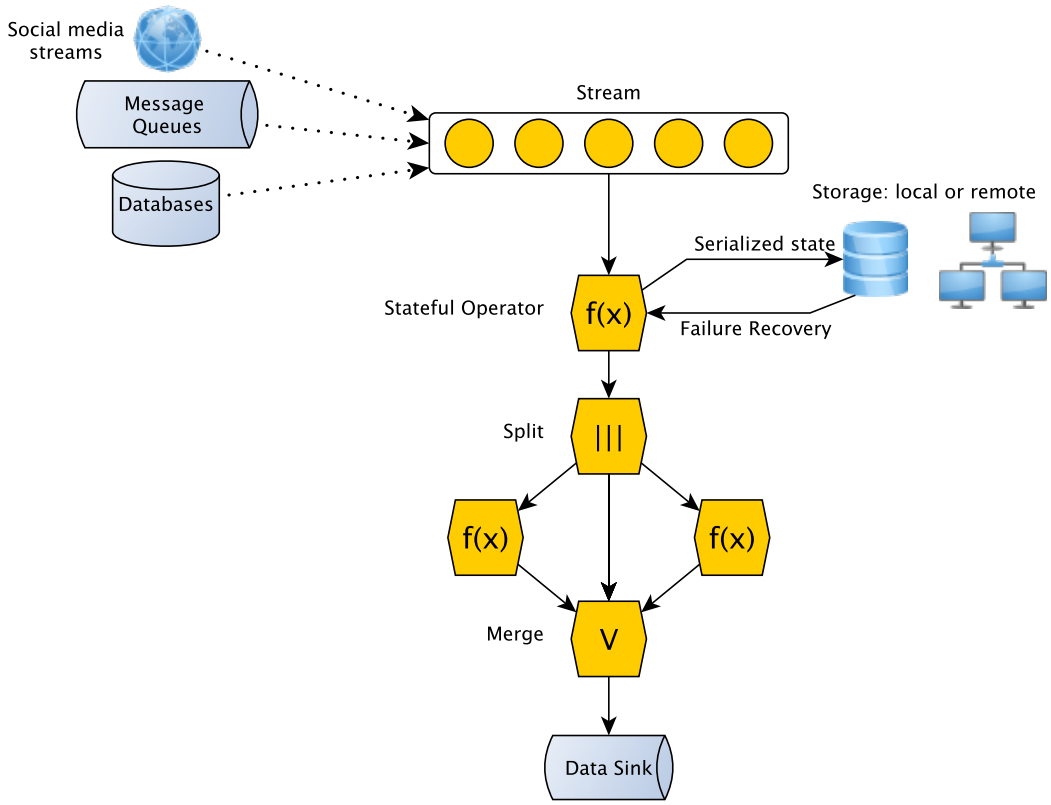

Figure 2.2 – Stream operators overview.

As will be presented in Section 2.8, many Big Data frameworks implement stream processing concepts. [Mis17] classifies their APIs as *declarative* or *topological*. Declarative APIs can be informally described as APIs that expose stream operators such as *Map*, *Reduce*, and other functions and methods commonly found in functional programming languages. Topological APIs allow the programmer to create topologies composed of graph nodes forming a DAG (directed acyclic graph), which are components of a topological API. The work of [Mis17] creates a formal definition of declarative APIs.

Operators that keep state are called *stateful operators*. These operators, different from stateless operators, need a more elaborate implementation in the runtime. For instance, a de-duplication operator needs to keep a list of previous stream items. When

a stream item arrives, it is checked against the list and removed if it is a duplicated data [AGT14].

However, a stream operator might run in a multithreaded context, potentially causing data races during concurrent reads and writes. In these cases, stateful operators must use synchronization mechanisms. [MRR12] explains that, for parallelization of aggregation operations, their functions must be associative. Sums and multiplications are examples of associative operations, but there are caveats regarding integer overflow and floating-point arithmetic. Tools like TBB (Thread Building Blocks) allow the user to define a stateful Farm stage, where manual synchronization is necessary.

Stateful operators also further complicate application recovery when a failure happens in a distributed context. If a cluster node holds application state for a stateful operator, it must be reconstructed when the application node restarts [AGT14]. This requires the runtime to persist the state outside of the application by replication, saving a snapshot or checkpoint, transactional logs, or a combination of these techniques [NPP+17, CKE+15].

Stateless operators simplify execution and runtime implementation. They do not need to save any kind of state, thus their recovery is trivial. Furthermore, they can run in parallel and do not need any kind of synchronization.

2.3 Windowing

Some operators need a bounded stream to operate, for instance, a sum of a stream of numbers. It is infeasible (due to the continuous nature of many streams) or impractical (due to the volume of data) to perform an aggregation to the whole stream [Mis17, AGT14, CKE+15]. Windowing refers to a strategy of buffering data items to apply functions over it. Windows enable extracting aggregate information over a finite list of items.

Therefore, windows have a triggering policy and an eviction policy. The triggering policy dictates when a window is generated or emitted from a window operator. The eviction policy dictates which elements can be held on the stream. For instance, the eviction policy might limit the buffering by a length of time (collect all items during 5 seconds) or by a count of items (create a window for every 100 items) [ZCF+10, CKE+15, AGT14]. Therefore, common eviction policies are count-based or time-based, however, the policy can have any implementation. Figure 2.3 shows an example of a sliding window with size 4 and step 1.

There are different types of windows. Some windows are fixed in size and do not overlap, often called as tumbling window. Another common type is the sliding window, where windows are fixed in size but overlapping is allowed, for instance: a list of events generated in the last 10 minutes, where one window is emitted every 2 minutes. In this example, there

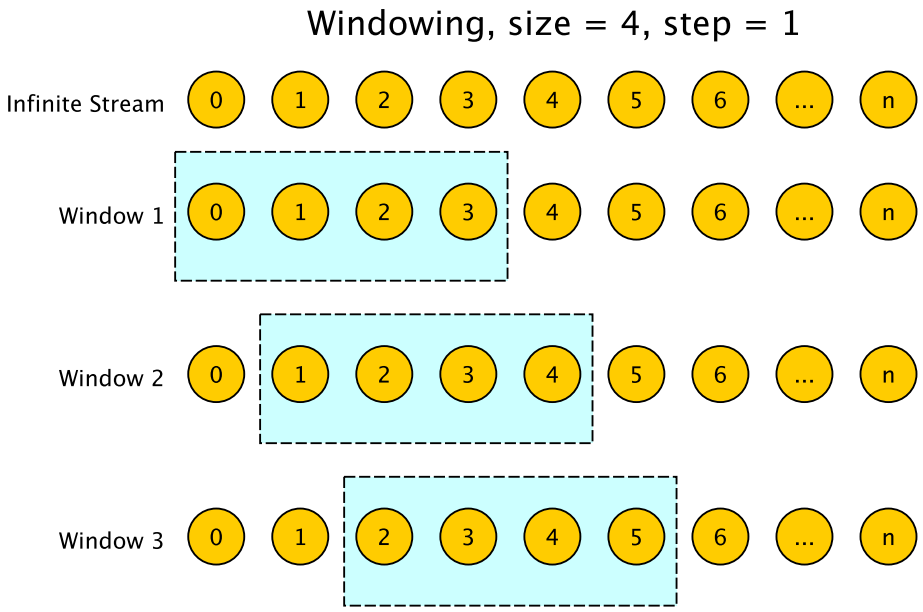

Figure 2.3 – Stream windowing.

is 8 minute per window emitted in sequence. In general, windowing can have a multitude of parameters to define the boundaries of a window [AGT14].

2.4 Fault tolerance

In a distributed stream processing context, it is necessary to consider how to handle failures. Fault-tolerance is the ability to deal with failures of transient nature, which can happen when: a node in the distributed system fails, a message fails to be delivered, or any kind of error that may stop the stream computing process. According to [AGT14], this does not include errors in the application logic, since this will require a change of the code.

Usually, this aspect is discussed in terms of delivery guarantees such as at-most-once (or best effort), at-least-once, and exactly-once [TS06, AGT14]. The meaning of each term varies depending on which tool is being considered. It is also discussed in terms of the guarantees of the data source itself: if a data source cannot be “rewound” or cannot ensure delivery, the guarantees are often reduced to at-least-once or at-most-once.

There are tools that also define "exactly-once" in terms of the effect each item has in the final state of the computation: the final state reflects the processing of each item exactly once by having idempotency (even if the underlying communication provides only at-least-once guarantees). Apache Kafka is frequently used as a data source that can provide exactly-once guarantees in streaming applications [KNR11] as we will describe in Section 2.8.8.

[AGT14] explains that not all operators are equally critical in a distributed stream processing application. Therefore, some operators might require less strict guarantees. It may be acceptable for some types of applications to lose a small number of events or fail sometimes, as long as the failures are kept to a minimum.

Fault tolerance can be implemented at application or runtime levels. The runtime might rely on monitoring of each stage to check if the failure is fatal and the application cannot continue, or if the operator can simply be restarted from a previous state. The runtime must also deal with delivery and processing guarantees. Application-level fault tolerance deals with checkpointing or other mechanisms that save the state of an operator, which sometimes require the programmer to provide functions to guide the runtime during checkpointing or state bootstrapping [AGT14].

2.5 Serialization

In a distributed stream processing scenario, the framework needs to serialize and deserialize data going through the network. When running a stream processing program in a single process, serialization is not necessary since they share the same memory space. The serialization process might be computationally expensive [AGT14]. Moreover, checkpointing requires serialization to save the operator state.

Automatic serialization is normally provided by frameworks running in a dynamic environment, such as most JVM-based frameworks [Lig19, CKE+15, Apa19]. This is possible because Java maintains class meta-data in memory during program execution. This meta-data can be queried using the Reflection API.

In contrast, C++ libraries such as Thrill[BAJ+16] and FastFlow[ADKT17] offer limited serialization capabilities due to the nature of C++. Thrill can automatically serialize data when it is contiguous in memory, using C++ templates. FastFlow allows zero-copy serialization and deserialization by offering specific methods in which the programmer can perform serialization and deserialization using their preferred method.

It is important, therefore, to choose a good format and serialization method that is fast enough and results in a small serialization overhead. Also, this term should not be confused with serializability, which is a property of execution schedules that lead to the same state as if the operations were applied sequentially.

2.6 Back-pressure

When an operator in the stream slows down, the runtime should prevent the application from failing due to upstream data sources producing more data than the system can handle. When a runtime has a back-pressure mechanism, it usually refers to the runtime ability to deal with producers and consumers operating at different speeds [KBF+15, Lig19]. When a source of data pushes more data downstream than the system can consume, the program becomes at risk of collapsing due to resource exhaustion (often memory or network buffers). Back-pressure attempts to slow down producers upstream (closer to the data sources) so that they do not overflow the stream with data items that will take too long to consume. Each implementation has its way of dealing with back-pressure.

When back-pressure happens, the slowest operator in the dataflow graph severely limits the throughput [AGT14]. To detect back-pressure, the stream must be constantly monitored. Some useful metrics are latency (the lower, the better), CPU utilization, and throughput. Frameworks such as Apache Flink can detect which operator in the stream dataflow graph is causing back-pressure [CKE+15].

To mitigate back-pressure, the programmer can optimize the operator or change its type. Operators of type split and merge help with load balancing. Meanwhile, since stateless operators have no state, they can run in parallel [AGT14, Lig19, Akk19].

2.7 Structured Stream Parallelism

The concept of algorithmic skeletons encompasses several parallel programming patterns that separate concerns between application programmers and system programmers [MRR12, Col89, MSM04]. Tools such as Thread Building Blocks (TBB) [Rei07] and FastFlow [ADKT17] implement several skeletons, including Map, Reduce, and the streaming Pipeline and Farm patterns. The field of structured stream parallelism studies patterns in which a stream computation can run in parallel using algorithmic skeletons.

A Pipeline is a common pattern in stream parallelism. A Pipeline separate stages of a computation into independent parts that can run in parallel, which is similar to the way a dataflow is composed of operators potentially running in parallel. The maximum throughput of a Pipeline is directly related to the number of stages, and do not benefit from extra processors [MRR12]. For instance, the maximum speedup of a Pipeline with n stages is n. However, the maximum throughput is also limited by the slowest stage, since all items must pass through all the stages.

Figure 2.4 shows a representation of a Pipeline pattern. The process starts by sending data to the first stage, or producing the data in the first stage. Data will flow through

the stages, where each stage can perform computation independently from other stages. The data emitted by each stage might be a different type than what was received.

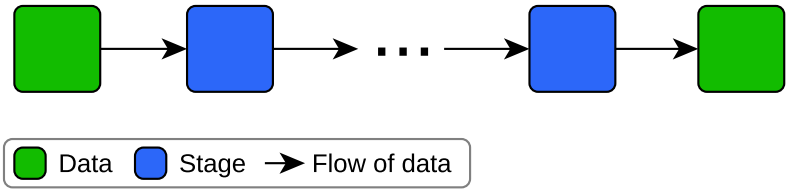

Figure 2.4 – Pipeline representation. Adapted from [MRR12].

To scale a Pipeline pattern implementation, stages can be replicated if they are stateless [MRR12]. This way, a Pipeline stage can benefit from extra processors. Each stage in a Pipeline can either be sequential in-order, sequential out-of-order, or parallel. A sequential stage is a stage that has only one Worker, while a parallel stage may have many Workers. A Pipeline with parallel stages is also called a *Farm* [Gri16, AD99].

Figure 2.5 illustrates the Farm pattern. Similar to Pipelines, there is data flowing through all components. Farms can be represented with the expression *farm*(*e*, *w*, *c*) [Gri16], where *e* is the Emitter, *w* is the Worker, and *c* is the Collector. Workers are stateless, and therefore can be replicated and execute in parallel.

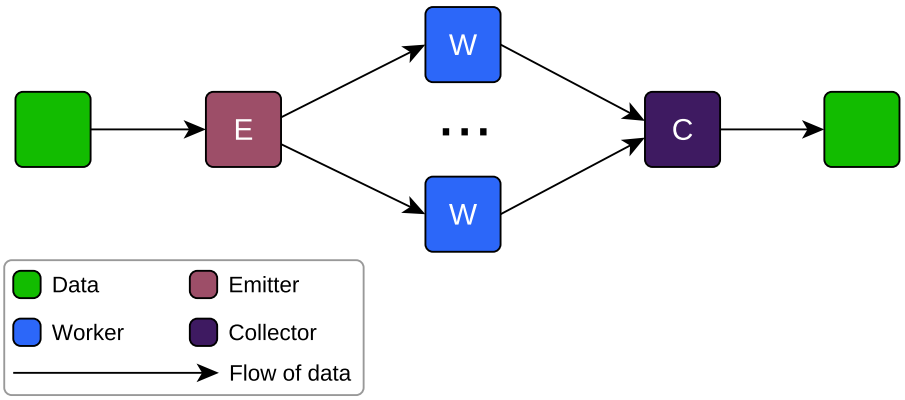

Figure 2.5 – Farm representation. Adapted from Pipeline example of [MRR12].

In general, a Pipeline stage has a queue of items to be processed [ADKT17], which is common in stream processing for shared-memory architectures. In the case of a Farm, each Worker may have a queue of items to be processed, or all Workers may share a common queue. Items are then distributed to queues using a round-robin or other load balancing methods, or all items are pushed to the queue of the stage where Workers perform polling [DMM17]. FastFlow [ADKT17] uses unbounded lock-free queues in each stage for efficient parallelism. For distributed architectures, the Emitter can send data to the Workers using round-robin or on-demand scheduling [GF17].

In particular, FastFlow allows an optimization that reduces resource usage when the Collector is unnecessary. The Farm can be expressed with *farm*(*e*, *w*) [Gri16], and Figure 2.6 shows its representation.

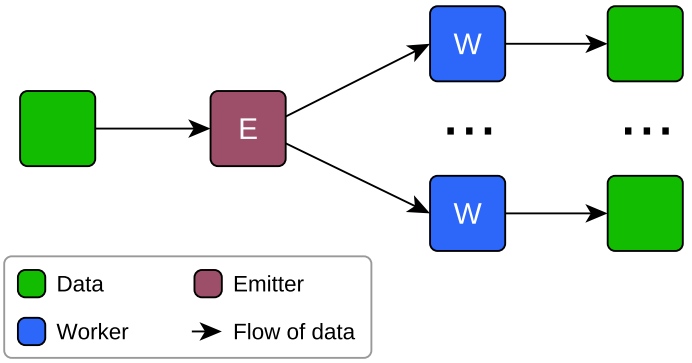

Figure 2.6 – Farm without Collector. Adapted from Pipeline example of [MRR12].

However, the Workers of a Farm might produce results out of order. This is a problem for applications such as audio and video processing, where the final result must be in the same order as the source material. To deal with this case, a Collector can be configured to process items in the order they were initially produced. A simple strategy is to tag each stream item with a sequential identifier, and then buffer out-of-order items in the Collector [GHDF18b]. Enforcing stream ordering needs extra runtime effort and may results in performance penalty [MRR12]. However, depending on the computation, the penalty may be not significant, according to [GHDF18b].

As seen in Section 2.3, windowing is necessary to perform aggregations in possibly infinite streams. The work of [DMM17] studied windowing mechanisms under an algorithmic skeleton approach. They proposed and implemented four new parallel patterns: Window Farming, Key Partitioning, Pane Farming, and Window Partitioning. We describe them briefly.

Window Farming is a simple approach where each window is buffered in the Emitter stage, and then each window is sent to a different Worker. This is shown in Figure 2.7. Window farming increases output but may not optimize for latency, because each window is processed sequentially by the Workers. An optimization exists to reduce latency by having Workers control the window buffering, while the Emitter just performs a round-robin load balancing.

Key Partitioning is similar to Window Farming. A key is used to split the stream, but parallelism only happens for each key. This becomes a problem for skewed data, where it becomes difficult to perform load-balancing. Items with the same key are processed by the same node in order.

Pane Farming is the most elaborated pattern. We show a computing graph in Figure 2.8. Pane Farming divides each window into panes, whose sizes depend on the

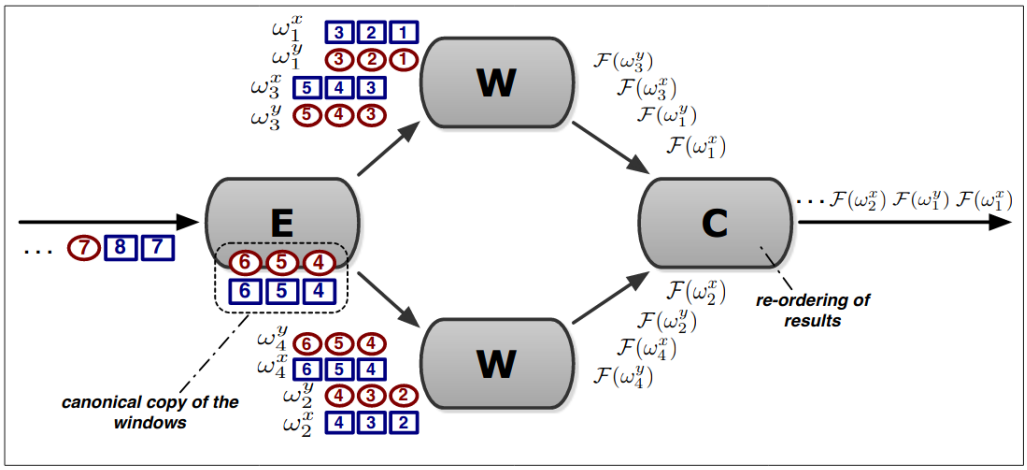

Figure 2.7 – Window Farming. Extracted from [DMM17].

eviction policies (window size and step). The pattern can be used if the function *f* executed by the stage can be decomposed into two functions *h* and *g* that can be used in the form $f(w) = h(g(p_1), g(p_2), ..., g(p_d))$ where p_n is a pane of the window, and the window *w* has *d* panes. The functions are used to calculate aggregates over panes and then merge the results using 2 farm stages. This process can also be done by key. Pane Farming improves throughput and latency.

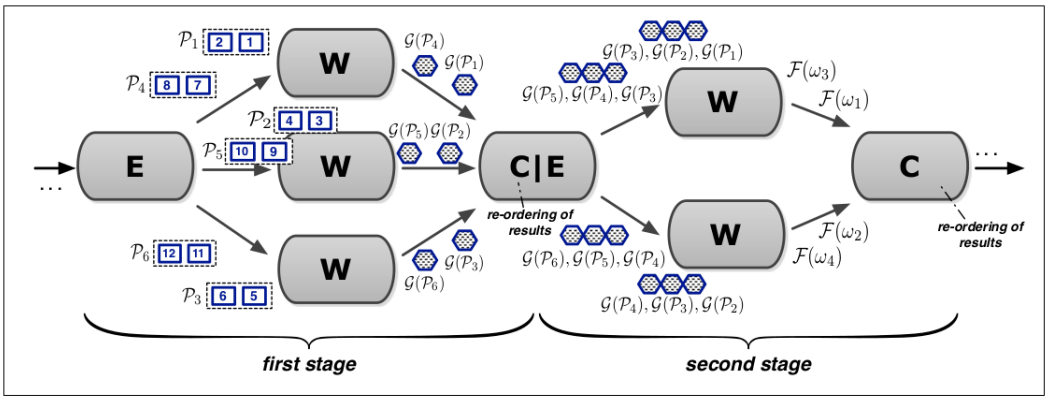

Figure 2.8 – Pane Farming. Extracted from [DMM17].

Window Partitioning implements the Map-Reduce pattern for data streams. A window is divided between *n* Workers, and at any time, only one window is being processed by the pattern. Since each Worker has a piece of the window, this pattern requires the Workers to perform window buffering logic. In a keyed scenario, each Worker must keep a window buffer per key. The Map function is executed in parallel for each partition (using a Worker Farm), and then the results are sent to the Collector to perform the Reduce function. This pattern optimizes for latency and slightly improves throughput. A diagram is shown in Figure 2.9.

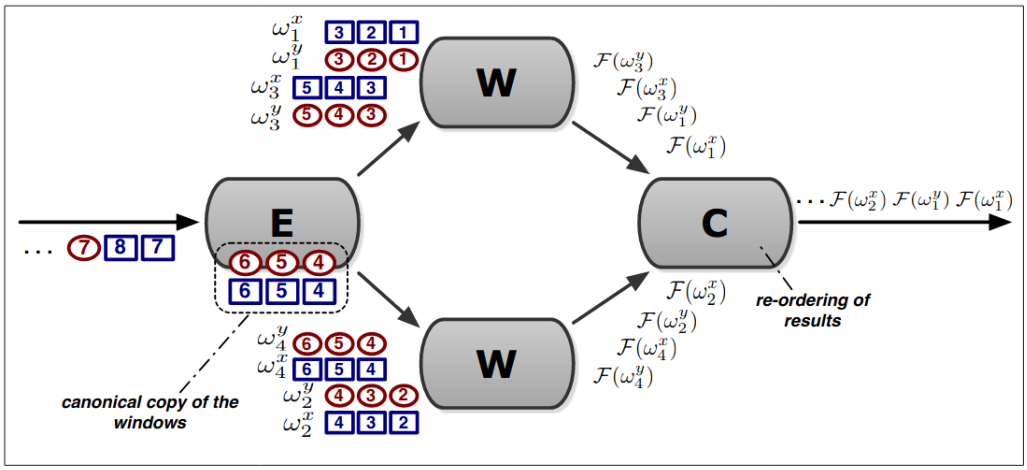

Figure 2.9 – Window Partitioning. Extracted from [DMM17].

Each pattern has its strengths and weaknesses. However, [DMM17] briefly suggests that, since skeletons can be nested arbitrarily, these patterns can be used together to improve latency and throughput at the same time.

So far we discussed the used of skeleton libraries for stream processing. Instead of using libraries, another approach is the use of DSLs for stream parallelism. In particular, StreamIt [TKG+01] is a DSL that has been developed for almost two decades at MIT. A newer approach is SPAR [Gri16], which supports C++11. The code is parallelized using C++ annotations. These tools and DSLs will be described in Chapter 4.

2.8 Frameworks and Libraries for Distributed Streaming

In this section, we present tools for the distributed stream processing domain. We explore the following characteristics: Design goals, runtime characteristics, serialization on distributed scenarios, the language in which the tool was implemented, how stream parallelism is achieved, back-pressure and fault-tolerance characteristics. Many of these tools are widely used *Big Data* frameworks.

To describe the API of each tool, we based on the work of [Mis17], which makes a distinction between *declarative* and *topological*-oriented APIs. There is an overlap between the tools we analyzed and the tools [Mis17] analyzed. We use the classification given by their work when this overlap exists.

2.8.1 Akka Streams

Akka Streams [Lig19] is a Java, Scala and .NET [Akk19] library that allows the programmer to define stream processing pipelines using a dataflow-like API. The library is part of Akka, an actor model implementation designed to build message-driven concurrent systems.

Akka Streams focuses on providing a minimal and consistent API, with compositionality as a key principle. The API is also designed to provide a complete solution for distributed stream processing, being able to express any topology (i.e. pipelines, graphs) found in the stream processing domain.

Akka Streams achieve these design goals through an API that allows programmers to define computation graphs. Each node in the graph has a given number of input and output "ports". The building blocks of the API consist of 3 nodes: Source, Sink, and Flow. The Source node has exactly one output port and is used at the beginning of a computation graph. The Sink node has exactly one input port and is used to express the end of a computation. The Flow node has exactly one input and one output port. It is used to express generic steps connecting downstream and upstream nodes.

Each node may contain several processing stages. This is achieved by providing a processing stage API that is idiomatic to the programming language. In the Scala API, this is done by using functions like `map`, `flatMap`, `filter`, and others. These functions are also part of the Scala enumerator API. In the C# API, the API is similar, where the programmer can use functions that are part of the `IEnumerator` API such as Select (equivalent to `map` in Scala), SelectMany (equivalent to flatMap) and Where (equivalent to filter). Each processing stage can also be marked as asynchronous, which creates an *async boundary*. An example of a word count program using the C# Akka Streams API is shown in Listing 2.1.

```
1 var words = ... //a stream of text
2 var counts = words
3   .GroupBy(10, x => x) //max groups: 10
4   .Select(x => new { Word: x, Count: 1 })
5   .Async()
6   .Sum((l, r) => new {
7     Word: l.Word,
8     Count: r.Count + l.Count})
9   .MergeSubstreams(); //merge groups
```

Listing 2.1 – Akka Streams. Adapted from [Akk19].

An async boundary separates groups of processing stages that will execute in parallel. In Listing 2.1, the `Async` method separates the computation in 2 groups: the first group is composed by `GroupBy` and `Select`. The other group is composed by `Sum`. These 2 groups run in parallel. Effectively, an item crossing an async boundary is forwarded to the next node

using asynchronous messages, or simply, an actor message. If no async boundaries are defined, then all processing stages are executed synchronously in the same actor (a process called “operator fusion” [AGT14]) without any performance penalties that come from sending an asynchronous message to another actor. Therefore, adding an async boundary increases processing overhead, but provides parallelism between stages.

Another way to achieve parallelism is by using nodes of type Balance and Merge. Balance is a node that has one input and many output ports. Each item processed is then forwarded to one of the output ports. Merge is a node that has many input nodes and one output ports, where each item received in any input port is processed and forwarded to the output port. To achieve parallelism, a Flow must be marked async and also must be put between the Balance and Merge. Additionally, each output port of the Balance node must be connected with the Merge node. By having both a graph-based and functional-based APIs, we consider that Akka Streams has a hybrid topological and declarative API.

Akka Streams provides a back-pressure mechanism. In general, items are produced by upstream nodes according to the demand signaled by downstream nodes. Upstream nodes may employ several strategies to slow down item production if downstream nodes are not able to cope with item production. Some of these strategies depend on whether it is possible to just not produce an item (if the production can be controlled), buffering items, drop items or ultimately tearing down the stream. These strategies are back-pressure strategies, preventing upstream nodes from overflowing the stream with items. Back-pressure comes with Akka Streams by default and is automatically handled, although the programmer can define buffer sizes for each node. Additionally, if consumers are faster than producers, then the producers can keep pushing items to the stream since downstream nodes keep signaling demand.

Akka Streams is built on top of the Akka actor framework. Actors encapsulate state and behavior, which is similar to a class in object-oriented languages. Actors, however, communicate via asynchronous messages, and each actor has a “Mailbox”, which is a queue of messages to be processed by the actor. Additionally, actor frameworks such as Akka provide an actor hierarchy (actors may have child actors) and a supervision strategy to handle failures. In general, each Akka Stream node in the graph is an actor, and async boundaries separate groups of processing stages in different actors.

Fault-tolerance is implemented using Akka fault-tolerance mechanisms. However, unless explicitly stated, any error will tear down the entire stream, meaning all processing will stop. Akka Streams has a distinction between errors and failures. Errors are normal data items and are expected in normal execution. The distinction between an error and a success can be done using Scala’s `Either[T1, T2]` type. Since it is a normal stream item, the programmer must detect and deal with them manually. Failures occur when the stream itself had an error. Exceptions are considered failures (for instance, a Java `DivisionByZeroException`). Akka Streams offers a wealth of recovering options, which include dropping

the item from the stream and continuing processing, tearing down the stream, or retrying the item on the failing stage. Retrying can be useful in scenarios where the stage calls a remote network service that could timeout or fail for any reason. For timeouts, a common strategy is to use an Exponential Backoff strategy, where the retry is delayed by a delay that exponentially increases up to a given maximum value. An example of error handling in Akka Streams is shown in Listing 2.2. The listing is written in C# using Akka.NET, but the Java and Scala versions are similar.

```
var planB = Source.From(new List<string> {
  "five", "six", "seven", "eight"
});
Source.From(Enumerable.Range(0, 10))
  .Select(n => n < 5 ? n.ToString() :
      throw new ArithmeticException("Boom!"))
  .RecoverWithRetries(attempts: 1, exception =>
      exception is ArithmeticException ? planB : null)
  .RunForeach(Console.WriteLine, materializer);
```

Listing 2.2 – Error handling in Akka Streams. Adapted from [Akk19].

The API supports defining windows with simple grouping functions. The function `groupedWithin` creates a non-overlapping window by time or count. For instance, the programmer can generate windows every N minutes with a maximum of M items. The `sliding` function allows to create a sliding window for a given number of items and is also a function of the Scala enumerator API. Therefore programmers familiar with Scala may also find this function intuitive enough.

Akka Streams supports distributed computing with Akka Cluster and StreamRefs. In general, Akka does not provide any delivery guarantees, which is a deliberate design decision, inherited by Akka Streams. Sending an item over a network connection may cause failures, and the user must deal with it manually. Additionally, the design principles of Akka are the same as Erlang regarding delivery guarantees, where the user itself must implement idempotent logic to deal with at-least-once deliveries. The user must retry an operation if it fails, which could lead to an item being received twice in the other actor or stream operator.

To send a message between separate computers (or JVM and .NET instances), the message must be serialized. Akka supports any serialization strategy. By default, the JVM implementation uses Protocol Buffers, but the user may choose the serialization method (global or type-specific). The .NET implementation uses Protocol Buffers as well, but a common option is to use JSON. Additionally, the .NET implementation may use Hyperion, a library that provides polymorphic serialization, where the types of each field are also present in the serialized format.

2.8.2 Apache Flink

Apache Flink [CKE+15] is a platform for distributed stream processing written in Scala, designed for high availability and high scalability. Flink provides many APIs with different abstraction levels. The lowest-level API is the DataStream API, which resembles the API of common stream parallelism tools such as Akka Streams and Storm. There are high-level APIs such as Table API and SQL, which makes data streaming available to a wider audience that is familiar with relational databases. However, we focus on the DataStream API. We notice that there is also a DataSet API, but a DataSet in Flink is a stream that has an end, while DataStream represents potentially infinite streams. DataSet and DataStream are part of the Core API, and in the following paragraphs, we will refer to both APIs as "Core API" for simplicity, since they are similar.

The building blocks of the Core API are data sources, transformations, and data sinks. Flink comes with predefined data sources and data sinks, which may be in-memory collections, files, network sockets, and standard input. Moreover, third-party connectors are available. For instance, Apache Kafka, Cassandra, Amazon Kinesis, Twitter Streaming API (source only) and others. Projects like Apache Bahir provide extra third-party connectors.

Stream transformations can be done via the Operator API. The expected operators are available, such as Map, FlatMap, Reduce, and others, which is considered to be a declarative API [Mis17]. The API provides a way to logically partition streams by a key using the KeyBy operator. The operator will output many streams. The next operators will be applied to each logical stream. This allows Flink to parallelize stream processing based on the logical partitions, while a non-keyed stream will not be processed in parallel. Additionally, Flink processes all stream items in FIFO order by default, however, this does not work when the stage outputs multiple streams. This behavior is the same as Akka Streams, as described in Section 2.8.1.

Apache Flink provides a rich API for windowing. There are 3 types of windows: tumbling, sliding, and session. Tumbling is fixed-size and non-overlapping, while sliding is overlapping. Tumbling and Sliding can be configured based on time length or item count. The session windowing strategy generates windows based on bursts of activity, where the user specifies a maximum time of inactivity. After the configured time passes and no items arrive at the operator, a new window is generated. The windows are non-overlapping but may have varied size. There is another type of window called “global”, which is useful when the programmer needs to specify a custom windowing strategy.

In listing 2.3, we show an example of Apache Flink using the Scala programming language. The example takes a stream of text lines and prints a count of the words received in the last 5 seconds, printing the result each second. In the example, `WordWithCount` is a class with fields `word` and `count` which are used to group and aggregate elements.

```
val env = StreamExecutionEnvironment.getExecutionEnvironment

//read socker text stream lines
val textStream = env.socketTextStream(
  "localhost", 8080, '\n')

val windowCounts = textStream
  .flatMap { w => w.split("\\s") } // line to words
  .map { w => WordWithCount(w, 1) } //parse
  .keyBy("word") //group by "word" field
  .timeWindow(Time.seconds(5), Time.seconds(1)) //window
  .sum("count") //aggregate

//single-thread print
windowCounts.print().setParallelism(1)
env.execute("Socket Window WordCount")
```

Listing 2.3 – Word counting in Flink. Adapted from [Fli19].

Back-pressure is handled by using buffers. Each stage has a buffer size containing unprocessed items. If a stage attempts to add items to a stage with no available buffer space, this will cause back-pressure. When back-pressure happens, the stage will stop and wait for buffer space to become available. Special care is taken where two stages are processed in different machines, where Flink never writes too much data in the network channel. However, it is recommended to check whether back-pressure problems are happening. Flink provides monitoring, in which it is possible to check which node is causing excessive back-pressure.

Apache Flink also provides a fault-tolerance method called Checkpointing, which requires the use of a persistent data stream system such as Apache Kafka that is a persistent log-based messaging system. If Apache Flink has a failure, the computation state can be restored by using a checkpoint, which stores the stream number emitted by the data source, and the computation state of each stage. When all sinks receive a checkpoint, the checkpoint itself is acknowledged by the job manager. By using checkpoints, Apache Flink only reprocesses source data produced after the last acknowledged checkpoint. The programmer can choose the processing guarantees depending on the sources and sinks used. Using Apache Kafka as a source and sink, Flink can guarantee exactly-once processing (meaning that the final state reflects the processing of each item exactly once), and for low-latency scenarios, the programmer can use a more relaxed at-least-once guarantee. These guarantees are highly dependent on data sources and data sinks. For instance, a TCP socket data source has only at-most-once guarantees.

Flink implements a serialization framework that can automatically deal with POJO (Plain Old Java Object) objects, primitive types and their boxed variants, and core Java classes such as lists and sets. More complex types such as generic types are handled by the Kyro library, and the programmer may provide any serialization method when Flink or Kyro fail. The user can also register types beforehand to speed up serialization. For

instance, Flink can correctly serialize and deserialize objects based on their supertypes, but it is faster if the user can inform all subtypes that will be used during execution beforehand.

Apache Flink uses Akka, the actor framework introduced on Section 2.8.1 to coordinate remote communication. The architecture of Flink consists of a Job Manager and Task Manager. The Job Manager is responsible for coordinating task execution, while the Task Manager is responsible for executing tasks and managing stream buffers. However, Akka is used only for coordination; actual data streams between remote nodes are done via network connections that are managed by a network manager in each Task Manager. This mechanism is described in Figure 2.10.

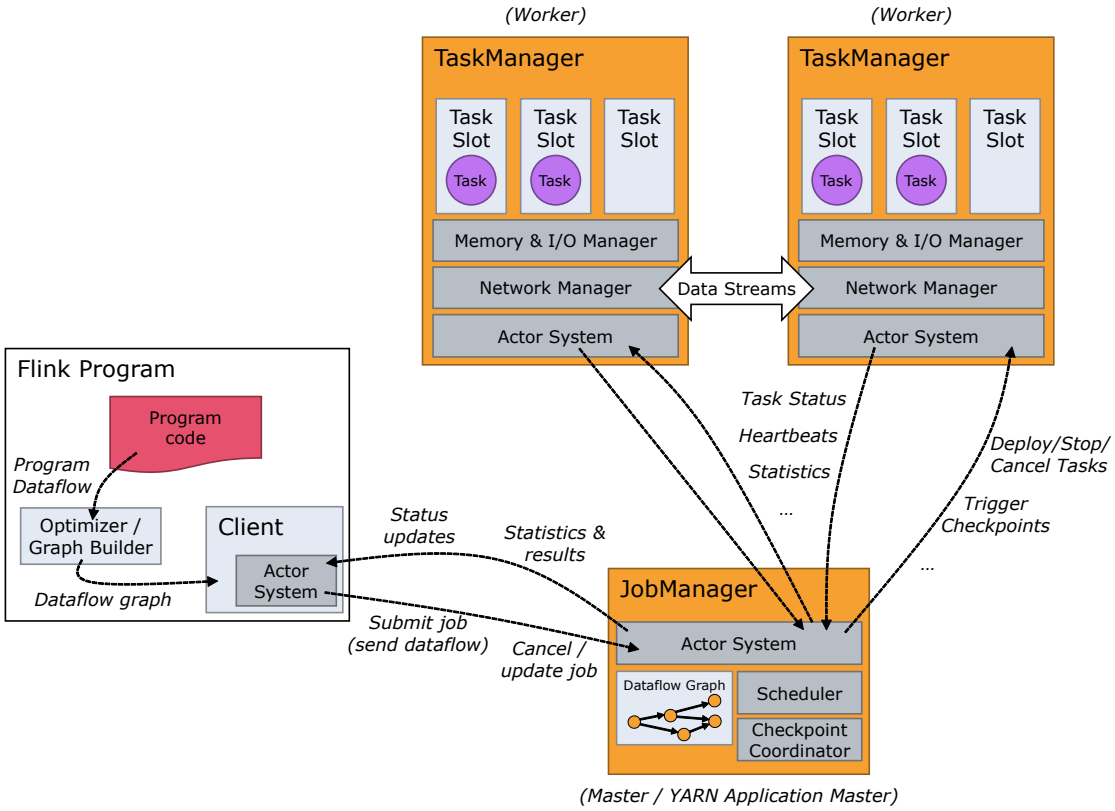

Figure 2.10 – Flink Job Manager. Actor System is Akka. [Fli19].

2.8.3 Apache Storm

Apache Storm [Apa19] is a real-time stream processing system written in Java. Storm is designed to be easy to use and can be used in any language that has an implementation of the Thrift protocol, which is a protocol that comprises RPC and serialization, similar to Protocol Buffers.

The building blocks of Apache Storm are Spouts and Bolts. Spouts are data sources while Bolts process data sources, possibly generating other data streams. Bolts

are used for data transformations and data sinks. For the rest of this section, we will refer to spouts and bolts as “components” for simplicity. Each component has a name, and a Topology represents components connected, creating a graph. The connection between components is done by groupings.

A grouping is a set of components connected by a name. Several types of groupings are available, and the most basic grouping available is the Shuffle grouping. Shuffle means that each bolt connected to another component will receive an equal amount of data items. Like Apache Flink, it is also possible to define logical partitioning using the Fields grouping, which defines a key to partition the stream. However, Storm can deal with key imbalance using the Partial Key grouping, which implements an algorithm that provides better load balancing when data is skewed towards a given key.

The programmer can implement classes that inherit from `IRichSpout` and `IRichBolt` to create spouts and bolts using the Topology API, which, as its name suggests, can be considered a topological API [Mis17]. The spout implementation must have a `nextTuple` method that returns a data item to be inserted in the stream. Likewise, the bolt must implement a similar `nextTuple` method, which receives by parameter the next item from the stream. Moreover, spouts and bolts must declare their outputs using an `OutputFieldsDeclarer` object. Additionally, the parallelism level of each component can be indicated on the component itself using the methods `setBolt` and `setSpout`. Listing 2.4 shows an example of the Topological API. We do not demonstrate how to create a bolt and a spout for simplicity.

```
1 TopologyBuilder builder = new TopologyBuilder();
2
3 builder.setSpout("sentences",
4   new RandomSentenceSpout(), 5);
5 builder.setBolt("split", new SplitSentence(), 8)
6   .shuffleGrouping("sentences");
7 builder.setBolt("count", new WordCount(), 12)
8   .fieldsGrouping("split", new Fields("word"));
```

Listing 2.4 – Word counting in Storm Topology API [Apa19].

Although the API’s topology is verbose, it is very simple to understand. The topology has at-least-once delivery guarantees while a higher-level Trident API ensures exactly-once guarantees. The Trident API has a simpler and less verbose API, including support for a declarative-style definition of the computation [Mis17]. Additionally, there are two experimental APIs for even higher-level declarative stream processing: the Streams API that closely resembles the Java Stream API, which lets the programmer use lambda functions for expressive stream computations in enumerators, and an SQL API. Listing 2.5 shows an example of the Trident API.

```
1 FixedBatchSpout spout = new FixedBatchSpout(
2   new Fields("sentence"), 3,
```

```
3     new Values("the cow jumped over the moon"),
4     new Values("the man went to the store"),
5     new Values("four score and seven years ago"),
6     new Values("how many apples can you eat"));
7   spout.setCycle(true);
8
9   TridentTopology topology = new TridentTopology();
10  TridentState wordCounts =
11    topology.newStream("spout1", spout)
12      .each(
13        new Fields("sentence"),
14        new Split(),
15        new Fields("word"))
16      .groupBy(new Fields("word"))
17      .persistentAggregate(
18        new MemoryMapState.Factory(),
19        new Count(),
20        new Fields("count"))
21      .parallelismHint(6);
```

Listing 2.5 – Word counting in Storm Trident API. Adapted from [Apa19].

The runtime of Storm is comprised of a worker process, executors (threads) and tasks. The parallelism hint passed in the `setBolt` and `setSpout` methods refer to the initial number of executors (threads) that will be spawned for the component, and can be changed at runtime. Each component also has a parameter for the number of tasks, which is the maximum number of parallel instances of a component. Usually, this is set to the same number as the number of executors, but the programmer can set it to 2 or more tasks per executor to allow for more flexibility when increasing the number of threads when more servers join the cluster.

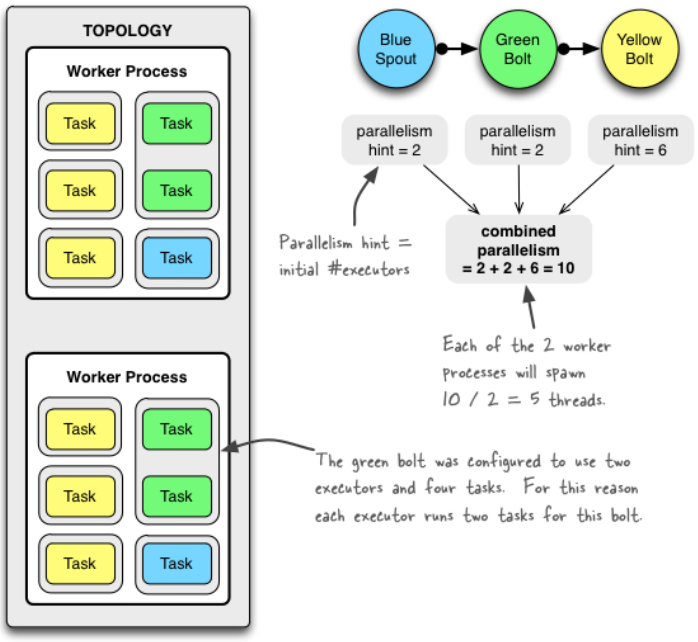

Figure 2.11 – Apache Storm Topology [Sto19].

Figure 2.11 shows a demonstration of the parallelism hint. The Green Bolt was set to 2 executors and 4 tasks (the task parameter is not present in the picture). Effectively, the maximum parallelism of the Green Bolt as shown in the picture is 2, however, if more worker processes and executors are added to the cluster, the Green Bolt will benefit from this extra capacity, possibly reaching a parallelism level of 4. The Blue Spout, however, will at most run in 2 threads, because the number of tasks is equal to the number of executors. The topology needs to be redeployed to the Storm cluster if the programmer wants to change the number of tasks.

The back-pressure mechanism is similar to Apache Flink, where each component has a buffer with limited capacity. Storm components have 2 buffers: a sending buffer and a receiving buffer. Storm checks how full the receiving buffer is before adding an item to the buffer, and if it passes a certain threshold, the spout is throttled.

Apache Storm runs in a cluster managed by Apache Zookeeper, and communication is done via the Thrift protocol. Zookeeper is used only for coordination. Communication between components in the same worker process is done via LMAX Disruptor, a high-performance inter-thread messaging library. Communication between components in different worker processes is done via ZeroMQ. Since Apache Storm can be managed via the Thrift protocol, it enables any language with a Thrift client to communicate with Apache Storm. Currently, Python, Ruby, and Javascript are supported by the Apache Storm team, while a C# client is maintained by the Microsoft Azure team.

2.8.4 Apache Spark

Apache Spark [ZCF+10] is a general-purpose cluster computing system written in Scala. It executes computation graphs in a Spark cluster and has a rich set of APIs for batch processing, machine learning, graph processing, and micro-batch streaming. In general, the APIs are designed to be easy to use and supports many languages, such as Java, Scala, Python, and R.

The building block of Spark is a data structure called RDD (Resilient Distributed Dataset). Any RDD can be built from in-memory collections or Hadoop-compatible data sources. The RDD parallelizes operations automatically while also allowing logical partitioning with keys similar to Apache Flink and Storm. Common operators are available, like map, filter, aggregate, and others.

The RDD is resilient and can be restored if any computing node in the cluster fails. The RDD also carries a history of operations that lead to the current state, so that when an RDD fails, it can be restored later based on the history of computations, assuming that all operations are deterministic.

For the streaming domain, Spark offers a declarative DStream API, which allows consuming data from live data streams, such as Kafka topics, Twitter, TCP sockets, and others. DStream means Discretized Stream and is composed of a sequence of RDDs. Each RDD can be thought as a micro-batch, where each RDD comprises of 200 milliseconds of streamed data. This setting can be configured for smaller, more frequent batches. Data transformation operators are applied to each RDD in the stream. This process is shown in Figure 2.12 and an API example is shown in 2.6.

Figure 2.12 – Spark DStream mini-batching [Spa19].

Besides supporting common streaming operations via micro-batching, Spark also allows windowing (sliding and non-sliding) with duration and step. The APIs are rich and support several levels of abstractions, from RDDs to Datasets and SQL. Spark also has APIs for graph processing (GraphX), machine learning with MLlib and support for the R programming language.

```
1 val textFile = sc.textFile("hdfs://...")
2 val counts = textFile
3   .flatMap(line => line.split(" "))
4   .map(word => (word, 1))
5   .reduceByKey(_ + _)
6 counts.saveAsTextFile("hdfs://...")
```

Listing 2.6 – Word counting in Spark. Adapted from [ZCF+10].

Serialization in Spark can be done with default Java serialization or a library called Kyro, which is faster and generates a more compact representation than Java. However, Kyro comes with several shortcomings regarding support for all Java serializable types, and the user must register all classes that can be serialized upfront. Spark uses Kyro for internal data structures, such as RDDs composed of known simple types. It is recommended to try Kyro for network-intensive applications.

Delivery guarantees depend on data sources. With resilient distributed data sources like HDFS, Spark can provide exactly-once processing. However, data output is at-least-once, meaning the same data may be outputted twice. The user must implement idempotent property if needed. Regarding reliability, back-pressure is handled automatically but must be enabled in the configuration. Spark measures batch scheduling delays and processing times to determine how fast the system can process incoming data.

2.8.5 Apache Heron

Apache Heron [KBF+15] is a streaming system maintained by Twitter written in Java and C++. Twitter used Storm in production for several years, but Storm presented may shortcomings at the scale that Twitter operates, regarding performance and debugability. We noticed that Heron was released as open-source around 2016, and a complete rewrite of Apache Storm core (from Clojure to Java) was completed in 2019, therefore, many shortcomings of Storm back in 2015 may have been addressed by newer releases.

One of the key features of Heron is compatibility with Apache Storm API, where it is possible to easily migrate applications from Storm to Heron. However, Heron also offers the Streamlet API which is heavily inspired by Java 8 Stream API. It is still possible to use Spouts and Bolts, as described in Section 2.8.3. Therefore, all features supported in Storm are also supported in Heron. In this section, we briefly describe the differences between Heron and Storm.

Heron has 2 core architectural components: Stream Manager and Topology Manager. The topology manager manages stream processing topologies, which are composed of a logical plan and a physical plan. The logical plan contains information on the operations that will be executed and how components connect each other. After submitting a topology to the Topology Manager, it creates the physical plan, which defines how the processes are divided between containers. A container has a Stream Manager and multiple Heron Instances. The Stream Manager is responsible for routing data items between components (spouts and bolts) and connect to other Stream Managers in other containers. Additionally, Stream Managers deal with back-pressure, which is similar to the current strategy used in Storm. Essentially, the Stream Manager communicates with other Stream Managers to slow down the spout that feeds into the lagging component.

This process is shown in Figure 2.13. The bolt B3, shown in Red in the container A is operating at a slower speed than normal. The stream manager slows down the feeding of S1 into B3 for back-pressure, but this alone does not solve the problem, since the network communication from containers B, C and D is still happening. If the container A does not consume from the network, the network buffer will fill up. Therefore, the Stream Manager in container A communicates with B, C, and D that a back-pressure is happening. Finally, containers B, C and D will analyze the physical plan of A and stop feeding data to the S1 bolt in container A. When B3 begins normal operation, container A will communicate with B, C and D to resume normal operation and stop the back-pressure.

The Streamlet API aforementioned is a declarative API, very similar to the Java 8 Stream API. The Streamlet API allows to define type-safe stream operations without the verbosity and debugging challenges presented by the original Topology API present in Storm. The API is similar to the Akka Streams API, where key components are Sources, Operators,

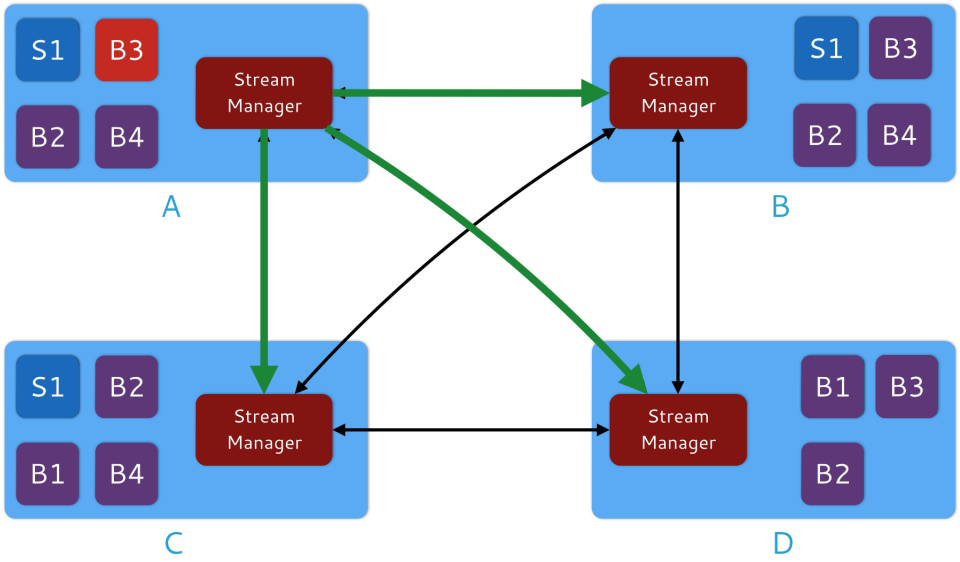

Figure 2.13 – Heron back-pressure mechanism [Her19].

and Sinks. Heron automatically converts these components into spouts and bolts. The user can also chain Streamlet operations such as map, flatMap, filter, and others. Similar to Flink and Storm, windowing and logical partitions by key are also available. Listing 2.7 shows an example of the Heron Streamlet API.

```
Builder builder = Builder.newBuilder()
  .newSource(() -> "mary had a little lamb")
  .flatMap(sentence -> Arrays.asList(
    sentence.split("\\s+")))
  .reduceByKeyAndWindow(
    // Key extractor
    word -> word,
    // Value extractor
    word -> 1,
    // Windowing
    WindowConfig.TumblingCountWindow(50),
    // Reduce operation (a running sum)
    (x, y) -> x + y
)
// The result is logged
.log();
```

Listing 2.7 – Word counting in Heron Streamlet API. Adapted from [Her19].

As usual, delivery guarantees are highly context-dependent. Heron uses the term "effectively-once" as an alternative to "exactly-once" since the term in some contexts can be misleading. The description given by the Heron documentation of what “effectively-once” means is similar to the exactly-once guarantees of Apache Flink: The final state of the computation will reflect the processing of each item once, but an item can be processed more than once. For effectively-once delivery guarantees, Heron requires the stream state

to be rewindable, like a Kafka stream. Additionally, Heron requires the topology to be stateful and idempotent.

2.8.6 Apache Samza

Apache Samza [NPP+17] is a distributed real-time stream processing system. Apache Samza is designed to solve problems related to the *Lambda architecture*, which consists of keeping duplicated logic for batch and stream processing as well as the high costs regarding code maintenance that it causes [Mis17, NPP+17, Cur18].

The use case presented by Samza is as follows: it is common for companies to keep 2 systems in parallel: a stream processing system that deals with live data streams (Flink, Storm, etc), and a batch processing system that reprocesses the whole dataset (Hadoop, Spark, etc). Meanwhile, it is necessary to reprocess the whole dataset sometimes, given that business logic code changes very often, and bugs are inevitable in practice. Therefore, the code changes must be done in both the stream processing code and the batch processing code. Samza offers a unified model for batching and processing, meaning that programmers only need to change the code in one place. This may also reduce infrastructure costs since there is no need to keep separate servers for both deployments.

When a code change happens, sometimes reprocessing needs to be applied in the whole dataset. However, with appropriate monitoring, bugs can be quickly caught. The reprocessing needs to be applied only for the data items produced in the latest minutes. Samza is designed to make this operation simpler. Also, in distributed systems, data items may arrive late and out of order. Samza is designed to be able to detect this case, rewind the state and reapply the operations so that the state reflects the processing of data items in the correct order.

In Apache Samza, application state is kept in the local disk or RAM. For each incoming data item, Samza processes it and the state changes. This change of state is stored in a changelog, which is often implemented as a Kafka topic that can be rewound and replayed. The changelog is used when a node cannot be restored after a failure. In general, Samza tries to restore the node in the same physical machine where its application state was being recorded, which helps to recover application state faster. This mechanism is opposed to other streaming systems that store data in a remote storage system, causing high latency, excessive resource usage and long recovery times. Samza calls this mechanism Host Affinity. In experiments at LinkedIn, Samza has used the Host Affinity strategy effectively in 85% of restart cases, reducing recovery time from 30 minutes to 30 seconds. Thus, the main differential of Samza is being able to handle large application states, up to 100s of terabytes in size.

Another aspect in which Samza differs is in the back-pressure mechanism. Samza uses Apache Kafka for streaming, and Kafka writes data items into a log file in the disk. Effectively, there is a large disk buffer between two computation stages, meaning that Kafka handles the buffering between stages. Samza itself has no back-pressure mechanism due to this characteristic. Samza offers at-least-once processing guarantees.

Samza has a rich set of APIs that cover many use cases, in a similar way to Heron and Spark. There is a High-Level Streaming API that supports common operators using a similar API to the Java 8 Stream API. Samza supports windowing, sliding window and logical partitioning of streams using the PartitionBy operator. Additionally, there is a low-level Task API, in which the programmer can implement an interface with a process method, while the output of the task is collected by a Message Collector. An example of the high-level streaming API is shown in Listing 2.8.

```
1  lines
2    .map(kv -> kv.value)
3    .flatMap(s -> Arrays.asList(s.split("\\W+")))
4    .window(Windows.keyedSessionWindow(
5      w -> w, Duration.ofSeconds(5),
6      () -> 0,
7      (m, prevCount) -> prevCount + 1,
8     new StringSerde(),
9     new IntegerSerde()), "count")
10   .map(windowPane ->
11    KV.of(windowPane.getKey().getKey(),
12        windowPane.getKey().getKey() + ": "
13        + windowPane.getMessage().toString()))
14   .sendTo(counts);
```

Listing 2.8 – Word counting in Samza Stream API. Adapted from [NPP+17].

2.8.7 Apache NiFi

Apache NiFi [NiF19] is a distributed stream processing system with a key differential: it represents computation graphs in a visual interface through a browser. The user can edit a graph in the browser using the many components that ship with NiFi. NiFi integrates with many tools such as Kafka, Amazon S3, Hadoop, and others.

NiFi is a system that tries to solve the problem of frequent changes in stream processing application code and how to connect systems that originally may have not meant to operate together. The key building blocks of the graph are FlowFiles and Processors. A FlowFile is a piece of data (which we are calling data item in this section) and a processor is a NiFi component that acts as a data producer, transformer, and data sink.

NiFi provides a rich interface that allows defining a graph and monitor execution. There is an extensive set of predefined processors (more than 300). NiFi features an Expression Language to work with data. NiFi also has a Java API to define new processors. This makes NiFi a good tool for abstracting stream processing and possibly enabling a wider audience to use streaming tools. Additionally, the interface shows performance statistics and diagnostics to check whether a stage in the graph is facing any problems.

Apache NiFi can run in a cluster managed by Zookeeper. A node of the cluster is elected by Zookeeper as the primary node, which will replicate any changes to the graph to other nodes. NiFi offers "guaranteed delivery", which we conservatively assume to mean at-least-once delivery. NiFi achieves this by writing data items in a persistent write-ahead log. It also supports back-pressure by monitoring queue sizes and setting an expiration time for data items. Some performance aspects can be tuned in a fine-grained, processor-specific fashion, to allow a choice between low latency and high throughput as well as data loss tolerance.

2.8.8 Kafka Streams

Many of the distributed stream processing frameworks presented in this chapter implement message delivery guarantees, often expressed in terms of "at-least-once" or "exactly-once" semantics. Frameworks often rely on external data producers that can rewind and replay data to provide exactly-once semantics, which generally means that the final state of the computation reflects the processing of each item exactly once. Apache Kafka [KNR11] provides these features.

Kafka is a distributed streaming platform. The core of Kafka is composed of messaging topics and partitions implemented as a commit log, in which each message has a unique offset. The commit log is persisted for a configured amount of time. A common use-case for Kafka is for Publish & Subscribe systems, where Kafka provides the Producer and Consumer API. However, Kafka also has a Stream API, which will be discussed in this section.

The Stream API has three building blocks: A Source Processor, Stream Processor, and a Sink Processor. The Source Processor is a stream processor that has no input streams. Instead, it only consumes messages from a one or multiple Kafka topics, and produces a stream which is another Kafka topic (with the same characteristics regarding offsets, replayability, ordering, etc). A Sink Processor takes input streams and sends them to a specific Kafka Topic. The Stream Processor takes a stream and apply stateful or stateless transformations on it, and then produces one or more output data items to downstream processors. The Kafka Streams DSL provides common operators such as map, flatMap,

filter, aggregations, windows, and others. Operators can be chained together, and support lambda functions in Java 8+ and Scala, as shown in the example in Listing 2.9.

```
KStream<String, String> textLines = builder.stream(
      "streams-plaintext-input",
      Consumed.with(stringSerde, stringSerde));
KTable<String, Long> wordCounts = textLines
    .flatMapValues(value ->
       Arrays.asList(value.split("\\W+")))
    .groupBy((key, value) -> value)
    .count();
wordCounts.toStream().to("streams-wordcount-output",
  Produced.with(Serdes.String(), Serdes.Long()));
```

Listing 2.9 – Word counting in Kafka Streams API. Adapted from [KNR11].

Fault-tolerance is a key feature of Kafka itself. Therefore, this property extends to Kafka Streams as well. When a node or computation fails, Kafka can rewind state and replay a changelog to rebuild the computation state. This is possible because all topic messages are durable and have an offset, which enables restoring the stream to a previous point in time. This strategy of having a changelog is also used by Apache Samza, which itself uses Kafka to enable this feature. However, Samza also includes a Host Affinity mechanism to restore state faster without replaying the whole changelog. Another aspect similar to Apache Samza is the lack of back-pressure. Kafka consumers operate in a pull-based messaging model, so the consumers can choose the rate at which they consume data. Moreover, since Kafka writes to a commit log in the disk, the buffer space is very large for any connection between producers and consumers.

Since Kafka Stream relies on Kafka topics, other characteristics such as scalability and performance are also dependent on the capacities of Kafka itself. Kafka runs in a Zookeeper cluster to provide distributed computing. To add a node to a Kafka cluster, the number of partitions in a topic must be changed. Moreover, once created, the number of partitions in a topic cannot be reduced. Partitions are the way to achieve parallelism in Kafka. When a message arrives in a topic, one of the partitions will receive the message. However, these details concern low-level Kafka topics, which is out of the scope of this section.

Serialization in Kafka Streams is handled by default serialization and deserialization mechanisms. They can deal with simple types such as String and Java numbers. For more complex types, Kafka uses the JSON format. However, the serialization method can be completely overridden by the user. Therefore, Kafka Streams uses Kafka, a core component used for fault-tolerance in many other streaming systems. Kafka Streams provide an easy-to-use streams DSL over Kafka topics with exactly-once processing semantics.

2.9 Tools for distributed programming

In this section, we cover HPX and MPI, which are tools used for distributed-memory programming. While there are other tools, we focus on HPX and MPI. Both are significantly different from each other. HPX is a newer C++ runtime library for efficient parallelism exploitation while MPI is a mature and well-known standard for message passing.

2.9.1 Message Passing Interface

One of the most important challenges to be solved when developing a distributed system is how to coordinate several machines in a network towards the same goal. In this domain, message passing is one of the most successful paradigms available, and it has been extensively studied for decades. In 1994, several interfaces for message passing were made available, and a standardization effort was put in place. The result was the Message Passing Interface standard, also known as MPI [For12]. This effort studied several of the available tools at the time, and a standard was created to provide a solid foundation on which distributed applications can be built. The design goals of the MPI standard are focused on efficient communication, portability, language-independent semantics, and thread-safety.

The MPI standard covers a programming API and tools to execute MPI programs. The MPI standard does not *implement* MPI, but defines how implementations should behave. The standard defines the MPI for C and Fortran programming languages, although bindings for other languages also exist. MPI additionally defines startup utilities, such as `mpiexec` to run MPI programs. It is common for MPI implementations to provide also an `mpirun` utility, but it is not standardized. In this section we cover basic MPI concepts.

```
1  int main(int argc, char** argv) {
2      MPI_Init(&argc, &argv);
3      MPI_Comm comm = MPI_COMM_WORLD;
4      int rank;
5      MPI_Comm_rank(comm, &rank);
6
7      int senderRank = 0;
8      int receiverRank = 1;
9      if (rank == 0) {
10       int dataToSend = 42;
11       MPI_Send(&dataToSend, 1, MPI_INT, receiverRank, 0, comm);
12     }
13     else if (rank == 1) {
14       MPI_Status status;
15       int dataReceived;
16       MPI_Recv(&dataReceived, 1, MPI_INT, sourceRank, 0, comm, &
             status);
17       printf("received %i\n", dataReceived);
```

```
18      }
19      MPI_Finalize();
20  }
```

Listing 2.10 – MPI basic point-to-point communication.

The basic communication mechanism in MPI is point-to-point communication. This is shown in Listing 2.10. The MPI standard describes `MPI_Send` and `MPI_Recv` functions to send and receive data, respectively. In this case, there are 2 *processes* involved in the communication: the sender and the receiver. Each process is identified by a *rank* in a *communicator*. MPI defines the communicator `MPI_COMM_WORLD`, which provides a flat view of all process ranks available. Both `MPI_Send` and `MPI_Recv` have similar parameters:

- `buf`: the address of the data buffer to send to another process. This is also the receive buffer for the `MPI_Recv` call, and must be as large as the sent buffer.
- `count`: number of elements in the send buffer, or the number of elements to receive;
- `datatype`: the datatype of each element in the buffer;
- `dest`: the process rank of the destination. This parameter is only present in `MPI_Send`.
- `source`: rank of the source, or `MPI_ANY_SOURCE` to receive messages from any process. This parameter is only present in `MPI_Recv`.
- `tag`: a message tag, which can be used to distinguish between types of messages. `MPI_Send` must specify a tag, while `MPI_Recv` may use `MPI_ANY_TAG` to receive messages with any tag.
- `comm`: the communicator. Can be `MPI_COMM_WORLD` or a communicator created using other MPI functions.
- `status`: an object that informs the source process, tag, and error of the message. This parameter is only present in `MPI_Recv`.

In MPI, the sender "pushes" data to the receiver [For12]. The receiver can *match* a message using the `source`, `tag` and `comm` parameters to receive messages only from a particular process. However, the sender can also specify wildcards for `source` and `tag`, to receive messages regardless of the source process or message tag. Therefore, senders must know the receivers, while receivers can receive from any process. When wildcards are used, the `status` object will contain the source process and the actual message tag.

The receiver must allocate a buffer large enough to receive the data. This means that both sender and receiver must agree on the size of the communicated data. However, this is not always possible. A solution for this problem is to send 2 messages: the size

of the buffer first, and then the actual data. The receiver allocates a buffer based on the first message, and use the buffer on the second. Listing 2.11 shows a pseudocode for this solution. In this example, the sender allocates an array with a random size, which the receiver does not know. We use 2 messages: one for the size and another for the data itself. We set different tags in each message to specify their purpose. There is a built-in solution in MPI for this problem, without having to send 2 messages.

```
#define SIZE_TAG 0
#define DATA_TAG 1
MPI_Comm comm = MPI_COMM_WORLD;
if (rank == 0) {
  int size = random_between(1, 1000);
  int* buffer = new int[size];
  MPI_Send(size, 1, MPI_INT, 1, SIZE_TAG, comm);
  MPI_Send(buffer, size, MPI_INT, 1, DATA_TAG, comm);
  delete[] buffer;
}
else if (rank == 1) {
  int size = 0;
  MPI_Status status;
  MPI_Recv(&size, 1, MPI_INT, 1, SIZE_TAG, comm, &status);
  int* buffer = new int[size];
  MPI_Recv(buffer, size, MPI_INT, 1, DATA_TAG, comm, &status);
}
```

Listing 2.11 – MPI dynamic buffer size solution. MPI has built-in mechanisms for this purpose.

In Listing 2.12, we use `MPI_Probe` and `MPI_Get_count`. First, we probe MPI to test for a message. This function blocks until a message is received. `MPI_Probe` can also match on a wildcard source and tag. In this case, we probe without using wildcards. The `status` object is used in the `MPI_Get_count` to get the size of the transmitted data. Using this method, only one message is sent, but we first receive the *envelope* of the message. The envelope of the message are the source, destination, tag, and communicator. The `MPI_Get_count` uses the envelope and queries the size of the message.

```
#define ARRAY_TAG 999
MPI_Comm comm = MPI_COMM_WORLD;
if (rank == 0) {
  int size = random_between(1, 1000);
  int* buffer = new int[size];
  MPI_Send(buffer, size, MPI_INT, 1, ARRAY_TAG, comm);
  delete[] buffer;
}
else if (rank == 1) {
  int size = 0;
  MPI_Status status;
  MPI_Probe(1, ARRAY_TAG, comm, &status);
  MPI_Get_count(&status, MPI_INT, &size);
  int* buffer = new int[size];
  MPI_Recv(buffer, size, MPI_INT, 1, DATA_TAG, comm, &status);
```

```
16 }
```

Listing 2.12 – MPI probe and get count calls.

The MPI functions presented so far are blocking. There are many other functions that specify runtime behavior, and blocking/nonblocking operation. Specifically, there are many communication modes for point-to-point communications: standard, buffered, synchronous, and ready.

- **Buffered**: In this communication mode, the send operation can start whether or not a matching receive has been called on the receiver process. It can also complete before the receive call (it does not wait for the receiver to call a matching receive). The data sent is *buffered* if a receive has not been called. This function may fail if there is not enough buffer space;
- **Synchronous**: The send call only completes when a matching receive starts receiving the message;
- **Ready**: The send call can *only* be started if a matching receive had *already been called*. Otherwise, the behavior of the call is *undefined*;
- **Standard**: Behavior depends on the MPI implementation. It is up to the implementation to choose whether a call is buffered or not.

There are variants of `MPI_Send` for each communication mode. Each communication mode adds a prefix (B, S, or R) to the name:

- `MPI_Send`: No prefix, standard mode;
- `MPI_Bsend`: Buffered mode;
- `MPI_Ssend`: Synchronous mode;
- `MPI_Rsend`: Ready mode;

Additionally, MPI specifies nonblocking operations. In this case, the call always returns immediately. An example of such call is `MPI_Isend`. The parameters of the call are the same as `MPI_Send`, but it also returns a `MPI_Request` object. This object can be used in `MPI_Wait` to wait for the operation to complete when necessary. This allows the programmer to overlap communication and computation. Furthermore, nonblocking calls can also specify the communication mode. For instance, `MPI_Issend` specifies non-blocking (call returns immediately) but synchronous (will only finish when the receiver starts actually receiving the data).

```
#define ARRAY_TAG 999
MPI_Comm comm = MPI_COMM_WORLD;
if (rank == 0) {
  int size = random_between(1, 1000);
  MPI_Request req;
  int* buffer = new int[size];
  MPI_Isend(buffer, size, MPI_INT, 1, ARRAY_TAG, comm, &req);
  doOtherComputation();
  MPI_Wait(req);
  delete[] buffer;
}
...
```

Listing 2.13 – MPI nonblocking calls.

The example in Listing 2.13 shows how overlapping communication and computation is supported by the MPI standard. If the programmer wants to check whether a request has finished already without waiting, `MPI_Test` can be used. This strategy could improve performance when it is possible to overlap MPI communication and computation. However, [DT16] shows that many MPI implementations have poor overlapping. Therefore, the programmer must test whether nonblocking gives a performance gain for their particular application and MPI implementation.

In addition to point-to-point operations, there are also *collective* operations that require all ranks in a communicator to participate in the MPI call. A simple example of a collective operation is `MPI_Barrier(comm)`, which synchronizes all process. Examples of other collective operations are:

- `MPI_Bcast(data, count, type, root, comm)`: Broadcasts a value to all processes in the communicator. The parameter `root` is the source rank that has the data to broadcast. All processes must call root with the same value if the communicator is an intra-communicator. We omit instructions for intercommunicators for brevity.
- `MPI_Scatter` and `MPI_Gather`: For brevity, we do not show the parameters of these functions. `MPI_Scatter` allows to send chunks of an array to different processes. `MPI_Gather` allows to collect arrays from other processes and join them in one of the processes. Additionally, `MPI_Allgather` does the same as `MPI_Gather`, but all processes will have a copy of the entire gathered array.
- `MPI_Reduce`: Allows many processes to participate in a reduce operation. This functions works in the same way as the previous ones. The root rank will receive the result of the operation. An example of reduce operation is multiplying all numbers in arrays distributed over many ranks.
- Management functions: Functions can be used to manage communicators, process groups, and spawning new processes dynamically. Some examples are `MPI_Comm_`-

`spawn` to create new processes, `MPI_Comm_split` to split a communicator into smaller communicators, and `MPI_Intercomm_merge` to merge inter-communicators into a new intra-communicator. For instance: the `MPI_Comm_spawn` returns an inter-communicator for the spawned processes. They can be merged into a single communicator by having the original process and the newly spawned process call `MPI_Intercomm_merge`.

Stream processing in MPI has been studied before [PMG+17, MMP10, WR09, MnDdRA+18]. We will detail these studies in Section 4.

MPI is not a high-level library for parallelism. Although it abstracts some network details, the programmer must be aware of the costs of sending and receiving data to other clusters. Serialization is done by sending buffers using pointers, types and buffer lengths. MPI therefore only specifies a generic message passing interface. All other details must be taken care by the programmer itself.

2.9.2 HPX

HPX [KHAL+14] is a C++ parallel runtime system that extends the C++11 and C++14 standards to support fine-grained and distributed-memory efficient parallelism exploitation. HPX focuses on providing a runtime system ready for exascale computing. The work identifies 4 key issues that limits the scalability of any application. They use the appropriate acronym **SLOW**:

- **S**tarvation: concurrent work is insufficient to fully utilize computing resources;
- **L**atency: Intrinsic delays in remote resource access;
- **O**verheads: Sequential work does not need the overhead of parallel resource management;
- **W**aiting for contention resolution: This causes delays due to oversubscribed shared resources.

The work proposes a solution for these 4 issues using a combination of modules: LCO (Local Control Objects), AGAS (Active Global Address Space), a threading subsystem, and the Parcel subsystem. The Parcel subsystem is responsible for network communication between *localities*. A locality in HPX is an entity with bounded and finite latencies. An example of locality is a compute node in a cluster. Localities communicate via parcels, which encapsulate remote method calls. Parcels contain the ID of the object in which to perform the remote call, the method to call, arguments, and an optional continuation (which is called with the result of the method call).

The object ID is a global ID of an object, possibly located in another locality. To support this functionality, HPX implements AGAS (Active Global Address Space). AGAS implements a dynamic and adaptive address space, which allows HPX to move data between localities without changing its global ID.

When a locality receives a parcel message, HPX executes the appropriate method call in the threading subsystem. This subsystem implements fine-grained task-based parallelism, where each message becomes an *HPX-thread*. An HPX-thread is a lightweight thread that does not require kernel calls to implement context switching. A single CPU core can execute millions of threads per second (assuming they perform no work). Additionally, HPX adds the lightweight thread to a work queue, and executes them using work-stealing to improve load balancing.

In comparison to MPI, HPX focuses on “carrying the computation to the data”, instead of “carrying data to the computation”. In MPI, the common procedure is to pass messages with the data that needs to be computed to other processes. In HPX, the parcel message tells the locality which method should run. This approach seeks to reduce the network usage [KHAL+14]. HPX can also run on top of MPI, which can be used as a fast networking layer between localities.

The programmer interacts with HPX using LCOs (Local Control Objects). There are several types of LCOs:

- Futures: Proxies for results that are not yet available. If the user requests the data when the future LCO is not yet completed, the HPX-thread is suspended until data becomes available. Suspended threads are also considered as LCOs themselves.
- Dataflow: Operations on sets of futures. Manages data dependencies without global synchronization. Dataflow LCOs allow the programmer to specify a function to execute when a set of futures become ready.
- Traditional concurrency control mechanisms: HPX implements mutex, semaphores, condition variables and barriers. HPX also follows the C++14 standard, and implements these mechanisms in compliance with the standard.

The runtime of HPX is asynchronous in nature, and the APIs reflect this model. There are 3 ways of calling functions: synchronously, asynchronously and *fire&forget*. Synchronous calls are similar to plain C++ function calls, where the caller waits for the function to complete. In HPX, synchronous calls cause the caller HPX-thread to suspend. Asynchronous calls do not suspend the caller thread. Instead, the call to an asynchronous function returns a future. Finally, *fire&forget* simply schedules a computation into an HPX-thread. This call does not return values to the caller, and may need less communication since the caller does not need to be notified about the completion of the future.

Using HPX dataflow LCOs, the programmer can to some extent, write code that looks sequential, but executes within the runtime of HPX. With dataflow LCOs, HPX understands the dependencies between futures, and compute them in parallel when possible. Listing 2.14 presents a sequential and a dataflow version of compound interest calculation[1]. This example is not parallel, but it shows how dataflow works. The `for` loop sets up the dataflow execution graph, it does not calculate. Notice that in the HPX dataflow version, `interest` and `principal` are futures, but the `add` and `calc` are plain C++ functions. The `unwrapping` function allows plain C++ functions to be called with future arguments.

```
1  double add(double principal, double interest) {
2      return principal + interest;
3  }
4
5  double calc(double principal, double rate) {
6      return principal * rate;
7  }
8
9  //Sequential version
10 int t = 5;    // number of time periods to use
11 double principal = init_principal;
12 double rate = init_rate;
13 for (int i = 0; i < t; ++i) {
14      double interest = calc(principal, rate);
15      principal = add(principal, interest);
16 }
17
18 //HPX dataflow version
19 shared_future<double> principal = make_ready_future(init_principal);
20 shared_future<double> rate = make_ready_future(init_rate);
21
22 for (int i = 0; i < t; ++i) {
23     shared_future<double> interest = dataflow(
24         unwrapping(calc), principal, rate);
25     principal = dataflow(unwrapping(add), principal, interest);
26 }
27
28 // wait for the dataflow execution graph to be finished
29 double result = principal.get();
```

Listing 2.14 – HPX Local Control Objects. Adapted from HPX documentation.

However, to efficiently distribute a computation over many localities, programmers must be aware of serialization and networking. A complete example of a 1D stencil is available on HPX documentation website[2]. We do not include the examples here for brevity. An efficiently distributed application in HPX is concerned with load balancing, reducing network communications, serialization, and efficiently performing the computation without blocking. In that sense, HPX is not a high-level abstraction for parallel programming.

[1] Adapted form the documentation website in `https://hpx-docs.stellar-group.org/latest/html/examples/interest_calculator.html#examples-interest-calculator`

[2] Available at `https://hpx-docs.stellar-group.org/latest/html/examples/1d_stencil.html`

To send and receive data to other localities, the programmer must implement components or actions. Each component has one or more actions, which should be defined using macros such as `HPX_REGISTER_COMPONENT` and `HPX_REGISTER_ACTION`. HPX's documentation often uses a client/server model in the example applications. Components are servers, and can be accessed with a client. HPX provides abstractions to define a client for a component (server).

In HPX, it is possible to execute programs using the same model as MPI. In MPI, all processes execute the same binary, and the programmer often needs to check the current process rank, and run logic based on it. The flow of the program must be coordinated based on the ranks. It is also possible to execute multiple binaries. However, by default, the HPX process launched in the command line controls the entire program flow while other localities provide the computing resources via components, AGAS, and the parcel system. The programmer must explicitly define which locality will be used in a component/action call.

[KHAL+14] reports good performance results, being consistently faster than MPI and OpenMP state-of-the-art benchmark programs. Benchmarks show HPX is able to achieve 98% of the theoretical peak performance on a single node, and 87% on a distributed workload.

To the best of our knowledge, there are no studies about stream processing with HPX. There is an online repository called HpxFlow[3] which seems to implement dataflow map/reduce patterns, but has not been actively maintained. It does not have a scientific publication, and only a single example of the API.

2.10 Summary

In this chapter, we reviewed tools for distributed stream processing. Table 2.1 shows a summary of this section. Our goal in this table is to compare distributed-memory programming tools that were designed for or can be used for implementing parallel stream processing applications. The guarantees marked with an asterisk * can be upgraded to exactly-once by using a persistent commit-log data source such as Apache Kafka. The majority of the reviewed tools are Java-based, which is one of the main languages used in businesses applications. They provide similar declarative and functional-based APIs to define stream computations. There are tools that offer topological and graph-based APIs. There are also tools that implement micro-batch streaming, such as Spark. In batch streaming, data is buffered until a trigger is activated (e.g. Spark by default buffers 200ms of data). Other tools such as Apache Flink, implement real-time steaming, which does not need to buffer items.

[3] Available in `https://github.com/STEllAR-GROUP/hpxflow`

Most tools discussed in Section 2.8 implement at-least-once guarantees. There are tools that claim to support exactly-once guarantees, often relying on Apache Kafka as a streaming channel that can be "rewinded" and "replayed". Most tools that support at-least-once guarantees recommend users to implement *idempotent operators*. These operators should be able to receive the same message multiple times without changing the result.

Serialization in the JVM can be done automatically due to runtime type information (also known as *reflection*). Frameworks have default serialization methods, frequently using external libraries such as Protobuf and Kyro. Other frameworks support JSON serialization. NiFi and Samza lets the user define serialization and by default support a wide range of formats. Specifically, NiFi lets the user define the *mime-type* of the data (such as `application/json`) on the graph node. Kafka offers its own serialization abstraction, but allows the user to override it. On the other hand, MPI does not offer serialization abstractions. HPX uses Boost serialization.

Back-pressure is implemented in different ways. Flink, Storm and NiFi implement message buffers. If the buffer is full, the stream operator stops and waits until the buffer space is available. Other frameworks such as Akka Streams use demand signals, where the consumers send notifications to the producers when they are ready to receive messages. MPI and HPX do not abstract back-pressure.

Regarding fault-tolerance, each framework has a different implementation. Akka uses the actor supervision model to restart actors when they fail as well as implements basic retry strategies. Other frameworks implement more robust strategies, saving the messages to permanent storage with write-ahead logs in Kafka, checkpoints in Flink, and channels in Samza. Open MPI implements User-Level Fault Mitigation (ULFM) to detect failures. The programmer has to perform recovery manually. HPX does not offer any mechanisms for fault-tolerance besides standard C++ exceptions, which also have to be handled by the programmer.

Meanwhile, most tools execute in the JVM that has a garbage collector, which may impact application performance [Mis17, BAJ+16] or kill the JVM due to GC pressure [Hü15]. This inspired the creation of C++ big-data frameworks, which will be discussed in Section 4.2. However, frameworks such as Apache Flink implemented techniques to avoid the GC (Garbage Collector) by managing memory manually [Hü15]. Therefore, even when running in the JVM, it can achieve good performance with low resource utilization by significantly reducing GC time. We have not found studies that compare this technique with a C or C++ implementation.

We cover HPX and MPI, which are libraries for distributed-memory programming. Both use different approaches: MPI is a simpler message-passing model while HPX implements a global address space, a message subsystem, and re-implement the C++ standard library to work with its threading subsystem. While stream processing libraries have been proposed for MPI, we could not find any well documented or well maintained library. This is

in contrast with HPX itself, which is actively maintained and has documentation, examples, and tests.

Table 2.1 – Distributed-memory programming tools that were designed for or can be used for implementing parallel stream processing applications.

Name	API	Serialization	Language	Back-pressure	Fault-tolerance	Guarantees
Akka Streams	Topological & Declarative	Protobuf, JSON	Scala	Yes, demand signals	Actor supervision, retry strategies	None
Akka Streams .NET	Topological & Declarative	Protobuf, JSON, Hyperion	C#	Yes, demand signals	Actor supervision, retry strategies	None
Apache Flink	Declarative	Own method + Kyro	Java	Yes, buffer space	Checkpoints	At-least-once*
Apache Storm	Topological & Declarative	Thrift and JSON	Java, Clojure	Yes, buffer space	Spout message replay	At-least-once*
Apache Spark	Declarative	Java or Kyro	Scala	Yes, by measuring processing times	RDD keeps the history of operations	At-least-once
Apache Heron	Declarative	Protobuf	Java and C++	Yes, spout slowdown	Spout message replay	At-least-once
Apache Samza	Declarative	Customizable	Java	No, uses Kafka queues as a buffer	State changelogs, host-affinity recovery	At-least-once*
Apache NiFi	Visual	Defined on graph node	Java	Yes, buffer sizes, data item expiration	Write-ahead log	At-least-once
Apache Kafka	Declarative	Own method	Scala	No, Kafka queues	Consumer offsets	Exactly-once
MPI Impls (MPICH, Open MPI)	MPI Standard	None	C	N/A	User-level fault mitigation	None
HPX	HPX LCOs	Boost	C++	N/A	None	None

3. SPAR: STREAM PARALLELISM DOMAIN-SPECIFIC LANGUAGE

In this section, we introduce SPAR (acronym for Stream Parallelism), a domain-specific language compatible with C++ for structured stream parallelism [Gri16]. The language addresses limitations found in other DSLs regarding productivity and performance by providing an annotations-based syntax to define streams, and ultimately generating parallel code with FastFlow for shared-memory architectures. The approach of using DSLs for stream parallelism is not new. In particular, the StreamIt language [TKG+01] provides a productive syntax. However, it requires the user to learn an entirely new language dedicated to stream processing, which is an obstacle for introducing stream parallelism in existing code. Meanwhile, major C++ compilers include tools such as OpenMP [DM98], which allows the user to parallelize code using data parallelism. However, support for OpenMP needs to be implemented in the compiler itself, and stream parallelism on OpenMP is challenging.

C++11 brought support for attributes. For instance: if a method or class is deprecated, the programmer can mark the method with the `[[deprecated]]` attribute, and optionally specify a reason for the deprecation. Users of this method will see a deprecation warning when compiling the program. The list of supported attributes in major C++ compilers is limited, but they are part of the C++ syntax. Each compiler may support different attributes. Any C++11 compliant parser needs to deal with attributes and include it in a syntax tree. SPAR relies on C++ attributes. This is inspired by the REPARA project [DGS+16], which describes the use of C++ attributes and code transformation rules for expressing parallel patterns. SPAR implements a new set of attributes friendly to the stream processing domain. It allows application programmer to tag a region of the code indicating a intensive stream processing region (by annotating with the `ToStream` attribute) along with sequential code wrappers indicating the stream operators/stages (by annotating with the `Stage` attribute). SPAR offers also auxiliary attributes to declare the input and output data dependencies (`Input` and `Output`) as well as the degree of parallelism of a given stateless operator by using the `Replicate` attribute. Therefore, SPAR does not add different syntax to C++, which makes it suitable to be used in existing code that compiles with C++11. It also makes SPAR easier to extend, since the work of parsing the attributes is already done and they are fully represented in the syntax tree. The approach taken by OpenMP may require additional work in the parsing phase since it uses C/C++ directives which are not fully represented in the syntax tree (i.e. they are unparsed `#pragma` strings).

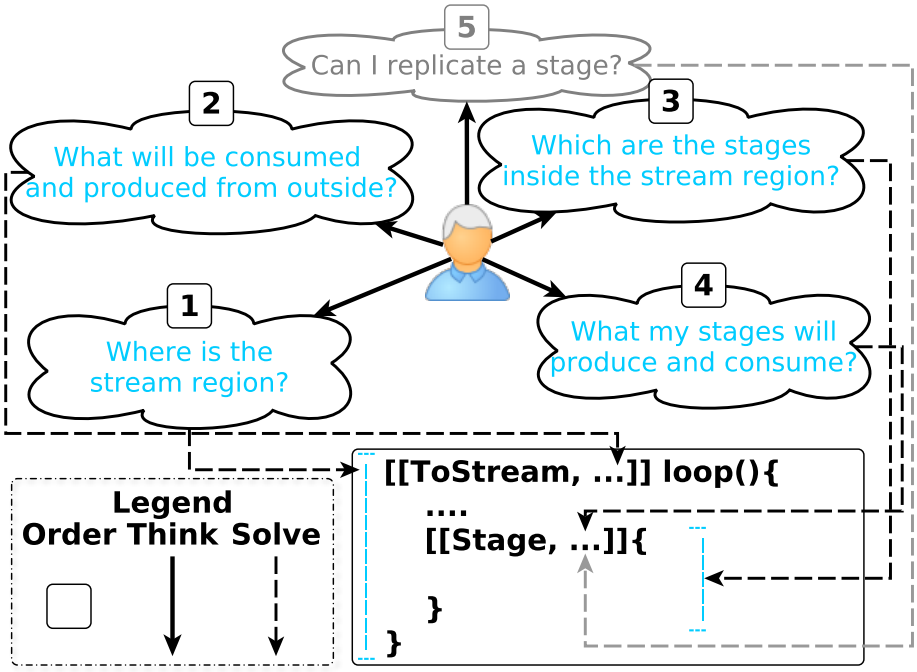

Figure 3.1 – SPAR methodology. Extracted from [Gri16].

3.1 SPAR Annotations

SPAR has an annotation methodology to guide application developer through 5 questions. Figure 3.1 contains the questions and the result when answering these questions. The next paragraphs explain each of the questions:

1. **Where is the stream region?** In general, a stream computation starts by producing stream items to be processed by subsequent stages. This is normally a `for` or `while` loop. The `[[spar::ToStream]]` can be placed before any region that must be parallelized.

2. **What will be consumed and produced from outside?** The programmer must identify the input and output data in form of variables. If the stream uses data parameters outside of the stream area, they must be passed using `spar::Input` annotation. Likewise, if the stream produces a value, they can be declared in the `spar::Output` annotation. For instance: `[[spar::ToStream, spar::Input(param), spar::Output(result)]]`.

3. **Which are the stages inside the stream region?** The programmer has to identify which are the code wrappers that can run independently as separated stages. A common use case is decoupling network or disk I/O from computation, allowing to read data while doing CPU-bound work. Each stage is tagged with `[[spar::Stage]]`.

4. **What my stages will produce and consume?** The programmer must identify the input and output data in form of variables for each stage. This is because each stage

produces data for the next stage. These dependencies must be specified with annotations similar to `[[spar::Stage, spar::Input(x), spar::Output(y)]]`

5. **Can I replicate a stage?** If a stage contains no state, it can be replicated for increasing the degree of parallelism. The stage can easily be replicated with `spar::Replicate(n)` where *n* is the number of Worker replicas.

Listing 3.1 shows a toy example when using SPAR for expressing stream parallelism. The annotation `[[spar::ToStream]]` is added in the beginning of the stream processing region before the `while` loop. Stream elements are read in a loop until the stream ends. The first stage has 4 annotations: `spar::Stage` to indicate that the block is a stage; `spar::Input(stream_element)` means that the stage takes `stream_element` as input. Likewise, `spar::Output(stream_element)` means that the stage emits `stream_element` to the next step. In this example, we multiply the stream items by 2. If this is an expensive stateless operation, it can be replicated and run in parallel. This is done with `spar::Replicated(10)`. SPAR generates a farm with 10 Workers. The last stage prints the results.

```
1  [[spar::ToStream]] while(1) {
2      int stream_element;
3      if (!read_in(&stream_element)) { break; }
4      [[spar::Stage,
5          spar::Input(stream_element),
6          spar::Output(stream_element),
7          spar::Replicate(10)]] {
8          stream_element *= 2;
9      }
10     [[spar::Stage,spar::Input(stream_element)]] {
11         std::cout << stream_element << std::endl;
12     }
13 }
```

Listing 3.1 – Example of stream parallelism with SPAR. Adapted from [Gri16].

3.2 SPAR transformation rules

SPAR uses the annotation information to transform the sequential code into parallel code. Based on the annotations, SPAR chooses the appropriate parallel pattern. It can be a Farm, Pipeline, or a composition of Pipeline with Farms. This is done using code transformation rules. The rules are implemented inside the compiler so that they are completely abstracted from the programmer. The following notation were created to describe the rules:

- T_{id} represents a region of the code marked with the ToStream attribute, where *id* is a numeric identifier.

- S_{id} represents a stage, which is a subsection of the code that can run in parallel in relation to other stages.
- I_i is an auxiliary annotation for the `Stage` (S_{id}) attribute. It declares the input variables of the stage and is equivalent to the `Input` annotation. The attribute may contain a list of variables to define the input variables in the code.
- O_i is an auxiliary annotation for the `Stage` (S_{id}) attribute. It declares the output variables of the stage. The attribute may contain a list of variables to define the output variables in the code.
- R_n is an auxiliary annotation for the `Stage` (S_{id}) attribute. It declares the number of independent replicas of the stage.
- $\square_n$ represents a generic block of code.
- [[...]] represents a list of attributes such as T_{id}, S_{id}, I_i, O_i and R_n. This is similar to the actual C++ attribute list syntax.
- $\{...\}$ denotes the scope of the attributes.

Given these definitions, a set of rules are applied to the source code to instantiate farm and pipeline parallel patterns. A pipeline can be defined as *pipeline*($S_1, S_2, ..., S_n$) with two or more stages. A farm can be defined as *farm*(E, W, C) where E is an Emitter stage, W is a Worker that can be replicated, and C is a Collector stage to gather the results. Definitions for transformation rules can be found in Table 3.1.

Table 3.1 – Definitions for transformation rules. Extracted from [GDTF17].

$D0$	A generic stage ψ is a $\square$ annotated with S that contains in its attribute list R_n and O_i and therefore requiring a further $\square$ gathering its results.
$D1$	A $\square$ may appear as a *pipe* stage or as an E or C stage in a farm if its annotation list S does not contain the attribute R_n.
$D2$	A $\square$ with an annotation list S containing an R_n attribute may only appear as a W stage in a farm.
$D3$	A T is a farm when the first S annotation contains R_n in the attribute list of a maximum two S.
$D4$	A T is a pipe when the first S does not have R_n in the attribute list or when there are more than two Ss
$D5$	A farm is a stage of pipe when $D3$ cannot be applied and $\square$ is annotated with S that contains R_n in the attribute list.

We present 5 of the main transformation rules of SPAR. There are more rules defined in [Gri16].

Rule 1 in Equation 3.1 will transform the given annotation schema into a Farm by applying definitions $D1$, $D2$ and $D3$. When $D1$ is applied, $\square_0$ becomes an Emitter stage

because it is not replicated (R_n). $D2$ implies $\square_1$ must be a Worker node. $D3$ implies that T_0 is a farm, since the first $\square$ with an S annotation has the R attribute.

$$[[T_0]]\{\square_0, [[S_0, R_n]]\{\square_1\}\} \Rightarrow farm(E(\square_0), W(\square_1)) \tag{3.1}$$

Rule 2 in Equation 3.2 generates a generic block $C(\Psi)$. From definition $D0$, there must be a $\square$ if a block annotated with S is also annotated with O_i.

$$[[T_0]]\{\square_0, [[S_0, O_i, R_n]]\{\square_1\}\} \Rightarrow farm(E(\square_0), W(\square_1), C(\Psi)) \tag{3.2}$$

Rule 3 in Equation 3.3 generates the Collector C of the Farm from a $\square$ instead of being an automatically generated generic block $C(\Psi)$.

$$[[T_0]]\{\square_0, [[S_0, R_n]]\{\square_1\}, [[S_1]]\{\square_2\}\} \Rightarrow farm(E(\square_0), W(\square_1), C(\square_2)) \tag{3.3}$$

Rule 4 in Equation 3.4 generates a Pipeline instead of a Farm when there is no R in the first S. This rule is applied due to definition $D4$.

$$[[T_0]]\{\square_0, [[S_0]]\{\square_1\}\} \Rightarrow pipe(\square_0, \square_1) \tag{3.4}$$

Finally, rule 5 in Equation 3.5 allows composition of Pipeline and Farm. In this case, $D3$ cannot be applied because R is in the second block annotated with S instead of the first. Instead, definition $D5$ is applied.

$$[[T_0]]\{\square_0, [[S_0]]\{\square_1\}, [[S_1, R_n]]\{\square_2\}\} \Rightarrow pipe(\square_0, farm(E(\square_1), W(\square_2))) \tag{3.5}$$

Listing 3.1 shows a minimal example of a SPAR stream. The annotation `[[spar::ToStream]]` is added in the beginning of the stream before the while loop. Items are then read and passed to the first `[[spar::Stage]]` where they are computed using a pipeline-farm (notice the `spar::Replicated(10)` annotation).

3.3 SPAR Compiler & Runtime

At the time SPAR was created, a C++ parser was needed and tools such as GCC and Clang were tested to provide the syntax tree. SPAR needs to perform aggressive code transformations based on a syntax tree that accurately represents the code. GCC and Clang would provide a more abstract syntax tree with different semantics, making them unsuitable for use in SPAR. This prompted the creation of CINCLE (Compiler Infrastructure for New C/C++ Language Extensions), which can provide a C++11 syntax tree and allows source-to-source transformations directly in the AST.

Figure 3.2 illustrates the CINCLE infrastructure. First, the front-end calls GCC to check the code. This means that *the annotated code must be valid C++* before the code transformations. If the code is valid, the AST is produced using Flex and Bison. The middle-end and back-end analyze the AST and transform the code. For instance: SPAR analyzes the annotations, and based on the annotations, generates code using transformation rules. Finally, the transformed code is compiled with GCC.

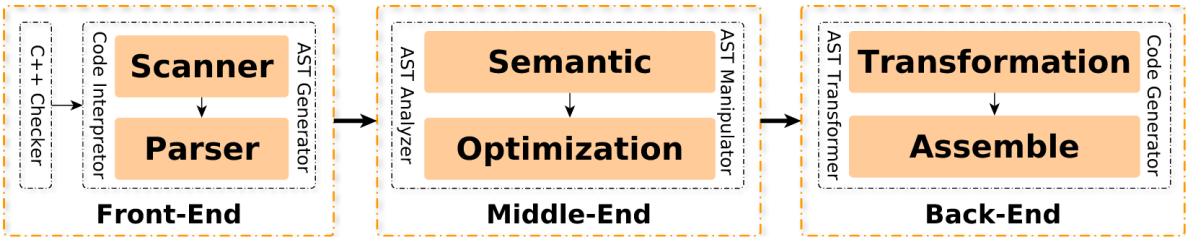

Figure 3.2 – CINCLE Infrastructure. Extracted from [Gri16].

Figure 3.3 shows the structure of the SPAR compiler. The Semantic Analysis and AST Transform steps are the logic implemented for SPAR while other steps are implemented by CINCLE.

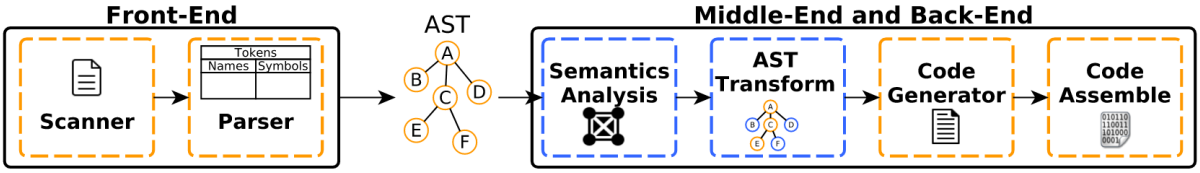

Figure 3.3 – SPAR compiler. Extracted from [GDTF17].

Additionally, SPAR accepts the following compiler flags to control runtime execution characteristics:

- `-spar_ordered`: This option is useful when a stream computation needs to output results in the same order that the stream items were produced. For instance: a video application needs to output the video frames in the same order they were captured by

the camera. This might not be the case in a farm computation, since the Workers might finish computing their items in an unpredictable order. This option is implemented by FastFlow with the `ff_ofarm` class, which reorders the items in the last stage of the farm (Collector).

- `-spar_ondemand`: By default, FastFlow uses non-blocking queues to store stream items received from other stages. The default size is 512. When a queue size reaches 512 items, previous stages stop producing items. This option changes it to 1, which changes the scheduling behavior to act on-demand: only produce items when the next stage finished processing an item.

- `-spar_blocking`: By default, FastFlow uses active pushing and popping from communication queues to achieve low latency. However, this might use significant CPU resources. This option changes it to a non-active, blocking behavior which uses fewer CPU resources, but might increase latency.

In general, annotation-based stream parallelism has been focused on shared-memory architectures. So far, this is the case of SPAR as well. [GF17] initiated efforts to implement an MPI backend for SPAR, but so far SPAR does not support MPI or any form of distributed computing. While FastFlow, the main backend for SPAR, has support for distributed computing, [Gri16] encountered several limitations using it for distributed stream processing. Regardless of any FastFlow shortcomings in this area, [WR09] argues that an MPI implementation is desirable in the HPC context because several numerical libraries already use MPI to parallelize the computation. Furthermore, due to the popularity of MPI, we expect MPI would be installed in most HPC clusters. By using MPI, we can increase the scalability that is limited to a single host with FastFlow in SPAR.

The results of [GF17] towards supporting distributed streaming on SPAR shows that SPAR increases productivity with negligible performance loss. The researchers used the FastFlow code generated by SPAR and rewrote it manually to use MPI instead of FastFlow. A proper backend for MPI in SPAR does not exist. However, many problems still need to be solved to provide a complete solution. Namely, sizes of dynamic arrays need to be informed to SPAR, but currently, there is no syntax for it. Serialization and deserialization of arbitrary data types is also a significant challenge. More importantly, this previous work also argues that a library for parallel streaming patterns is necessary, due to the complexity of generating code for MPI directly. A library that abstracts the details of MPI while having an API similar to FastFlow can make the code generation phase easier, reliable, and compatible with the pattern-based transformation rules proposed in the Griebler's thesis [Gri16].

4. RELATED WORK

In our related work, we include parallel programming APIs for stream parallelism in C++. We divide this chapter in 2 sections: Section 4.1 introduces C++ libraries for stream parallelism, while Section 4.4 introduces domain-specific languages and high-level programming interfaces for stream parallelism.

4.1 Runtime Libraries

This section shows libraries for stream parallelism in C++. We describe each library with a brief example. We also discuss serialization strategies, networking and the programming interface.

4.1.1 FastFlow

FastFlow [ADKT17] is a high-level C++ patterns-based parallel programming library. The architecture of FastFlow is conceptually composed of 3 layers: building blocks such as queues and nodes, core parallel design patterns such as Pipeline and Farm, and high-level patterns such as `parallel_for` and `parallel_reduce`. FastFlow components communicate via non-blocking queues. Its communication is designed to have low overhead and has good performance for fine-grain parallelism. The framework architecture can be seen in Figure 4.1.

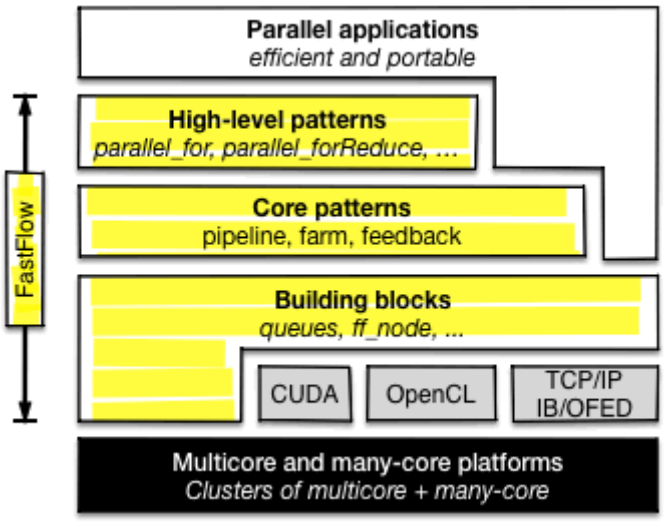

Figure 4.1 – FastFlow architecture [Fas19].

FastFlow abstracts parallelism details using the algorithmic skeletons approach [Col89]. The programmer uses an abstraction that implements parallelism efficiently. FastFlow is also designed to support heterogeneous computing, possibly running algorithms on clusters with computing accelerators.

Regarding distributed computing, FastFlow has support for multi-node setups using ZeroMQ as the transport layer, and a specific abstraction to define distributed nodes [ACD+12]. For shared-memory parallelism, FastFlow offers the Pipeline and `ff_node` API, in which the programmer defines an operation to be applied to a stream item. Nodes can be connected to form any streaming topology in the form of a graph. For distributed-memory architectures, the `ff_dnode` is available, which contains the a similar API as `ff_node` with extra methods for serialization.

```
1  //sender
2  void* svc(void* task) {
3      char* s1 = new char[12 + 1];
4      strncpy(s1, "hello world!", 12 + 1);
5      mystring_t* s = new mystring_t(12, s1);
6      return s;
7  }
8  void prepare(svector<iovec>& v, void* ptr, const int) {
9      mystring_t* p = static_cast<mystring_t*>(ptr);
10     struct iovec iov = {ptr, sizeof(mystring_t)};
11     v.push_back(iov);
12     iov.iov_base = p->str;
13     iov.iov_len = p->length + 1;
14     v.push_back(iov);
15 }
16
17 //receiver
18 void* svc(void* task) {
19     mystring_t* s = (mystring_t*)task;
20     printg("Received %s\n", s->str);
21     return task;
22 }
23 void unmarshalling(svector<msg_t*>* const v[], const int vlen, void
       *& task) {
24     mystring_t* p = static_cast<mystring_t*>(v[0]->operator[](0)->
           getData());
25     p->str = static_cast<char*>(v[0]->operator[](1)->getData());
26     assert(strlen(p->str) == p->length);
27     task=p; //task will be passed to the svc method
28 }
```

Listing 4.1 – FastFlow serialization.

Listing 4.1 shows how serialization is done in FastFlow[1]. The serialization in FastFlow is not abstracted from the application programmer. The methods `prepare` and `unmarshalling` need to be implemented. The `prepare` method transforms the data into a set of

[1] `http://calvados.di.unipi.it/storage/talks/2012_dFF_CGS.pdf`

messages using `iovec`. The same data can be deserialized in the `unmarshalling` method. The user also needs to cast the data to the correct type manually, it is not a type-safe API.

The support for distributed computing in FastFlow is still evolving. There is no documentation discussing about delivery guarantees and back-pressure mechanisms. There is also no written documentation about the distributed API, only code examples. FastFlow uses ZeroMQ for distributed communication, which by itself has no delivery guarantees and no back-pressure mechanism. We describe FastFlow further in Chapter 5.

4.1.2 GRPPI

GRPPI (Generic Reusable Parallel Pattern Interface) [dRADFG17] is a C++ library that implements composable and generic interfaces for parallel patterns. Given the number of parallelism libraries currently available such as TBB, FastFlow, and Open MPI, no standard or implementation covers all of the parallel patterns, according to [dRADFG17]. Specifically, GRPPI implements the patterns Pipeline, Farm, Filter, Accumulator, Map, Reduce, Stencil and Divide & Conquer. The back-end of GRPPI can be chosen by the user. Currently, GRPPI supports C++ threads, OpenMP, Intel TBB, FastFlow, and CUDA Thrust.

The GRPPI API makes extensive use of C++ templates and C++11 lambda functions. Patterns can be composed together seamlessly since patterns themselves are implemented as functions. From the programmer point of view, minimal code refactorings are needed, and different back-ends can be used in the same computation, taking advantage of different hardware capabilities when possible.

Listing 4.2 shows an example of GrPPI's API. A Pipeline is composed with a Farm pattern, which in turn transforms each character coming from a string stream to its uppercase form. The internal Map is executed sequentially, while Pipeline and Map are executed by a back-end chosen by the user.

```
1  grppi::pipeline(executor,
2    [&]() -> optional<string> {
3      string word;
4      in >> word;
5      if (in) return word;
6      else return {};
7    },
8    grppi::farm(4,
9      [](auto word) {
10       grppi::sequential_execution seq{};
11       grppi::map(seq, begin(word), end(word),
12           begin(word), [](char c) {
13         return std::toupper(c);
14       });
15       return word;
16     }),
```

```
17      [&](auto word) {
18          out << word << std::endl;
19      });
```

Listing 4.2 – GRPPI example. Adapted from [Gar18].

Additionally, the support for distributed computing was studied by [MnDdRA+18, LGFd+19] using MPI. However, we could not find the source code to understand how serialization is done since the work does not provide a link to the online repository with the MPI code. However, [LGFd+19] mentions that GrPPI uses Boost MPI. We believe this is a just initial prototype work since it is not easy to provide a high-level abstraction to distributed parallel programming.

4.1.3 PiCo

Nowadays, different Java-based Big-Data platforms exist. However, each platform exposes a different API, which leads to a lack of standard in the big data domain. The work of [Mis17] surveyed many commercial platforms such as Apache Spark, Apache Flink, and Apache Storm to create a unified semantics for Big Data analytics frameworks.

By creating a unified model, the work compared multiple Big Data tools with different levels of abstraction (framework API, program semantics dataflow, parallel execution dataflow, process network dataflow and execution platform). One key aspect of the work by [Mis17] is the differentiation between *Topological* and *Declarative* APIs. Furthermore, [Mis17] argues that there is no semantic difference between batching and streaming operations. Any API exposed by these frameworks can be translated into a dataflow graph.

Using the unified model, [Mis17] created a data-model agnostic DSL for Big Data called PiCo (Pipeline Composition), based on polymorphic operators that can be reused with any data type. With this theoretical foundation, a fluent C++14 DSL was developed, which is demonstrated in Listing 4.3. The API exhibits a functional interface based on method chaining, where Pipelines can be composed together. Operations are represented as C++ types that are added to a Pipeline.

```
1  Pipe countWords;
2      countWords
3      .add(FlatMap<std::string, std::string>(tokenizer))
4      .add(Map<std::string, KV>([&](std::string in)
5          {return KV(in,1);}))
6      .add(PReduce<KV>([&](KV v1, KV v2)
7          {return v1+v2;}));
8
9  ReadFromFile reader();
10 WriteToDisk<KV> writer([&](KV in) {
11     return in.to_string();
```

```
12 });
13 /* compose the pipeline */
14 Pipe p2;
15 p2 //the empty pipeline
16  .add(reader) // add single operator
17  .to(countWords) // append a pipeline
18  .add(writer); // add single operator
19 /* execute the pipeline */
20 p2.run();
```

Listing 4.3 – PiCo example. Adapted from [Mis17].

PiCo runs on top of FastFlow. Currently, there is no support for distributed-memory architectures. However, it could be implemented using FastFlow itself or the new GAM (Global Asynchronous Memory) model [Dro17].

4.1.4 Thrill

Thrill [BAJ+16] is an experimental C++14 framework for distributed stream processing. Thrill is similar to Apache Flink and Apache Spark but is implemented in C++ for performance advantages and low-level control of memory structures. Thrill focus on providing an easy way to program distributed algorithms in C++. The building block of the API is the DIA (Distributed Immutable Array). A DIA supports common operators such as Map, Filter, FlatMap, windowing, aggregations and logical partitioning. An example of Thrill's API is shown in Listing 4.4.

```
1  auto word_pairs = ReadLines(ctx, input)
2   .template FlatMap<Pair>(
3      // flatmap lambda: split and emit each word
4      [](const std::string& line, auto emit) {
5          SplitView(line, ' ', [&](StringView sv) {
6              emit(Pair(sv.ToString(), 1));
7          });
8      });
9  word_pairs
10  .ReduceByKey(
11     // key extractor: the word string
12     [](const Pair& p) { return p.first; },
13     // commutative reduction: add counters
14     [](const Pair& a, const Pair& b) {
15         return Pair(a.first, a.second + b.second);
16     })
17  .Map([](const Pair& p) {
18     return p.first + ": "
19     + std::to_string(p.second);
20  })
```

Listing 4.4 – Thrill example. Adapted from [Thr19].

Networking in Thrill is done via TCP sockets and optionally via MPI. Thrill is an experimental effort still in its early stages of development. Currently, no delivery guarantees or fault-tolerance mechanism are implemented.

Thrill deals with serialization using the Cereal library, which provides a lightweight serializer and deserializer with an API similar to Boost archives. Thrill by default implements serializers for numbers, strings, and `std::pair`. Additionally, Thrill implements serialization for types that are *trivial*, which are types that are binary-compatible with C. For instance, a class with 2 integer fields is trivial (also called POD - Plain Old Datatype). For more complex cases, the programmer can use the Cereal library manually.

4.2 MPI-based stream APIs

In the HPC context, MPI remains the *de-facto* parallelism abstraction [MMP10]. Libraries such as Open MPI [GFB+04] offer a robust MPI implementation, taking advantage of fast networks and newer MPI standards. Developers creating distributed algorithms need to understand the MPI standard to efficiently scale computations across multiple nodes.

Additionally, some computations can take advantage of the stream processing paradigm. MPI programs communicate by sending messages to one another. Therefore, it is possible to abstract this communication in a stream-like structure as shown by [PMG+17, WR09, MMP10]. However, the related works presented here attempt to extend the capabilities of MPI towards streaming, without necessarily abstracting low-level MPI details. In this section, we discuss these related works by describing their API, their design goals, and features relevant to parallel streaming such as load balancing and ordering constraints.

4.2.1 Hybrid MPI Streaming

The work of [MMP10] is focused on providing a high-performance MPI abstraction using a hybrid approach. The work is mainly concerned with supporting heterogeneous topologies. [MMP10] supports MPI and TCP sockets. The work also optimizes local communication by using POSIX Threads instead of OS processes. The authors do not give a name for the library. Therefore, we will call it “MPI Hybrid”.

The authors argue that streaming applications can exploit heterogeneous networks. In a streaming programming model, an application is composed of highly independent “kernels”. Kernels have inputs and outputs, like Pipeline stages. Kernels also run in parallel to each other and can be stateful or stateless. This is similar to Pipeline stages described in Section 2.7: stages have inputs and outputs, and run in parallel to each other. In Figure 4.2,

the work shows the use of MPI in a hybrid streaming scenario. An expensive computation is parallelized using a parallel MPI kernel. Communication with other sequential kernels is done via TCP sockets, although the work does not make this clear.

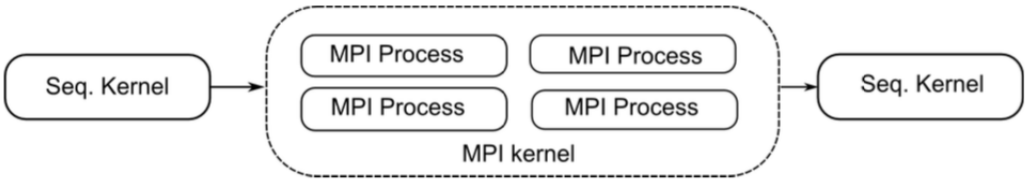

Figure 4.2 – Hybrid MPI streaming model. Extracted from [MMP10].

Readers familiar with structured parallel streaming patterns may recognize the similarity of Figure 4.2 with the Farm pattern. The first sequential kernel acts as an Emitter, the MPI kernel acts as a set of replicated Workers, and the last sequential kernel can be a Collector.

We show an example of the API in Listing 4.5. Each kernel is defined as a function that takes a `osf_KernelContext_t` and returns a `osf_Result_t`. The stream must first be opened with `osf_Open`. To send data to other kernels, the function `osf_Put` is used. No MPI-specific details are required. To receive an item from the stream, `osf_Get` is used.

The programmer can use `osf_RegisterKernel` to configure a kernel with different features. One of the features is the kernel type, which can be “polling” or “callback”. Not much is explained about these options. Based on the name given, we presume that “polling” means the consumer kernel keeps sending polling messages to the producer kernel. The programmer can configure a polling kernel by using the `OSF_KRNTYP_POLLING` flag, as shown in line 20. The callback option is not explained nor used in the code. The authors do not elaborate the load balancing strategies used for either polling or callback modes. Similarly, the stateful/stateless features are also unexplained. For instance, the `OSF_KRN_STATELESS` flag is used in line 20, but the work does not elaborate on the effects of these flags. We presume that there is a `OSF_KRN_STATEFUL` flag as well. After registering the kernels, the stream can be started with `osf_StartStreams()`, and then finalized with `osf_Finalize()`.

```
1  //Source kernel: opens an output stream
2  osf_Result_t SourceKernel(osf_KernelContext_t*ctx) {
3      static osf_Stream_t*s = NULL;
4      if (s==NULL)osf_Open(&s, OSF_STR_OUT, 1);
5      ...record = sin(t)+sin(4+2*t);
6      osf_Put(s, &record, sizeof(record));
7      return OSF_ERR_SUCCESS;
8  }
9  /*Destination kernel: takes the data from source kernel and
       elaborates*/
10 osf_Result_t FilterKernel(osf_KernelContext_t*ctx) {
11     static osf_Stream_t*sIn = NULL;
12     if (sIn==NULL)osf_Open(&sIn, OSF_STR_INPUT, 0);
13     ...res = osf_Get( sIn, &x, sizeof(double),&receive );
```

```
14      return OSF_ERR_SUCCESS;
15  }
16  /*Main function: registers the kernels and starts the streams*/
17  int main (...) {
18  osf_KernelContext_t*kctx
19      osf_Init(...)
20      osf_RegisterKernel(0,OSF_KRNTYP_POLLING, OSF_KRN_STATELESS,
            SourceKernel, &kctx);
21      osf_RegisterKernel(1,OSF_KRNTYP_POLLING, OSF_KRN_STATELESS,
            FilterKernel, &kctx);
22      osf_StartStreams();
23      osf_Finalize();
24  }
```

Listing 4.5 – Hybrid MPI example. Adapted from [MMP10].

Hybrid MPI can work with complex hybrid topologies. However, launching such an application might be complicated. Programmers familiar with MPI might recall that launching an MPI application can be done by simply calling `mpirun` or `mpiexec`. However, if the application needs TCP sockets and may run in multiple clusters (as well as multiple nodes in a cluster), it becomes difficult. Hybrid MPI includes a launcher application that reads a XML file and starts the processes. This XML contains all the necessary information to generate scripts and MPI hostfiles. It is unclear whether the launcher needs to be installed on every node that participates in the computation.

MPI Hybrid was used to parallelize a financial application that simulates the stock market. The application is described in Figure 4.3. The work is randomly generated in a sequential kernel. Then, an MPI application processes the items with a matching engine and an order confirmation module. After processing the order, it is stored on disk by a sequential kernel. The MPI implementation used was MPICH2.

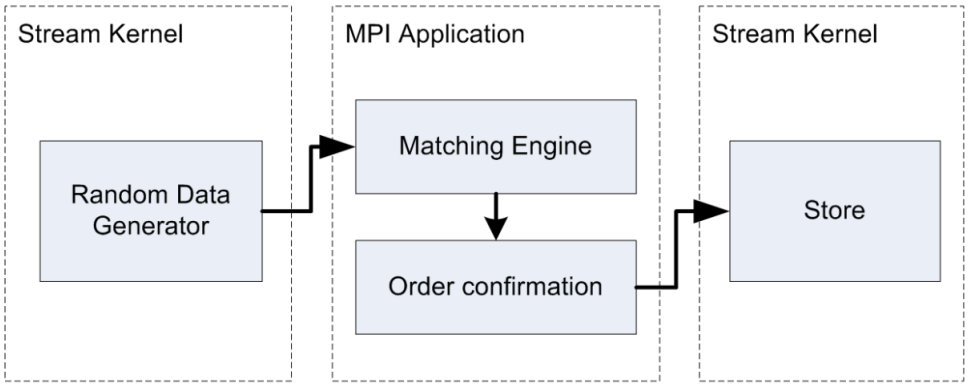

Figure 4.3 – Hybrid MPI evaluation program [MMP10].

The authors achieved an improvement of 32% in latency compared to a MPI-only approach. However, it is unclear whether the MPI application is replicated or not, or how kernels communicate. For instance: how the random data generator communicates with the MPI application? In our understanding, the communication is done with TCP sockets. We

also understand that nothing is replicated, since the matching engine and order confirmation module are stateful. The performance improvement, as argued by the authors, come from avoiding the overhead of MPICH2 by using a simpler socket-based approach. The work does not test an application based on the topology shown in Figure 4.2.

Although some details of the work are still unclear, it provides a method to improve latency by avoiding MPI whenever necessary. However, the work does not explore alternatives such as Open MPI, which implements optimizations for local communication [GFB+04, Ope20][2]. It also does not show the performance characteristics of using POSIX threads, which can further improve performance by avoiding the network entirely. The authors chose a real-world application in which the worker stages are stateful and cannot be replicated. Therefore, it is difficult to test other scenarios, such as the effects of replication. The work also does not say whether the library supports ordering constraints. However, it may support dynamic work scheduling with the `OSF_KRNTYP_POLLING` kernel flag. We could not find a code repository to investigate the questions raised in this section.

4.2.2 Lightweight API for MPI Streaming

[WR09] proposes a lightweight API for MPI stream processing. This work focuses on the correctness of the communication and studies the implementation of a graph-coloring algorithm that automatically creates intra- and inter-communicators. These communicators are created to ensure that a message is not sent to the wrong process. This becomes important when using MPI in different parts of a program at the same time. The authors do not give a name for the library, this way, we will call it “Lightweight Stream MPI”, which is in the title of [WR09]. Listing 4.6 shows an example of the streaming part of a program written with the library. We adapted multiple code listings in [WR09]. The code is an implementation of the Eratosthenes’s Sieve.

```
1  #include <workflow.h>
2  int main(int argc, char*argv[])
3  {
4      environment_t env;
5      getEnvironment(argc,argv,&env);
6      MPI_Init(&argc,&argv);
7      MPIComm comm;
8      workflow_Init(MPI_COMM_WORLD,&env,&comm);
9      int stop = -1, maxnum = 10000, num = 0, prime = 1, first = 1;
10     while (TRUE) {
11         if (env.numin == 0) {
12             // First stage
13             if (num++ < maxnum)
14                 workflow_Send(&num, MPI_INT, comm);
15             else {
```

[2] https://www.open-mpi.org/faq/?category=sm

```
16                  workflow_Send(&stop, MPI_INT, comm);
17                  break;
18              }
19          } else if (env.numout != 0) {
20              // Middle stage
21              workflow_Recv(&num, MPI_INT, comm);
22              if (num == stop) {
23                  workflow_Send(&stop, MPI_INT, comm);
24                  break;
25              } else if (first) {
26                  prime = num;
27                  first = FALSE;
28              } else if (num % prime != 0)
29                  workflow_Send(&num, MPI_INT, comm);
30              else /*not a prime*/
31                  ;
32          } else {
33              // Last stage
34              workflow_Recv(&num, MPI_INT, comm);
35              if (num == stop) break;
36              printf(" number is %d\n", num);
37          }
38      }
39      workflow_Finalize(&comm);
40      MPIFinalize();
41  }
```

Listing 4.6 – Lightweight MPI example adapted from [WR09].

Lightweight Stream MPI abstracts details regarding process communication. First, the programmer must initialize the environment. In this particular scenario, the authors construct the environment using a MPICH `config` file, which specifies the inputs and outputs of each MPI process. The `environment_t env` variable contains the input and output information for the process.

Lightweight Stream MPI provides `workflow_Send` and `workflow_Recv` functions to communicate with other processes. It is unclear whether [WR09] supports Farm patterns. The work does not explain in detail how `workflow_Send` distributes work, but it seems that a process can have multiple output ranks, as suggested by the `numout` field in line 19. However, the text in [WR09] also suggests that `workflow_Send` broadcasts the data item to all the output ranks. This makes implementing a Farm pattern difficult. The work also does not clarify whether any load balancing is done at any point, and does not present any performance analysis.

Lightweight Stream MPI is not a high-level abstraction for streaming applications. `workflow_Send` and `workflow_Recv` still resembles `MPI_send` and `MPI_recv`. Notice that the functions need to receive a type (like `MPI_INT`) and a communicator. Therefore, we assume serialization is done in the same way as MPI. The stream stopping signal also has to be handled manually, as shown in lines 14, 21 and 35.

In our work, we intent to abstract more MPI details, including MPI data types, communicators and stream management such as starting and stopping a stream. [WR09] focuses on reliably integrating a lightweight streaming API into existing applications that already use MPI. Work scheduling, performance, serialization and higher-level abstractions are not the concern of [WR09].

4.2.3 MPI Streams

MPI Streams [PMG+17, PML+15] proposes an extension to the MPI standard to support streaming applications. The work creates a generic streaming model consisting of producers, consumers, and communication channels. MPI Streams isolates producers and consumers using MPI communicators. This allows for safer operation in a larger project with multiple modules using MPI, since processes in a communicator do not interfere with other communicators [WR09].

MPI Streams define 8 new functions. Many details of these functions are specifically made to support Map and Reduce operations. The main functions are:

1. `int MPIStream_CreateChannel(int isProducer, int isConsumer, MPI_Comm comm, MPIStream_Channel* channel)`: Creates a communication channel. This is a collective operation, and the programmer must set the `isProducer` and `isConsumer` parameters accordingly. This allows the programmer to create multi-producer multi-consumer streams. This can facilitate the creation of a Farm pattern by creating a single-producer multi-consumer channel for the Emitter and Workers. The `MPIStream_Channel*` parameter is the resulting channel.

2. `int MPIStream_Attach(MPI_Datatype streamDataType, MPI_Datatype resultDataType, MPI_Datatype processDataType, MPIStream_Operation * operation, MPI_Stream * stream, MPIStream_Channel * channel)`: Attaches an operation in a stream. `MPIStream_Operation` is an object that stores 2 functions: an intermediate function, and a terminal function. The intermediate function executes for each stream item. This function can be used to store local state in the process. The terminal function is called by data consumers when the stream ends in a collective *reduce* operation. This function must be compatible with the `MPI_reduce` operation (such as `MPI_SUM`, or an user defined operation with compatible input and output parameters). The datatype parameters are MPI datatypes, therefore serialization is not abstracted. The parameters `streamDataType` and `resultDataType` specify the inputs and outputs of the operation. The role of `processDataType` is only briefly explained, without much detail. All the code examples in [PML+15] set the 3 datatype parameters to `MPI_INT`. Oddly, the most recent paper about the library [PMG+17] mentions a `MPIStream_Attach` method that

only receives the `streamDataType`. The `MPI_Stream` object contains metadata about the stream structure.

3. `int MPIStream_Send/MPIStream_Isend(void * sendbuf, MPI_Stream * stream)`: Allows the producer to send data to a stream. There is the synchronous and asynchronouos version following the standard of MPI, in which asynchronous functions are prefixed with MPI_I. The data type is set on the `MPIStream_Attach`, therefore it is not needed here.

4. `int MPIStream_Operate/MPIStream_Ioperate(MPI_Stream*stream)`: Runs the intermediate function of the `MPIStream_Operation` object configured in the `MPIStream_Attach` call. Automatically receives and process stream elements.

Listing 4.7 shows an example of the MPI Streams API. First, the programmer must initialize MPI (omitted for brevity) and determine if the process is a data producer or consumer. The channel is created using the `MPIStream_CreateChannel` function. Then, the programmer defines an operation and attach it with `MPIStream_CreateChannel`. Notice that the function `AccumulateOverTime` receives pointers to the stream data with `void* in`, and a textttvoid* inout to store local state. These pointers need to be casted to the correct pointer type. The producer emits 100 stream elements, while the consumers store local state containing the sum of all elements they processed. The terminal operation `MPI_SUM` sums the local state of all consumers. The example does not show how to access this reduced value.

```
1  #include "MPIStream.h"
2  int main(int argc, char **argv) {
3     // .. setup MPI, get nprocs, myrank
4     // two data consumers, the rest are data producers
5     int is_data_producer = myrank < (nprocs - 2);
6     int is_data_consumer = myrank >= (nprocs - 2);
7     // stream initialization stage
8     MPIStream_Channel channel;
9     MPIStream_CreateChannel(is_data_producer, is_data_consumer,
          MPI_COMM_WORLD, &channel);
10    // specify stream operation
11    MPIStream_Operation operation;
12    operation.intermediate_function = &AccumulateOverTime;
13    operation.terminal_function = MPI_SUM;
14    // attach the specified operation to a parallel stream
15    MPIStream stream0;
16    MPIStream_Attach(MPI_INT, MPI_INT, MPI_INT, &operation, &stream0,
          &channel);
17
18    if (is_data_consumer) // data consumers keep on processing data
19       MPIStream_Operate(&stream0);
20    if (is_data_producer) {
21       for (int i = 0; i < 100; i++) // stream out 100 elements
22          MPIStream_Send(&i, &stream0);
23       MPIStream_Terminate(&stream0); // send termination signal
24    }
25    // Finalize
```

```
26     MPIStream_FreeChannel(&channel);
27     MPI_Finalize();
28     return 0;
29 }
30 // user-defined operation on stream
31 void AccumulateOverTime(void *in, void *inout, int *len, MPI_Datatype
       *datatype) {
32     if (*len == 0)
33        *(int *)inout = 0; //initialize the result
34     else if (*len == 1)
35        *((int *)inout) += *((int *)in); //accumulate the result
36  }
```

Listing 4.7 – MPI Streams example. Adapted from [PML+15].

MPI Streams do not abstract serialization. The user must be familiar with MPI_Datatype and use them. Furthermore, the library also does not support ordering constraints [PML+15]. The consumers support multiple producers and process elements using a "first-come first-served" approach. The work does not elaborate on the producer side of load balancing, but we presume it has some dynamic scheduling behavior. It only produces data when necessary. However, each consumer has a buffer of stream elements. The buffer stores up to 5 asynchronous requests and use `MPI_Testany` to receive the data. The work does not elaborate on what happens when the buffer is full, and a producer sends data to it. We presume the producer blocks until an element is processed and buffer space becomes available again. As an optimization, the work uses persistent communication available in MPI 2. This reduces the overhead of point-to-point communications (like `MPI_recv`).

[PML+15] evaluated the performance of MPI Streams, running an extensive set of benchmarks on the Beskow Cray XC40 and Vesta BlueGene/Q supercomputers. With 32 data producers and 32 data consumers, the maximum available network bandwidth on Beskow is 15GB/s, and 7GB/s on Vesta. Using an extensive set of benchmarks, MPI Streams used 65% of the total available network bandwidth of Beskow (9GB/s), and 52% of Vesta (3.5GB/s). For 2,048 producers and consumers, the library achieves 200GB/s on Beskow, and 80GB/s on Vesta. [PMG+17] tested MPI Streams on LHC (Large Hadron Collider) events, and processed 2.2 million events/second with 1000 consumer processes. Additionally, MPI Streams also processed 55 million words/second on a word counting application.

MPI Streams allows the programmer to build streaming applications with a producer-consumer architecture, but it does not clarify whether a Pipeline application with many stages or a whether a Farm pattern is possible. The abstractions provided by MPI Streams are instead focused on the Map/Reduce patterns. MPI Streams also does not abstract MPI details, instead it provides an abstraction for streaming while requiring the programmer to deal with MPI details, such as communicators, process ranks, and serialization. In our work, we intend to abstract these details and provide a high-level stream parallel pattern library, focused

on Pipeline and Farm patterns. We could not find an available online repository with the MPI Streams code.

4.2.4 Skeleton-Based MPI Libraries

In addition to MPI-based libraries that extend MPI to support stream processing applications, other C/++ skeleton libraries have been proposed that include support for MPI. [GVL10] performed an extensive survey on skeleton libraries and we describe some of the libraries here. We also describe their design goals and how they differ from our work. It is especially interesting for us to understand how the libraries perform data serialization in MPI. In general, the libraries presented here are not being developed anymore.

ESkel [Col04] is proposed by the same authors of Algorithmic Skeletons [Col89]. ESkel is a C library that implements the patterns Pipeline, Farm, and Divide-and-Conquer. ESkel is a simple library based on MPI. Users of eSkel must be familiar with basic MPI concepts as well. Serialization is done by using *"eDM"* (eSkel Data Model). This model is very similar to MPI, in which the user must specify a pointer to data, size, and type of data. The difference is that eDM also needs a "spread" argument, which specifies whether the data is local or distributed among many processes. In our work, we intend to abstract serialization and MPI details. The library itself has not changed since 2005. However, the work proposes 4 principles for skeleton libraries:

1. Propagate the concept of skeletons with minimal disruption: eSkel realizes this by implementing skeletons in C rather than in functional languages. Also, skeletons should work well with current technologies, acting as bridges to the current standards.

2. Allow ad-hoc parallelism: Some algorithms are not obviously expressed as instances of skeletons, such as the Cannon's matrix multiplication. Therefore, skeletons should also be allowed alongside ad-hoc parallelism.

3. Accommodate diversity: Sometimes, skeleton libraries impose constraints that make implementing algorithms as skeletons not viable. For instance, a Pipeline skeleton implementation that requires every input to produce an output makes some algorithms difficult to implement if they need multiple outputs per input.

4. Show the pay-back: the library should be easy to learn, and should demonstrate that the benefits outweigh the initial overheads. The library should also outperform the conventional implementation.

SkeTo [MIEH06] is a C++ library that uses the theory of Constructive Algorithmics to instantiate and optimize skeletons. SkeTo supports MPI and applies optimizations to the

patterns using source-to-source transformation with OpenC++ [MKI+04]. The optimizations reduce the number of intermediate data structures passed between skeletons. SkeTo also provides parallel data structures, such as a Parallel List, Tree, and Matrix. SkeTo implements Map and Reduce patterns. [MIEH06] does not provide details regarding serialization. However, we presume serialization is not a particularly difficult issue because the programmer must use the parallel data structures provided by the library. This data structure is made to be distributed among processes or threads. In our work, we implement Pipeline and Farm patterns, which SkeTo does not provide.

Muesli [EK14] implements MPI support for Farm patterns in a hybrid environment using MPI and OpenMP. Similar to SkeTo [MIEH06], Muesli also provides parallel and distributed data structures such as distributed arrays and matrices. It also allows serializing arbitrary data types. The programmer can implement the abstract class `MSL_Serializable` on the type that needs to be serialized. The API requires the programmer to copy the data to another buffer, provided by Muesli. Our approach for serialization has similar goal to FastFlow, where data is not copied to another buffer (also called zero-copy serialization).

Although [GVL10] mentions that SKELib [DS00] implements distributed-memory support using MPI, it actually uses TCP/IP sockets. SKELib implements Farm, Pipeline, Map, While, and Seq skeletons. The work does not elaborate on how serialization is done.

Quaff [FSCL06] is a C++ template-based skeletons library, aiming at reducing the overhead of object-oriented abstractions. According to [FSCL06], libraries like Muesli frequently introduce overheads due to virtual function calls. Quaff seeks to provide efficient, low-overhead skeletons with a high-level API. The work supports MPI but does not elaborate on the serialization process. However, we found an online repository[3] and it seems that it uses the Boost MPI implementation, which includes serialization functionality. In our work we do not enforce any serialization method, although we provide abstractions using C++ templates to simplify serialization for types that are contiguous in memory.

4.3 Summary of runtime libraries

In this section, we summarize our findings regarding C/C++ libraries for distributed stream parallelism. Many high-level dataflow and patterns-based C++ exist with support of MPI or other distributed frameworks. In general, they are not well documented or have tradeoffs that we do not believe to be ideal. All libraries have one or more of the following problems:

- Lack of documentation for the distributed support or no documentation at all;

[3] https://github.com/MKG/quaff

- No public repository available or downloadable source files (therefore we cannot use in our work);
- Obligatory use of Boost serialization or external libraries (like Cereal);
- Not actively in development anymore (for instance, Muesli v3.0 is 4 years old);
- Some libraries do not abstract MPI details or are based on dataflow, which are not skeleton-based.

Although Thrill also supports distributed computing over MPI, it works in a way similar to Apache Spark with batch streaming, and offers dataflow-like stream processing. DSPARLIBimplements a more generic API based on Pipeline and Farm patterns. PiCo describes itself as a DSL. However, the implementation is in essence a C++ template library for shared-memory architectures. Thrill and PiCo have shown that C++ can be an effective language for stream processing due to its performance and expressiveness using the features of C++11 and C++14. We verified that Thrill does implement MPI [4], since the implementation is publicly available and serialization and networking is described in [BAJ+16]. We also found the code for MPI and TCP socket communication on the repository of Thrill. For GrPPI, we found the GitHub repository without the MPI support code. We presume the library is still in an early prototype state, being not ready to be used in other works.

We also reviewed how these works implement distributed computing. We are interested in implementation details such as networking and data serialization. These works have helped us understand more of the stream processing and stream parallelism domains in a distributed context. MPI-based streaming is of the utmost importance for our work, for reasons that will be described further in Section 5.4. However, we found that many MPI-based solutions do not abstract MPI details [PMG+17, MMP10, WR09]. This is a design choice of these libraries, which focus on extending the MPI standard towards streaming.

Moreover, some skeleton-based libraries also do not abstract MPI details [Col04]. Other libraries have unclear or difficult serialization strategies, enforcing the use of a specific serialization method chosen by the authors [EK14, FSCL06]. Other skeleton libraries do not implement Pipeline and Farm patterns [MIEH06].

A common theme among skeleton libraries that support distributed-memory programming is the need to use their distributed data structures [EK14, MIEH06]. This makes the libraries difficult to use for simpler data structures that are not distributed (and do not need to be). For instance, in a video processing application, each item in the stream can be a different frame. In a streaming scenario, we transfer the bytes of the frame to other nodes in the cluster. It is important for a streaming library to be compatible with any video API. In our case, we use OpenCV in tested applications. It is important that the library offers as few obstacles as possible for this, using a straightforward approach. Our approach is

[4] https://github.com/thrill/thrill/tree/master/thrill/net/mpi

described in Section 5.6. Furthermore, it is important that serialization is a core part of the library. Most works do not describe serialization in depth or enforce a single mechanism [EK14, LGFd+19]. It is important to let the user specify serialization in a different way if needed, to improve performance.

To implement support for distributed-memory architectures in SPAR, we need a structured stream parallelism library based on parallel patterns. Since MPI is ubiquitous in HPC, this library can support MPI as a runtime environment. The serialization strategy must be clearly explained and must allow zero-copy serialization, with a simple API that facilitates code generation. We prefer *not* to adventure ourselves into undocumented and unmaintained libraries, therefore, we propose DSPARLIB to tackle the issues we found in the related work. Differently, our programming abstraction level is much higher in the sense that the listed libraries and API could be used as runtime for DSPARLIB.

Table 4.1 shows a summary of the runtime libraries. We show that libraries often do not implement stream patterns. Some of then implement data-parallel patterns. Serialization is often done with low-level MPI or external libraries such as Boost and Cereal. Documentation for the libraries is scarce, therefore, it is difficult to find information on their APIs, serialization methods, or automatic scheduling implementation. Communication in general is done implicitly, where the programmer does not need to know how programs communicate with each other. However, for libraries that extend MPI, communication is explicit. DSPARLIB covers the gaps in these related works by implementing streaming patterns with a clear serialization approach based on C++ templates, allowing zero-copy serialization. DSPARLIB also implements auto-scheduling, which is often poorly documented in the literature, and communication is implicit.

Table 4.1 – Related work comparison for C++ programming and runtime libraries.

Work	Stream Patterns	Runtime	Serialization	Doc.	Auto-Sch.	Comm.
FastFlow [ADKT17]	Yes	ZeroMQ	Zero-copy	Almost none	Yes	Implicit
GrPPI [dRADFG17]	Yes	MPI*	Boost*	None	Yes	Implicit
PiCo [Mis17]	No	None	N/A	N/A	Yes	Implicit
MPI Streams [PMG+17]	No	MPI	Low-level MPI	None	No	Explicit
MPI Hybrid [MMP10]	No	MPI	Low-level MPI	None	No	Explicit
MPI Lightweight [WR09]	No	MPI	Low-level MPI	None	No	Explicit
Thrill [BAJ+16]	No	TCP, MPI	“Cereal” library	Yes	Yes	Implicit
eSkel [Col04]	Yes	MPI	Low-level MPI	Yes	No	Implicit
Muesli [EK14]	Yes	MPI	Boost	Yes	Yes	Implicit
Quaff [FSCL06]	Yes	MPI	Boost	None	No	Implicit
DSPARLIB	Yes	MPI	Zero-copy with template-based helpers	Chapter 5	Yes	Implicit

4.4 High-level parallel programming interfaces

StreamIt is a programming language developed by MIT since 2001. StreamIt has language-level support for stream parallelism and generates code in the Java programming language [TKA02, TKG+01]. StreamIt provides a toolset with a graph visualizer that shows the computation graph with dotviz, and a benchmarking suite of 12 real-world applications parallelized with StreamIt. StreamIt supports multi-core architectures and distributed computing via TCP/IP.

A StreamIt program is composed of filters. Each filter has an initialization function and a worker function. The worker function declares its inputs and the rate in which it pops and pushes from an implicit FIFO queue that connects filters through a Pipeline.

StreamIt has many language constructs that allow for flexible stream parallelism dataflow graphs. Listing 4.8 shows some of StreamIt features. The stream is built by the `MovingAverage` block, which adds 3 stages to a Pipeline. `IntSource` generates a stream of integers in increasing order. Notice that the `work push 1` means it pushes 1 element to the stream each time it is requested. The `Averager` generates an average of the last 10 elements as indicated in the `peek` keyword and effectively calculates a sliding window with a window size of 10 and a step size of 1. Ultimately, the `IntPrinter` takes an element from the stream and prints it. Notice that each element must be popped from the implicit FIFO queue.

```
1  void->void pipeline MovingAverage {
2      add IntSource();
3      add Averager(10);
4      add IntPrinter ();
5  }
6  void->int filter IntSource {
7      int x;
8      init { x = 0; }
9      work push 1 { push(x++); }
10 }
11 int->int filter Averager(int n) {
12     work pop 1 push 1 peek n {
13         int sum = 0;
14         for ( int i = 0; i < n; i++) {
15             sum += peek(i);
16         }
17         push(sum/n);
18         pop();
19     }
20 }
21 int->void filter IntPrinter {
22     work pop 1 { print(pop()); }
23 }
```

Listing 4.8 – StreamIt example. Adapted from [Str06].

Spidle [CHR+03] is a stream programming DSL. It allows users to define connectors and filters to form a computation graph. The DSL is designed to define the structure of the computation while the computation itself is implemented in C or C++ programming languages. For instance, Listing 4.9 shows the definition of a filter.

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

Listing 4.9 – Spidle example [CHR+03].

DirectFlow [LB07] is a DSL for information-flow systems that seeks a balance between expressiveness and efficiency. The language has few components, where a *pipe* is the basic building block. Pipes can be further connected to form a Pipeline. A basic example is shown in Listing 4.10, where the inputs and outputs are declared, and a process is executed where a data packet is taken from an input port, processed, and then the packet is sent to the output port.

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

Listing 4.10 – DirectFlow example [LB07].

There are parallel programming interfaces that offer annotation-based parallelism. OpenMP [DM98] is considered the *de-facto* standard for parallel programming in shared-memory paradigm [Gri16]. OpenMP uses C pragmas to express parallelism. There are ways to introduce stream parallelism via task dependencies on OpenMP 4.5 [Ope20, Gri16]. However, OpenMP lacks more natural stream parallelism abstractions, therefore, the user must deal with low-level details of stream parallelism. Another project is ompSs [Omp20b], which is an implementation of OpenMP standard API. OpenMP features such as task dependencies were first proposed by OmpSs. OmpSs also supports MPI for cluster computing,

however, memory must be allocated using ompSs functions, where memory can be allocated locally or across all cluster nodes [Omp20a].

OpenStream [PC13] is an extension to OpenMP that adds features for better exploitation of stream parallelism. This is done by adding Pipeline parallelism directives in a GCC-based compiler. OpenStream implements stream parallelism via data flow or FIFO queue behaviors. Application programmers use pragma annotations to use the features. No distributed-memory architectures are suported.

[HBB+18] reviewed other languages, such as CQL [ABW06], Lustre [CPHP87], SPL [HSG17, AGT14] and others. The review offers a view into the principles of stream DSLs, and an assessment of the adoption of such languages. In general, DSLs follow principles around performance, generality, and productivity. Performance principles are windowing, partitioning, stream graph, and restrictions. These principles dictate how the stream is processed and how parallelism is achieved. Generality concerns the language elements themselves, such as orthogonality (no redundant language features), no built-ins (operators are user-defined), and others. The productivity principle concerns familiarity (some streaming languages are similar to SQL) and conciseness (common tasks have concise syntax). However, streaming languages may not follow all principles defined by [HBB+18]. Adoption is a problem for streaming DSLs. There is no single language broadly adopted. However, it is possible to create a standard language that could be used by many streaming frameworks, analogous to the SQL standard supported in all mainstream relational databases. [HBB+18] also mentions a lack of coordination in streaming research, where new researchers do not know each other's work.

4.5 Summary of high-level programming interfaces

In the Section 4.4 we reviewed some high-level programming interfaces for stream parallelism. The programming interfaces are external DSLs, extensions to GCC to implement compiler directives. The programming interfaces are similar in their goals, each one attempting to optimize a certain aspect of stream processing. Specifically, SPAR is the only one which allows the programmer to seamlessly use C++ and annotations, which may help the language to be adopted to parallelize existing code.

Table 4.2 shows a summary of DSLs reviewed in Section 4.4. Tools such as StreamIt, Spidle, and DirectFlow implement new external DSLs that require rewriting the application source code on the new language. Spidle can utilize original C code, however, it must be put in a separate file. Although SPAR is also a DSL, it is compatible with C++11 code. The application programmer does not need to rewrite existing source code. The external DSLs are either not based on C++ or do not support distributed-memory architectures. Furthermore, the approaches followed by [PC13] and [Omp20b] require parsing C pragmas,

which are not fully represented in the C++ syntax tree. With C++11, the annotations are fully represented in the syntax tree, which makes it more suitable to create new DSLs [Gri16]. The high-level programming interfaces exploit parallelism via operating system threads or custom runtime implementations. Serialization is not necessary for most of them, since they run in shared-memory architectures. Therefore, SPAR is the only high-level programming interface based on C++ annotations, supporting shared-memory and distributed-memory architectures.

Table 4.2 – Related work comparison for high-level programming interfaces.

Work	**Interface Type**	**Distributed**	**Language**	**Serialization**	**Runtime**
StreamIt [TKA02]	External DSL	Yes	Java	Java	Custom
Spidle [CHR+03]	External DSL	No	C	N/A	Threads
DirectFlow [LB07]	External DSL	No	Java	N/A	Threads
OpenStream [PC13]	C pragmas	No	C/C++	N/A	Threads
ompSs [Omp20b]	C pragmas	Yes	C/C++	N/A	Custom
SPAR [GDTF17]	C++ annotations	Yes	C++	Sections 5.6 and 6.3.6	FastFlow and DSPARLIB

5. DSPARLIB: DISTRIBUTED STREAM PARALLELISM LIBRARY

In this chapter, we describe the design and implementation of DSPARLIB (acronym for Distributed Stream Parallelism Library), a library for stream parallelism for distributed architectures.

The library will be used to generate parallel code for distributed architectures using SPAR. However, the API aims easy of use for C++ application programmers. DSPARLIB aims also performance comparable to MPI handwritten programs. The abstractions provided by the library can not add much overhead.

This chapter is structured as follows: Section 5.1 covers the design goals of the API. Section 5.2 provides a high-level overview of the main building blocks of DSPARLIB. In Section 3 we show how SPAR generates FastFlow code for shared-memory architectures while the support for distributed-memory architectures (such as clusters) has not yet been implemented. In Section 4.1.1, we provided an overview of FastFlow. Differently, now we describe how SPAR uses the FastFlow API in Section 5.3. This analysis will tell which features DSPARLIB needs for compatibility with SPAR. Section 5.4 discusses our choice for MPI in this work. Sections 5.5 to 5.10 describe how the library works, which are the main abstractions, and how serialization is done. Finally, we test the performance of DSPARLIB against handwritten MPI applications in Section 5.11.

5.1 Design Goals

In this section, we cover the design goals of DSPARLIB. One of the goals of this work is to support distributed-memory architectures in SPAR. This will be done by generating code using DSPARLIB. However, supporting SPAR code generation is not the only use case planned for the library.

DSPARLIB has to be easy to use by application programmers. We focus on providing a safe and high-level API that checks for incorrect usage at compilation time as much as possible. In the latest years, the Rust programming language has been focusing on memory and type safety. We are familiar with the Rust programming language [PGF19], and we take inspiration on these design goals and apply them in the design of DSPARLIB. Specifically, DSPARLIB does not unnecessarily expose the application programmer to raw pointers, and DSPARLIB also ensures that types are correct at compile-time using C++ templates, never using `void*`.

During compilation time, DSPARLIB has to detect the intended usage of the abstraction, for instance, whether a sequential wrapper is intended to be an Emitter, Worker, or

Collector. Depending on the intended usage, DSParLib emit usage errors at compile time if any are found.

Having compile-time checks helped during the development of the code generation, described in Chapter 6. In the beginning, our code generation process would often generate wrong code, but the compile-time checks ensured that we could catch these errors early, instead of having to run the code and debug it to understand what went wrong. Therefore, we argue that this strategy is good for application programmers, and also good for automated code generation development.

Serialization is a major issue that needs to be solved in C++, as discussed in previous sections. Instead of using a third-party library such as Boost or Cereal, DSParLib has to offer a simpler and lightweight strategy that abstracts details of serialization, while allowing efficient and zero-copy serialization. If low-level serialization in the chosen distributed runtime (MPI) is needed, DSParLib has to allow it.

Finally, refactoring operations have to be done with safety and simplicity. An example is changing the sequential wrapper used as a Worker to be used as an Emitter (when it makes sense). If the Emitter does not receive stream items, the programmer only implements the necessary methods without receiving parameters that are ignored. We explain this further in Section 5.5. [Col04] created 4 principles for developing skeleton-based libraries as shown in Section 4.2.4. We describe in the following how the principles apply to DSParLib:

1. **Propagate the concept of skeletons with minimal disruption**. The libraries should have simple concepts and act as a bridge to the standards of the day. We implement DSParLib in C++, which is a widely used language. The distributed runtime is MPI, which is a widely-used parallel programming standard for HPC. We design a simple API that allows the programmer to "wrap" existing code with few refactorings. The library is header-only where there is no need to compile dynamic or static libraries separately. We also used the concepts commonly found in the literature such as Pipeline and Farm.

2. **Allow ad-hoc parallelism**. The libraries should allow the programmer to employ other parallelism techniques, because it is unrealistic to assume all algorithms are efficiently expressible using skeletons. We do not prevent the programmer from spawning threads or using other runtime libraries together with DSParLib. For instance: using MPI or GPU parallel linear algebra libraries [BCC+97] together with DSParLib should be possible, but we have not tested it. However, the user must be careful to not interfere with DSParLib messaging mechanisms (i.e. the user should not send MPI messages to the DSParLib nodes).

3. **Accommodate diversity**. The libraries should balance design simplicity with the pragmatic need for flexibility so that more algorithms can be expressed in a skeletal system.

We support semi-arbitrary pattern nesting, where the programmer can use a Farm inside a Pipeline when needed. Furthermore, sequential code wrappers are easy to change and maintain.

4. **Show the pay-back**. To increase acceptance of the libraries, they must show how they improve the status quo and why the benefits outweigh the initial overheads of moving code to such libraries. In this work, we show that the API is simpler than writing MPI code by hand. We also show that the overheads for real-world applications are not significant when comparing to MPI programs. Furthermore, developers have easy access to features that can significantly improve performance such as the on-demand scheduling.

5.2 DSParLib Building Blocks

DSParLib was designed for programmers to develop parallel stream processing applications following a building block concept, which is adopted in different styles by state-of-the-art structured parallel programming APIs such as FastFlow and TBB. In our context, a block consist of kernels named as Sequential Wrapper, Input Serializer, and Output Serializer, which are illustrated in Figure 5.1. A block must have a Sequential Wrapper and optionally Input or Output Serializers. This input and output customization are needed depending on the parallel pattern and dataflow dependency. Therefore, the programmer has three block types to be used on the parallel patterns.

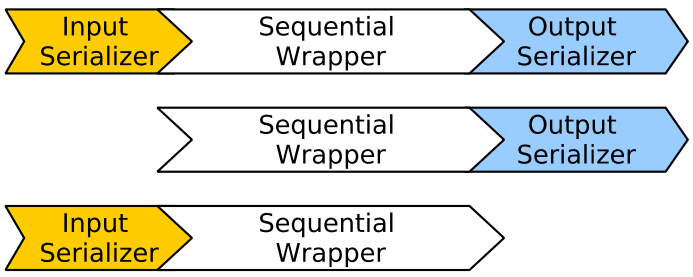

Figure 5.1 – DSParLib's fundamental block.

Figure 5.2 is presenting the parallel patterns available. The Farm requires three blocks. The first one becomes the Emitter (scheduler) of the stream items. The second one becomes the Worker (intensive computation) for replicating stateless operators. The third one is to become the Collector (gathers Workers' results) for joining the Workers' stream items. The communication is from the left hand to the right hand side, which is following the formal definition in Section 2.7. Therefore, the three block types are necessarily applied. The Pipeline pattern may have two or more blocks that become the parallel stages, which is analogous to workstations of an assemble line.

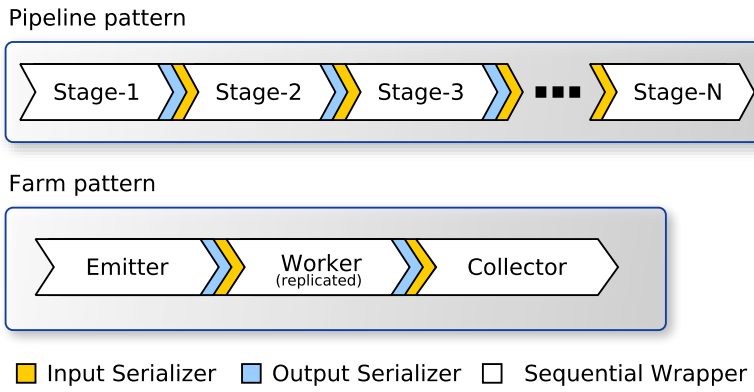

Figure 5.2 – DSPARLIB's building blocks.

Pattern composition is also supported in DSPARLIB by nesting Farm into Pipeline stages such as depicted in Figure 5.3. Doing that, the Farm pattern has to change the original block types for Emitter and Collector, where Input and Output Serializers are respectively need as they communicate with other stages in the Pipeline.

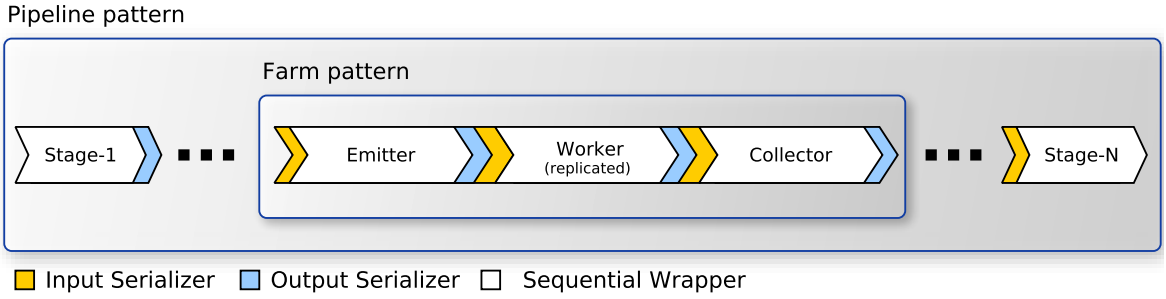

Figure 5.3 – Pattern composition/nesting in DSPARLIB.

For now these are the supported building blocks, which can also be seen as *lego bricks* to build parallel stream processing programs without entering on the low-level distributed-memory architecture developing. In the future, we plan to increase the flexibility, offering new block types, arbitrary nesting, and new parallel patterns. The next sections will provide coding details following this building block concept.

5.3 FastFlow in SPAR

DSPARLIB was inspired by FastFlow. FastFlow has a very dynamic API where arbitrary computation graphs can be built. However, SPAR uses only a subset of these features to generate Pipeline and Farm patterns, as described in Section 3. These patterns are provided by the FastFlow API as the core patterns, and serve as building blocks for more elaborate patterns. In this section, we analyze how SPAR uses FastFlow to understand which features are important for our work.

One of the building blocks of FastFlow is the `ff::ff_node struct`. There is also a `ff::ff_node_t<IN_t, OUT_t = IN_t>: ff_node` where `IN_t` and `OUT_t` are the types of data processed by the node. This typed node provides a typed pointer to the programmer, so casting `void*` to the actual type is not necessary. There are also more types that inherit `ff::ff_node` to implement other features, such as support for functions and lambdas. SPAR uses the typed node `ff::ff_node_t<IN_t, OUT_t = IN_t>`. To use it, the programmer needs to implement a `class` or `struct` as shown in Listing 5.1:

```
1 class Emitter: public ff::ff_node_t<TaskData> {
2     TaskData* svc(TaskData* notused) {
3         for (int i = 0; i< 100; i++) {
4             ff_send_out(new TaskData(i));
5         }
6     }
7 };
```

Listing 5.1 – FastFlow ff_node_t Emitter.

To implement a `ff::ff_node_t<TaskData>`, the programmer must override the `TaskData * svc(TaskData * notused)` method. However, in Listing 5.1, we have an Emitter of data, which does not depend on a task item coming from the stream (and in this example we name the task parameter as `notused`). In this case, the argument `TaskData* notused` will be `NULL`. The function `ff_send_out` is used to send the data to the next stage. The code produces 100 data items and sends them to the stream.

Listing 5.2 shows the implementation of a Worker. The programming interface is the same as the producer, but the code inside `svc` is different. The code takes an item from the stream, performs computation with the item, and sends the result to the next node.

The programmer must carefully place the `ff_node_t` instance in the correct position of the stream. For instance: if the programmer accidentally places a Worker in place of the Emitter and does not check for `NULL`, dereferencing the `NULL` pointer would cause a runtime error.

```
1 class Worker: public ff::ff_node_t<TaskData> {
2     TaskData * svc(TaskData * t) {
3         int workerResult = t.number * 2;
4         delete taskData;
5         ff_send_out(new TaskData(workerResult));
6     }
7 };
```

Listing 5.2 – FastFlow ff_node Worker.

Another way to create a node is to use a function instead of a `class` or `struct`. The function can be passed to the pattern instance (Farms or nodes) later. This is shown in Listing 5.3.

```
TaskData * myWrapper(TaskData * t, ff_node *const) {
    int workerResult = t.number * 2;
    delete taskData;
    ff_send_out(new TaskData(workerResult));
}
...
ff_node_F<TaskData> wrapperNode(myWrapper);
```

Listing 5.3 – FastFlow function wrapper.

Listing 5.4 shows the implementation of a Collector. The interface and usage are similar to the Worker example, but the stage must return the `GO_ON` tag to inform the FastFlow runtime that it must wait for new items.

```
class Collector: public ff::ff_node_t<TaskData> {
    TaskData * svc(TaskData * t) {
        std::cout << t.number << std::endl;
        delete t;
        return (TaskData*)GO_ON;
    }
};
```

Listing 5.4 – FastFlow ff_node Collector.

To create the Farm pattern, the programmer must create an instance of `ff::ff_farm`. If the programmer needs to process stream items in the order they were produced, `ff::ff_OFarm` can be used instead. Listing 5.5 shows how to create and run a Farm pattern. First, we instantiate the Workers and pass them to the Farm `class` instance using the constructor. Later, we can set the Emitter and Collector nodes, and call `set_scheduling_ondemand` if we intend to use on-demand scheduling.

```
std::vector<unique_ptr<ff::ff_node>> workers;
for(int i=0; i < replicas; i++){
    workers.push_back(std::make_unique<Worker>());
}
ff::ff_OFarm<task_data> ofarm(move(workers));

Emitter e;
ofarm.setEmitterF(e);
Collector c;
ofarm.setCollectorF(c);

ofarm.set_scheduling_ondemand();

ofarm.run_and_wait_end();
```

Listing 5.5 – FastFlow Farm example.

The SPAR compiler generates FastFlow code using the API described in this section, which follows a structured parallel programming approach [MRR12]. Since SPAR also

follows this approach for code generation inside the compiler and to be compliant with the design principles, DSPARLIB has to provide a “similar” structured parallel programming API such as FastFlow. For instance, Farm, Pipeline, and the combination of Pipeline with Farm patterns. In the next section, we will describe which are the similarities and benefits of DSPARLIB with respect to FastFlow's API as well as the implementation design choices.

5.4 Runtime for DSPARLIB

Our runtime choice is guided by two factors: supporting future autonomic computing research and programming flexibility for distributed-memory architectures such as clusters. The overall motivation of the SPAR project is to make structured stream parallelism easier for application programmers. This is achieved by having simple annotations to separate the computation into many parallel stages. However, the distributed-memory introduces new challenges: deploying the program in the cluster can be significantly more inconvenient and serialization of data stream items can be difficult in C++. In this section, we evaluate runtime libraries for distributed parallelism.

Griebler's thesis [Gri16] pointed out FastFlow's limitations for distributed-memory architectures in SPAR. FastFlow uses ZeroMQ, a message queue system that needs to be installed on every node of the cluster. In HPC, it is common for clusters to have an MPI implementation already available. The MPI program can be started by running `mpirun` or `mpiexec` while in FastFlow the processes need to be manually launcher. Moreover, the support for distributed computing in FastFlow is so far experimental and undocumented[1]. It is still worth considering the use of FastFlow, if the rich feature set it provides on shared-memory architectures can also be available for distributed-memory architectures.

HPX was also considered. HPX is a C++ runtime library for massive parallelism, covered in Section 2.9.2. HPX supports batch systems such as PBS, which facilitates running programs on a cluster. Performance is also shown to be better than MPI for some workloads [KHAL+14]. However, as discussed in Section 2.9.2, it is unclear how well HPX supports stream parallelism. We found no studies about HPX and streaming applications besides an unmaintained library on GitHub. It is also not clear how well HPX works when using ad-hoc parallelism, such as MPI linear algebra libraries or manually creating threads, and C++ synchronization primitives (mutexes, condition variables, and others). HPX re-implements these primitives to make them work with the threading subsystem without blocking threads. Data serialization abstractions are a little better then in MPI.

We use MPI [For12] since it provides the necessary features and flexibility for implementing DSPARLIB, including dynamic allocation of processes. MPI is already installed

[1] For instance, tutorial on distributed support is not available at `http://calvados.di.unipi.it/storage/tutorial/html/tutorial.html`

in many HPC clusters and researchers can take advantage of MPI implementations specific to that cluster. For instance: supercomputer manufacturers, such as Cray, support MPI 3.0 with their own MPI implementation which is based on MPICH. Furthermore, there are HPC libraries that use MPI, such as Scalapack [BCC+97]. Therefore, DSParLib could be used together with these libraries, although we have not tested it. There are studies showing that MPI can be used in streaming applications, covered in Section 2.9.1. Finally, our work continues a previous study [GF17] that shows it is possible to support distributed-memory architectures on SPar with MPI.

DSParLib will initially support Open MPI. We specifically support Open MPI because MPI implementations have different behavior when compared to each other, even if the API is the same. Early tests on Windows MPI with default configurations showed that it runs out of memory if the message consumer is too slow when using round-robin Farm scheduling. In Open MPI, the same behavior is not observed. Open MPI is also a mature product with extensive documentation. The downside of using MPI is the complexity. Specifically, collective operations need to be called by all processes in the communicator. For instance: when dynamically spawning a process, all other processes need to call the `MPI_Spawn` function. Other functions need only the participation of two processes, for instance: `MPI_Recv` and `MPI_Send`. DSParLib will hide this complexity behind a simpler API.

5.5 DSParLib's Sequential Wrappers

In this section, we describe the DSParLib API to express sequential code wrappers, which attempts to offer the functionality of FastFlow's `ff_node`. With this, it is possible to reuse the same code generation logic/strategy currently implemented in SPar. While this API is heavily inspired by FastFlow, we are implementing it differently in a way that might improve usability for programmers who will not use SPar. Many of those improvements are dedicated to catching errors in compile-time instead of run-time.

The concept of a sequential code wrapper was introduced earlier in Section 5.2. The wrapper is a `class` that "wraps" the sequential code of the application. The wrapped code is managed by DSParLib. This allows DSParLib to run the code in parallel to other codes. The wrapped code also has access to an `Emit` function to send data to other components (Pipeline stages or Farm components).

In Listing 5.6, we show an example of a sequential code wrapper that takes an integer and emits the square root of the integer as a double. The method `Process` is called once for every stream item and has no return type (`void`). Notice that this method executes *only when a stream item is received*: there is no other situation where this method is called. Each method of the interface has a well-defined role. Differently, in FastFlow the `svc` method is called in at least 2 situations: Called only once in an Emitter, or once for each stream item

in a Worker or a Collector. Therefore, `svc` has 2 responsibilities: start the stream and emit items, or compute stream items.

```
1 class Worker: public Wrapper<int, double> {
2 public:
3   void Process(int &i) override {
4         double square_root = sqrt(i);
5         Emit(square_root);
6   };
7 };
```

Listing 5.6 – DSPARLIB sequential wrapper example.

In DSPARLIB, a sequential code wrapper is a thin abstraction layer that just wraps sequential code. There is no platform-specific details in a sequential wrapper. This is in contrast to FastFlow, where distributed support is done with a `ff_dnode class` that wraps the code, communicates with the network, and specifies de/serialization. Furthermore, the programmer must be aware of the architecture of the stream: the `ff_dnode` is only used to communicate with the network. The programmer can use both `ff_node` and `ff_dnode` to write efficient programs that communicate over the network only when necessary. Shared-memory architecture support can be developed for DSPARLIB without changing the wrappers.

The wrapper in Listing 5.6 can be used as a Worker or any stage that is not the start or end of the Pipeline. Consider the following pattern instantiation with the schema $pipe(\square_0, farm(E(\square_1), W(\square_2), C(\square_3)))$, which is a Pipeline with a sequential stage with block $\square_0$ and a Farm stage. The wrapper on the Listing 5.6 can be put in place of $E(\square_1)$ or $W(\square_2)$. However, it can not be put in place of $C(\square_3)$, as the Collector is the last one, it can not emit data this scenario. This is checked at compile-time in the Farm pattern. If the programmer needs to collect the results produced by the code in Listing 5.6, they can use a Collector as shown in Listing 5.7. Notice that the base `class` is still the same `Wrapper<TIn, TOut>` and the overridden method is also the same `void Process(TIn& input)`.

Notice also that there is the `struct Nothing` in the second template parameter of the `Wrapper<double, Nothing>` declaration. This is a `struct` used to inform that no data will be emitted by this wrapper. Setting `TOut` to `Nothing` renders the `Emit` method (seen on Listing 5.6) unusable and by using C++ template metaprogramming and SFINAE (Substitution Failure Is Not An Error) techniques, the `Emit` method is removed from the base `class`. For instance, the programmer cannot even call `Emit(Nothing{})` because the method does not exist in this case. This prevents incorrect usage of the API at compile time. We tried to use `void` instead of `Nothing`, but we were unable to make it work.

```
1 class MyCollector : public Wrapper<double, Nothing> {
2 public:
3   void Process(double &sqrtResult) {
```

```
4       std::cout << sqrtResult << std::endl;
5       //Emit(Nothing {}); would cause compilation error
6     };
7   };
```

Listing 5.7 – DSPARLIB Collector example.

The input of the `Process` method is a reference instead of a pointer. C++ does not allow this type of value to be null, therefore, the input parameter is always a valid object. This reduces the need for defensive programming, such as checking for `NULL` or `nullptr` at every stream item. This is not necessary when using FastFlow either, but we argue that during a refactoring, the programmer might end up with code that accidentally tries to dereference a `NULL` pointer when transforming a Worker into an Emitter, as described in Section 5.3. We intend to catch as much as possible incorrect API usage during compilation instead of during runtime.

Revisiting the FastFlow Emitter example in Listing 5.1, we see that the `svc` method receives a parameter that is not supposed to be used (the value is `NULL`). DSPARLIB does it differently, as can be seen on Listing 5.8. The `TIn` template parameter is `Nothing`, which indicates that the stage will not receive anything. The effect is that the programmer cannot override the `Process` method, and there is no reason to implement `void Process(Nothing&)` which is never called. If the programmer tries to write the code `void Process(Nothing&) override`, the template meta-programming techniques employed will avoid marking the base method as virtual, therefore, the programmer cannot override it. Without the override keyword, the programmer will be implementing another method unrelated to the `Wrapper` interface. Take into account that the programmer can implement an Emitter that receives stream items, but in this example, we are describing the first stage or Emitter of the whole stream.

To avoid exposing pointers to the programmer and possible confusions caused by calling the `Process` method in multiple ways, all stages that receive `Nothing` as input must implement the `void Produce()` method. The method does not receive any parameters and the base class offers the `Emit(TOut&)` method. This is also achieved by using meta-programming techniques, where DSPARLIB requires the programmer to implement `void Produce()` and produces a compilation error if the method is not implemented.

```
1 class MyEmitter : public Wrapper<Nothing, int> {
2 public:
3   void Produce() {
4     for (int i = 0; i < 100; i++) {
5       Emit(i);
6     }
7   };
8 };
```

Listing 5.8 – DSPARLIB Emitter example.

The programmer can also run code when a stream starts or ends, and run code for the first stream item. When the first stream item is received, *both* methods `Process` and `OnFirstItem` are called. In Listing 5.9 we show how to use these methods. We use `OnFirstItem` on OpenCV video applications to open the result video file, with the resolution indicated on the first item.

```
class VideoWriterWrapper : public Wrapper<VideoFrame, Nothing> {
public:
    cv::VideoWriter video;
    void Start() override {
        std::cout << "Started!" << std::endl;
    }
    void OnFirstItem(VideoFrame& frame) override {
        cv::Size frame_size(frame.width, frame.height);
        video.open("result.avi", frame_size);
    }
    void Process(VideoFrame& frame) override {
      video.write(frame.AsOpenCVVideoFrame());
    }
    void End() {
        video.close();
    }
};
```

Listing 5.9 – Start and End methods on wrappers.

So far, we have not described how these stages will physically communicate with each other. In the next section, we will address the serialization problem in DSParLib with MPI.

5.6 Serialization and deserialization in MPI

Frameworks and libraries for structured parallelism that run in a shared-memory architecture do not have to handle serialization. FastFlow uses queues that store the pointers to the data in memory, therefore, the item data does not have to be copied. In MPI or in any other distributed environment, data must be copied between processes (in the same or different machines). This is a complex task with different options available. For instance, in Java and .NET based applications, serialization can be done mostly automatically since these environments allow the use of runtime reflection (or introspection). This allows users of these frameworks to write code that never handle serialization because frameworks will often handle it automatically.

Languages such as Rust allow the use of procedural macros to introspect structs at compile time and generate fast serialization code. Java frameworks running as a GraalVM native image can also use compile-time class introspection for low-memory usage and fast

serialization speeds. Recently, C# has also implemented similar features to allow compile-time source code generation.

However, serialization and deserialization are often expensive and can be a significant part of the computation. Although Big Data streaming frameworks usually deal with this problem transparently, they allow the programmer to customize the serialization method. FastFlow has a `ff::ff_dnode` `struct` for distributed computing, which allows the programmer to perform fast zero-copy deserialization of data coming from a ZeroMQ queue and operating system routines such as `iovec`.

DSPARLIB will use a similar strategy using the `MPI_Send` and `MPI_Recv` functions. In MPI, primitive data types and simple `struct`s can be serialized in a many ways. In Listing 5.10, we show 2 ways of sending and receiving a simple data type that is contiguous in memory: using the appropriate MPI type `MPI_INT` (lines 2 and 6) or sending it as bytes using `MPI_BYTE` (lines 3 and 7). We will discuss the implications of sending data using `MPI_BYTE` in Section 5.7. For data that is allocated contiguously in memory such as a simple `struct`, the programmer can use `MPI_BYTE` or define a `MPI_Datatype` as well, but for brevity, we do not show the use of `MPI_Datatype` here. We do not use `MPI_Datatype` in our final solution, however, the abstractions built into DSPARLIB do not prohibit the programmer to use `MPI_Datatype` if needed.

```
1 int data = getData();
2 MPI_Send(&data, 1, MPI_INT, proc, tag, comm);
3 MPI_Send(&data, sizeof(int), MPI_BYTE, proc, tag, comm);
4
5 int recv;
6 MPI_Recv(&recv, 1, MPI_INT, proc, tag, comm);
7 MPI_Recv(&recv, sizeof(int), MPI_BYTE, proc, tag, comm);
```

Listing 5.10 – MPI Send example.

For `struct`s with pointers, the pointer needs to be send separately, together with the size or use `MPI_Get_count` to discover the size of the message. Complex data structures also need their fields to be sent individually. Two of our tested applications (lane detection and face recognition) need to send OpenCV matrices of type `cv::Mat`. The class has a complex structure and many fields. It is possible to extract the field values and call an appropriate constructor, making it possible to serialize and deserialize data items.

In Listing 5.11, we show an example with a simplified version of the `cv::Mat` class. Observe that the order of the sends and receives matter: the order that the data is received must match the order of the data sent. The receiver must allocate memory based on the size received earlier. Another solution to reduce the number of sends and receives is shown in Listing 5.12. In this example, we send the whole contiguous data structure first. DSPARLIB assumes that all nodes in the cluster use the same data representation (same architecture, ex: x86-64). Since the `char* data` pointer is just a simple integer value in memory, there is

no problem sending it to the receiver as long as the receiver does not dereference the pointer before receiving the actual data. Sending fewer messages might improve performance even if the same total amount of data is sent between processes. We show it in Section 6.5

```
struct Mat {
  char* data;
  int size;
  int width;
  int height;
  int type;
};

Mat m = readFrame(...);

MPI_Send(&m.type, sizeof(int), MPI_BYTE, ...);
MPI_Send(&m.width, sizeof(int), MPI_BYTE, ...);
MPI_Send(&m.height, sizeof(int), MPI_BYTE, ...);
MPI_Send(&m.size, sizeof(int), MPI_BYTE, ...);
MPI_Send(m.data, m.size, MPI_BYTE, ...);

Mat m;
MPI_Recv(&m.type, sizeof(int), MPI_BYTE, ...);
MPI_Recv(&m.width, sizeof(int), MPI_BYTE, ...);
MPI_Recv(&m.height, sizeof(int), MPI_BYTE, ...);
MPI_Recv(&m.size, sizeof(int), MPI_BYTE, ...);
m.data = new char[m.size];
MPI_Recv(m.data, m.size, MPI_BYTE, ...);
```

Listing 5.11 – MPI Send complex data structure example 1.

```

Mat m = readFrame(...);

MPI_Send(&m, sizeof(Mat), MPI_BYTE, ...);
MPI_Send(m.data, m.size, MPI_BYTE, ...);

Mat recv;
MPI_Recv(&m, sizeof(Mat), MPI_BYTE, ...);
m.data = new char[m.size];
MPI_Recv(m.data, m.size, MPI_BYTE, ...);
```

Listing 5.12 – MPI Send complex data structure example 2.

5.7 Process communication and serialization in DSPARLIB

In DSPARLIB, each stream data item is called a message. A message is composed of a header and the data payload. The header `struct MessageHeader` is shown in Listing 5.13. The `sender` is the MPI rank of the process that sent the message and the `target` is

the MPI rank intended to receive the message. The `type` indicates whether the message is a stream data item or a stream stop signal. The `id` indicates the ID of the message, starting from 0. The Emitter increases the ID counter by 1 for each message emitted. This is used for implementing the ordering in the Collector later.

```
1 struct MessageHeader {
2   int sender;
3   int target;
4   int type;
5   uint64_t id;
6 };
```

Listing 5.13 – DSPARLIB Message Header.

Consider the communication channel between a sender and a receiver, represented in Figure 5.4. The sender sends 2 messages *msg*1, *msg*2, and each message has 2 fields *field*1, *field*2. The programmer decides to send *field*1, *field*2 in 2 separate messages. Consequently, the receiver starts by accepting any message of type `MessageHeader` from any sender. The receiver will receive the header for *msg*1 first. The next 2 messages are *field*1, *field*2. If many senders are sending data to the receiver, the messages from different senders could interleave. However, upon reception of a `MessageHeader`, the receiver will only match messages coming from a specific sender. This matching mechanism is described in Section 2.9.1. This mechanism is necessary to build a Farm pattern. The Collector might receive messages from many Workers at the same time, therefore, we need to match one sender at a time.

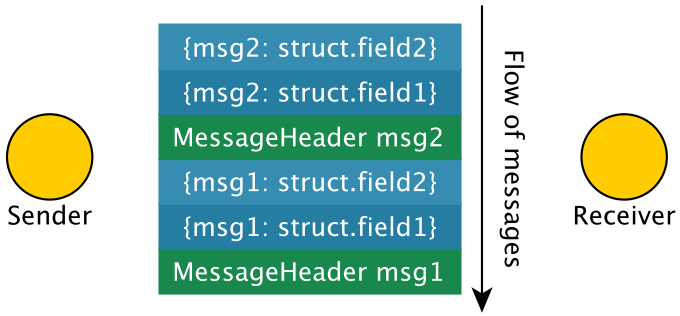

Figure 5.4 – Flow of messages.

Notice that there is no message to inform the end of a single data item. The end of the message is implicit in the code itself: it requires the sender and receiver to call equivalent send and receive methods. After sending or receiving messages, both processes can do other tasks such as processing the data or start sending another message. There is, however, a message to inform the end of the whole stream. In Section 5.9, we show how the stream is stopped.

In general, the process of exchanging messages is done by using simple MPI sends and receives. However, the `MPI_Send` and `MPI_Recv` functions have many parameters that the programmer have to deal with. DSPARLIB offers a simple way to emit a message to other processes without having to deal with MPI types, tags, and communicators. It is necessary that the programmer handles allocation of buffers and send messages in the correct order while the rest is abstracted.

MPI allows the programmer to specify the data type to send and receive. Let us consider why this parameter is useful for MPI implementations. There could be 2 processes running in different CPU architectures on MPI. By passing the correct MPI types, the implementation would do the correct conversions. For instance, different CPU architectures may have different *endianness* (big-endian, little-endian). The data needs to be correctly converted by the MPI implementation. Open MPI supports heterogeneous clusters[2] and does conversions when necessary. However, we presume this scenario is rare. We limit DSPARLIB for scenarios where the same binary runs distributed among many nodes running the same CPU architecture (for instance, x86-64), having the same data representation in all cluster nodes. By imposing this limitation on DSPARLIB, it is possible to use a simpler approach for serializing data: sending the raw bytes of the data and use `MPI_BYTE` data type.

We implemented the `MPISender class` to simplify the usability of the `MPI_Send` and `MPI_Recv` functions. This is shown in Listing 5.14. The main method is `StartSendingMessageTo(int target, uint64_t id)`. This method is responsible for sending the message header to the target rank and also returns the message header object to the caller. This header object can be used in the `SendTo` template methods. Since the message header contains the target rank, `SendTo` can use it to send the stream data to the correct rank. Moreover, by using template methods, it is possible to support any contiguous data type of any size, without having to specify sends and receives for each type. The endianness problem described earlier can be solved partially by using specialized overloads or template specializations on the `MPISender` and `MPIReceiver` classes, but we do not do that at the moment being.

The template methods allow sending up to 3-dimensional statically and dynamically allocated arrays, as well as single contiguously allocated data items. We do not show all these methods in Listing 5.14 for brevity. The template methods can send messages of any contiguously allocated type, and we use the type template parameter to get its size. Furthermore, for statically allocated arrays, we use template `size_t` parameters to get the size of each dimension. Based on the number of elements and size of the data type, it is possible to calculate the size of the buffers to send and receive. In Section 6.3.7, we also describe sending and receiving C++ STL types. However, first we want the reader to familiarize with the core DSPARLIB abstractions and then we will revisit this topic later.

[2]Documentation in `https://github.com/open-mpi/ompi/wiki/Heterogeneous`

```
class MPISender {
MessageHeader StartSendingMessageTo(int target, uint64_t id)  { ... }
MessageHeader SendStopMessageTo(int target, uint64_t id) { ... }
DemandSignal SendDemand(int target) { ... }
template <typename T>
void SendTo(MessageHeader &header, T &buffer) {
  size_t totalBytes = sizeof(T);
  MPI_Send(&buffer, (int)totalBytes, MPI_BYTE, header.target,
     MPI_DSPAR_STREAM_MESSAGE, comm);
}
template <typename T, size_t I>
void SendTo(MessageHeader &header, T (&buffer)[I]) {
  size_t totalBytes = sizeof(buffer);
  MPI_Send(buffer, (int)totalBytes, MPI_BYTE, header.target,
     MPI_DSPAR_STREAM_MESSAGE, comm);
}
template <typename T>
void SendTo(MessageHeader &header, T *buffer, size_t dimension) {
  size_t dataSizeInBytes = sizeof(T);
  size_t totalBytes = dataSizeInBytes * dimension;
  MPI_Send(buffer, (int)totalBytes, MPI_BYTE, header.target,
     MPI_DSPAR_STREAM_MESSAGE, comm);
}
};
```

Listing 5.14 – DSPARLIB MPI Sender.

Similarly to the `MPISender` , we implemented the `MPIReceiver` `class` shown in Listing 5.15. For each `SendTo` method in `MPISender`, there is a corresponding `Receive` method capable of receiving data up to 3-dimensional arrays. It also uses template parameters to calculate buffer sizes. Notice that both `MPISender` and `MPIReceiver` also deal with sending and receiving demand signals through `SendDemand` and `ReceiveDemand`, which are used for on-demand scheduling.

```
class MPIReceiver {
MessageHeader StartReceivingMessage();
DemandSignal ReceiveDemand();
template <typename T>
void Receive(MessageHeader &header, T *buffer);

template <typename T, size_t I>
void Receive(MessageHeader &header, T (*buffer)[I]);

template <typename T>
void Receive(MessageHeader &header, T(*buffer), size_t dimension1);
};
```

Listing 5.15 – DSPARLIB MPI Receiver.

By implementing `MPISender` and `MPIReceiver`, the calls to MPI are wrapped, making sure the correct MPI tags and buffer sizes are used. Furthermore, C++ templates provide a method to allow the programmer to call the same `SendTo` method for any data type, ab-

stracting the serialization process for more complex array types. However, the programmer must still call `SendTo` and `Receive` methods in the correct order to correctly serialize and deserialize a type.

Another layer of abstraction can be built on top of `MPISender` and `MPIReceiver`, called `SendReceive`. This `class` encapsulates the process of serialization and deserialization for a given type. The interface is shown in Listing 5.16. This `class` is implemented by the end user of DSPARLIB. For instance: if the programmer needs to serialize an OpenCV `cv::Mat class`, the programmer can implement a `class` that inherits from `SenderReceiver` and call the correct `SendTo` and `Receive` method calls. Check that the virtual methods in `SenderReceiver` take `MPISender` and `MPIReceiver` parameters as well as `MessageHeader&` parameters. The end user will never create `MessageHeader` instances by themselves, DSPARLIB will automatically handle them whenever a message needs to be sent. The `OnStart` method is called after `MPI_Init` is called. This is useful if the programmer wants to set up a `MPI_Datatytpe` and use low-level `MPI_Send` and `MPI_Send` instead of the higher-level `MPISender` and `MPIReceiver` objects.

```
1 template <typename T>
2 class SenderReceiver
3 {
4   virtual void Send(MPISender &sender, MessageHeader &msg, T &data) =
        0;
5
6   virtual T Receive(MPIReceiver &sender, MessageHeader &msg) = 0;
7
8   virtual void OnStart(){ };
9 };
```

Listing 5.16 – SenderReceiver class.

Some programs might only need to serialize simple `structs` or primitive types like `int`, `double`, `char`. In these cases, the programmer does not need to define a `SenderReceiver`, because DSPARLIB already provides a generic `SenderReceiver` to send any type contiguous in memory. We show the implementation in Listing 5.17, and how to instantiate it in line 21.

```
1  namespace dspar {
2  template <typename T>
3  class TrivialSenderReceiver : public SenderReceiver<T> {
4  void Send(MPISender &sender,MessageHeader &msg, T &data) {
5    sender.SendTo(msg, data);
6  };
7  T Receive(MPIReceiver &receiver, MessageHeader &msg) {
8    T data;
9    receiver.Receive(msg, &data);
10   return data;
11 };
12 };
```

```
13
14 template <typename T>
15 TrivialSendReceive<T> SendReceive() {
16   return TrivialSendReceive<T>();
17 };
18
19 }
20
21 auto doubleSenderReceiver = dspar::SenderReceiver<double>();
```

Listing 5.17 – TrivialSenderReceiver class.

Finally, we show how to send a dynamically allocated pointer in Figure 5.18. We represent an integer array with a `struct` called `IntArray`, which contains the size and the pointer of the allocation. The sender first sends the size field. In the receiving side, we instantiate the `struct` and receive the size, passing a pointer to the `size` field. The array `ptr` is sent by specifying its size. The receiver allocates data and passes the buffer `ptr` and `size` to receive it.

```
1  struct IntArray {
2      size_t size;
3      int* ptr;
4  }
5
6  ...
7
8  class SizedPtrSenderReceiver : public SenderReceiver<IntArray> {
9  void Send(MPISender &sender,MessageHeader &msg, IntArray &data) {
10     sender.SendTo(msg, data.size);
11     sender.SendTo(msg, data.ptr, data.size);
12 };
13 IntArray Receive(MPIReceiver &receiver, MessageHeader &msg) {
14     IntArray data;
15     receiver.Receive(msg, &data.size);
16     data.ptr = new int[data.size];
17     receiver.Receive(msg, data.ptr, data.size);
18     return data;
19 };
20 };
```

Listing 5.18 – Sending structs with pointers.

C++ compilers implement RTTI (Run-Time Type Information), which provides information about the types during runtime. However, it does not work for automatic serialization purposes. RTTI provides basic information about a type and does not allow to list the fields of a `struct`. Furthermore, there is no standard way to get the size of a dynamically allocated pointer. The programmer needs to specify the size of the allocations. However, other details are abstracted so that DSPARLIB makes this process significantly simpler. In many cases, a single `SendTo` call may be sufficient to serialize the data, without dealing with MPI low-level details.

5.8 Pipelines and Farms in DSParLib

In this section, we will present the Farm and Pipeline APIs in DSParLib. To build a Farm or Pipeline, the programmer needs to instantiate the wrappers and serializers properly. However, for the Farm pattern, this can be challenging, given that 4 generic templates are used in the type definition. These types depend on the input and output types of the wrappers. In this section, we show the constructors for the patterns and C++11 techniques to make it easier when using them.

Listing 5.19 shows the type definition for the `class FarmPattern`, with 4 type parameters and 7 constructor arguments. The constructor can be complicated to use, but ensures the object is correctly initialized with the proper stages and serializers (`SenderReceiver` objects). From the 7 constructor arguments, 3 are of type `Wrapper` and 4 are serializers (`SenderReceiver`). It needs 4 serializers: the Farm input, Emitter output (which is also the Worker input), Worker output (which is also the Collector input), and the Farm output (Collector output).

```
1  template <typename EmitterInput,
2            typename EmitterOutput,
3            typename WorkerOutput,
4            typename CollectorOutput>
5  class FarmPattern : public AbstractPipelineElement {
6   FarmPattern(
7    SenderReceiver<EmitterInput> &worldToEmitter,
8    Wrapper<EmitterInput, EmitterOutput> &emitter,
9    SenderReceiver<EmitterOutput> &emitterToWorkers,
10   Wrapper<EmitterOutput, WorkerOutput> &worker,
11   SenderReceiver<WorkerOutput> &workerToCollector,
12   Wrapper<WorkerOutput, CollectorOutput> &collector,
13   SenderReceiver<CollectorOutput> &collectorToWorld) { }
14 }
```

Listing 5.19 – FarmPattern constructor.

DSParLib also allows instantiating Pipeline stages. These Pipeline stages can be used in Pipeline compositions, which will be shown later. The constructor for `PipelineStage` is shown in Listing 5.20. Notice that this `class` also extends from another `class` called `AbstractPipelineElement`, which allows Pipeline composition with Stages and Farms. The constructor of `PipelineStage` takes a stage and 2 serializers for the input and output types.

```
1  template <typename TIn, typename TOut>
2  class PipelineStage:
3      public AbstractPipelineElement
4  {
5    PipelineStage(Wrapper<TIn, TOut> &_wrapper,
6                  SenderReceiver<TIn> &_inputReceiver,
7                  SenderReceiver<TOut> &_outputSender) {}
```

```
8 }
```

Listing 5.20 – PipelineStage class.

Listing 5.21 shows how the programmer can construct these objects. It is possible to use C++ type inference and the `auto` keyword. This allows the programmer to avoid having to specify all the type parameters manually. By analyzing the Farm pattern use case, we found that applications use recurrently the Farm pattern alone without any other stages. In Section 5.11, we will show several benchmark applications and all of them use only a simple Farm pattern. `dspar::Farm` automatically infers the types and fails compilation if the programmer specifies incompatible stages and serializers. The same is done for `dspar::Stage`. We also show a more common use case in the method `SimpleFarm`, where the application is just a Farm pattern. The `farm` object has the following configuration methods: `SetWorkerReplicas` to set the number of Worker replicas, `SetCollectorIsOrdered` to enable ordering constraints in the Collector, and `SetOnDemandScheduling` to enable on-demand scheduling. The default scheduling is round-robin.

Listing 5.21 also shows the `Pipeline` object, which is used for Pipeline composition. The stages of the Pipeline are passed as parameters. The method `Start` calculates the complete Pipeline graph and which MPI ranks will run in which stage. Each MPI rank process will be responsible for a stage. This process will be described with more detail later.

```
1  void PipelineComposition() {
2    auto doubleSerializer = dspar::SendReceive<double>();
3
4    auto stageBeforeFarm = dspar::Stage(firstStage, doubleSerializer);
5
6    auto farm = dspar::Farm(
7          doubleSerializer,
8          emitter, doubleSerializer,
9          worker, doubleSerializer,
10         collector,
11         doubleSerializer);
12
13   farm.SetWorkerReplicas(10);
14   farm.SetOnDemandScheduling(true);
15   farm.SetCollectorIsOrdered(true);
16
17   auto stageAfterFarm = dspar::Stage(lastStage, doubleSerializer);
18
19   Pipeline p(&stageBeforeFarm, &farm, &stageAfterFarm);
20   p.Start();
21 }
22
23 void SimpleFarm() {
24   auto doubleSerializer = dspar::SendReceive<double>();
25   auto farm = dspar::Farm(
26       emitter, doubleSerializer,
27       worker, doubleSerializer,
28       collector);
29
```

```
30    farm.SetWorkerReplicas(10);
31    farm.SetOnDemandScheduling(true);
32    farm.SetCollectorIsOrdered(true);
33    farm.Start();
34  }
```

Listing 5.21 – Simpler Pipeline creation.

To instantiate patterns, the application programmer must think about how the information flows through the pattern and build the patterns with the correct types. In Listing 5.22, we show the syntax to build the Farm pattern. In the "edges" of the Farm (`Emitter` and `Collector`), communication with other patterns may happen. In this case, the `Emitter`'s input or `Collector`'s output must be serialized. Therefore, the programmer must specify the `SenderReceiver`. `TIn` and `TOut` are the generic input and output types of the wrappers.

```
1 auto farm = dspar::Farm(
2     <if Emitter TIn != Nothing> SenderReceiver<Emitter TIn>
3     <Emitter wrapper>,
4     SenderReceiver<Emitter TOut>
5     <Worker wrapper>,
6     SenderReceiver<Worker TOut>,
7     <Collector wrapper>,
8     <if Collector TOut != Nothing> SenderReceiver<Collector TOut>
9 );
```

Listing 5.22 – Farm instantiation syntax.

We explained the patterns API of DSParLib and showed how programmers can use it for parallelizing stream processing applications on distributed-memory architectures. The next section will explain how these classes are used by the DSParLib runtime.

5.9 DSParNode class

In this section, we explain how the wrapper classes are used by the runtime to run the stream. This section will also explain in detail how the stream is managed, how stream ordering is done, and how it supports also on-demand scheduling. Listing 5.23 shows the interface of `DSParNode class`, which is run by all MPI processes in DSParLib.

```
class DSParNode: public DSparLifecycle
{
  AfterStart OnStart()
  void OnStop()
  virtual void OnReceiveMessage(MessageHeader &msg)
  virtual StopResponse OnReceiveStop(MessageHeader &msg)
};
```

Listing 5.23 – DSParNode interface.

`DSParNode` extends from another `class` called `DSparLifecycle`, which is an abstract `class` that listens for messages coming from other processes. The only concern of `DSParLifecycle` is to notify the derived classes about the most important events of the stream, such as stream start, stream stop, and new messages. All the methods shown in Listing 5.23 are overriding methods in the base `DSParLifecycle class`. They are called in a loop as shown in Listing 5.24. The `OnReceiveStop` processes a stop signal and returns a `StopResponse` telling whether the node must finish execution or ignore the signal. This is useful for Collectors to keep track of which Workers have stopped and only stop when the last Worker stops. Similarly, the `OnStart` method returns whether the node must start receiving messages immediately after starting or finish execution. This is useful for Emitters that do not receive stream items.

```
AfterStart behavior = OnStart();
if (behavior == AfterStart::ReceiveMessages)
    while (true) {
        MessageHeader msg =
            receiver.StartReceivingMessage();
        if (msg.type == STOP_TYPE) {
            StopResponse response = OnReceiveStop(msg);
            if (response == StopResponse::Stop)
                break;
        }
        else if (msg.type == MESSAGE_TYPE)
            OnReceiveMessage(msg);
    }

OnStop();
```

Listing 5.24 – DSParLifecycle main loop.

When a data message is received, the `OnReceiveMessage` is called. The `class DSParNode` receives the data message and calls the `Process` method in the `Wrapper class`, which was explained in Section 5.5. If the message was received out of order and ordering constraints are enabled, the data is stored and processed later. This is shown in Listing 5.25. This listing also shows the usage of `SenderReceiver` on line 2.

```
void OnReceiveMessage(MessageHeader &msg) override {
  StageInput data = inputReceiver.Receive(this->GetReceiver(), msg);
```

```
3   if (stageConfiguration.Ordered)
4     ReorderAndProcess(msg, data);
5   else
6     ProcessInput(data, msg);
7 };
```

Listing 5.25 – DSParNode OnReceiveMessage.

When the node has ordering constraints enabled, the method `ReorderAndProcess` is called. The code of this method is in Listing 5.26 and was based on [GHDF18b]. First, the ID of the message is checked. If the ID is in the expected order, then it is processed immediately. If the item is out of order, it is stored in an `std::priority_queue<MessageToReorder>`. The `struct MessageToReorder` stores the header and the data. It also implements `bool operator<` method to compare the header IDs, which is necessary for the `std::priority_queue` to order the messages in ascending order of the IDs. The message with the lowest ID is returned by `orderedMessages.top()` but the ID is checked again. If the ID is out of order, then the process stops. If the ID is in order, it is processed and removed from the queue. The variable `currentMessage` holds the current expected message ID. This ordering algorithm does not work if a node skips a message (does not call `Emit` when it receives an input). Support for these scenarios can be added in the future. This could be supported by implementing a stream control message to notify when a stream item is skipped by a node.

```
1  void ReorderAndProcess(MessageHeader &msg,
2                         StageInput &data) {
3  if (msg.id <= this->currentMessage) {
4      ProcessInput(data, msg);
5      this->currentMessage++;
6      while (true) {
7          if (orderedMessages.size() == 0) break;
8          auto top = orderedMessages.top();
9          if (top.header.id > this->currentMessage) break;
10         ProcessInput(top.data, top.header);
11         orderedMessages.pop();
12         this->currentMessage++;
13     }
14 }
15 else {
16     MessageToReorder<StageInput> msgReorder(msg, data);
17     orderedMessages.push(msgReorder);
18 }
19 }
```

Listing 5.26 – DSParNode Reordering.

The default scheduling is round-robin. The round-robin scheduling tries to emit data to each node in a circular list. The performance characteristics of this method may vary between MPI implementations. We observed that Open MPI makes `MPI_Send` return as soon as the data is written into an internal buffer or when the target rank finishes receiving it.

This is allowed by the MPI Standard as discussed in Section 2.9.1. In programs with small message sizes and a small number of stream items, we observed that the Emitter finishes executing soon after the program starts, but the Workers may still take a long time to finish because `MPI_Send` does not await for the message to be received in the other process. However, this behavior of `MPI_Send` is implementation-dependent [For12]. If the number of buffered messages increases, we Open MPI forces `MPI_Send` to block until more buffer space is available. Other MPI implementations may behave differently.

DSParLib also implements on-demand scheduling. With on-demand scheduling, the Workers of a Farm pattern receive new messages only when they request the Emitter for more work. The Emitter also only emits messages when a Worker sends the demand request. This can improve load balancing if the network is not a bottleneck. It increases network usage because more messages are being sent between the processes. To support on-demand scheduling, we implement a simple demand signaling algorithm. We chose to name it "demand signaling" because it is similar to Akka Streams, which calls this process "demand signaling" as explained in Section 2.8.1. Our implementation is simpler and not as full-featured as Akka Streams. Akka streams uses it to control back-pressure as well. Although we are not concerned with back-pressure in this moment, this algorithm also provides a basic back-pressure mechanism because the `Emit` function (called by the wrapper) will block until a demand signal is received. This is the mechanism that makes the Emitter only send data when necessary as shown in Listing 5.27. The demand signal has a `sender` field that tells DSParLib which rank is demanding more messages.

```
1 void WaitDemandAndEmit(StageOutput &data) {
2     DemandSignal demand = this->WaitForDemand();
3     int targetRank = demand.sender;
4     MessageHeader header = this->GetSender().StartSendingMessageTo(
          targetRank);
5     outputSender.Send(this->GetSender(), header, data);
6 };
```

Listing 5.27 – Wait for demand and emit.

Workers emit the demand signal in 2 moments: when the node starts in the `OnStart` method and right after processing a message on `OnReceiveMessage`. Listing 5.27 shows this process. Each `DSParNode` instance has a `NodeConfiguration` object. When `AskForDemandUpstream` is true, it will send demand to an upstream MPI rank (source). In the case of a Worker node, the source is the Emitter rank. The `Next` method returns the rank of the Emitter node. It is called `Next` because the `sources` variable of type `dspar::CircularVector` is a circular iterator: once it reaches the end of the list, it restarts from the beginning. In the Farm pattern, the Worker has only one source, therefore, `Next` just returns the same value over and over. The round-robin scheduling uses this `class` to iterate over the Worker ranks

repeatedly. Listing 5.28 shows the `OnStart` method, but the code in `OnReceiveMessage` is similar regarding on-demand scheduling, therefore, we do not show it here.

```
1  AfterStart OnStart() {
2      stage.Start();
3      stage.Produce();
4      if (stageConfiguration.AskForDemandUpstream)
5          this->SendDemand(sources.Next());
6      if (sources.Count() > 0)
7          return AfterStart::ReceiveMessages;
8      else
9          return AfterStart::StopNode;
10 };
```

Listing 5.28 – OnStart method.

5.10 Planning node allocation

Before starting the nodes, DSPARLIB needs to know which node (a `DSParLife-cycle` instance) must be ran by a given MPI rank. This section describes the process of allocating DSPARLIB nodes and ranks of Open MPI.

The programmer can set the parallelism degree of a Farm pattern. This is shown in Listing 5.21. The simplest Farm instantiation with only 1 Worker replica needs at least 3 processes: 1 process for the Emitter (E), 1 for the Worker (W), and 1 process for the Collector (C). If the programmer sets N processes for the parallelism degree on a Farm, then $N + 2$ processes are needed. It is possible to visualize this specific node configuration for MPI processes in Figure 5.5.

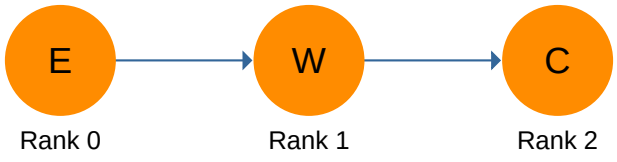

Figure 5.5 – Early MPI DSPARLIB layout proposal, single Worker.

The latest version of DSPARLIB places the Emitter and Collector as neighbors (Emitter on rank 0 and Collector on rank 1). The layout of this approach is shown in Figure 5.6. This approach enables testing different degrees of parallelism easier. In our `hostfiles`, we are able to isolate the Emitter and Collector by setting their ranks on a particular cluster node. The first approach, represented in Figure 5.5 would require generating a different `hostfile` (process allocation strategy) for each degree of parallelism, because the rank of

the Collector varies according to the total number of Workers. This approach does not work when testing Pipeline compositions inside the Farm, but we do not use such composition in the DSParLib benchmarks (Section 5.11).

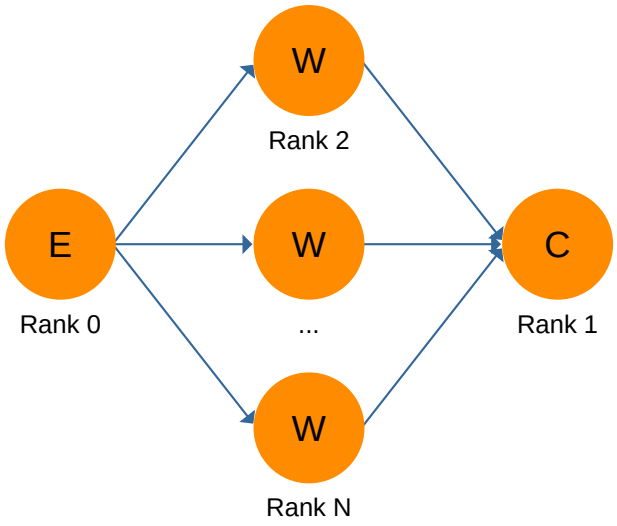

Figure 5.6 – Current MPI DSParLib layout, multiple Workers.

If the default Open MPI process allocation is used, rank 0 and 1 will be placed in the same cluster node equipped with multi-core processor. If the Emitter and Collector are network or disk I/O intensive, both processes may have degraded performance since they would compete for limited resources. It can be argued that the process allocation strategy of DSParLib may even increase the chances of having this problem, if no proper hostfiles are provided.

Each parallel pattern in DSParLib implements the `AbstractPipelineElement class`. This `class` informs the `Pipeline class` how the processes will be allocated. Specifically, each `AbstractPipelineElement class` has information about its inputs and outputs and how many processes are needed in total. These are communicated by rank offsets. A Farm, for instance, will communicate its input offset as 0 and output offset as 1. If there are other `AbstractPipelineElement` instances before the Farm, these offsets will be used to calculate the actual ranks where the Farm will be positioned. For instance: if there are 2 previous `PipelineStage` instances before the Farm, the input offset 0 will be summed to 2. Therefore, the Farm will have its Emitter on rank 2. The output offset 1 will be summed to 2, and therefore the Farm Collector will be 3. The offset interface is required when implementing `AbstractPipelineElement`, and is shown in Listing 5.29.

```
1 class FarmPattern: public AbstractPipelineElement {
2 int Start(MPI_Comm comm, int startingRank,
3             std::vector<int> inputRanks,
4             std::vector<int> outputRanks) {
```

```
5                   ...
6 }
7 ...
8 int GetTotalNumberOfProcessesNeeded() override {
9   return 1 + workerReplicas + 1;
10 };
11 std::vector<int> GetInputOffsetRanks() override {
12   return { 0 };
13 }
14 std::vector<int> GetOutputOffsetRanks() override {
15   return { 1 };
16 }
17 };
```

Listing 5.29 – AbstractPipelineElement offset methods.

Finally, `Pipeline` identifies the current process rank and then identifies which `AbstractPipelineElement` will run in the given rank. Once the node is identified, `Pipeline` calls the method `Start` of the `AbstractPipelineElement`. This method is also shown in Listing 5.29. Additionally, the `FarmPattern` checks whether the given rank is an Emitter, Worker or Collector.

In Figure 5.7, we show how classes in DSPARLIB relate to each other in a class diagram. `DSparNode` inherits from `DSParLifecycle`. `FarmPattern` and `PipelineStage` create instances of `DSparNode` and run them based on their process rank. Therefore, we represent them as an aggregation. The `FarmPattern` *should* also aggregate `Wrapper` and `SenderReceiver`, but in reality they are just a container for these objects. When the Pipeline runs, the `FarmPattern` just passes the wrappers to the new instance of `DSParNode`.

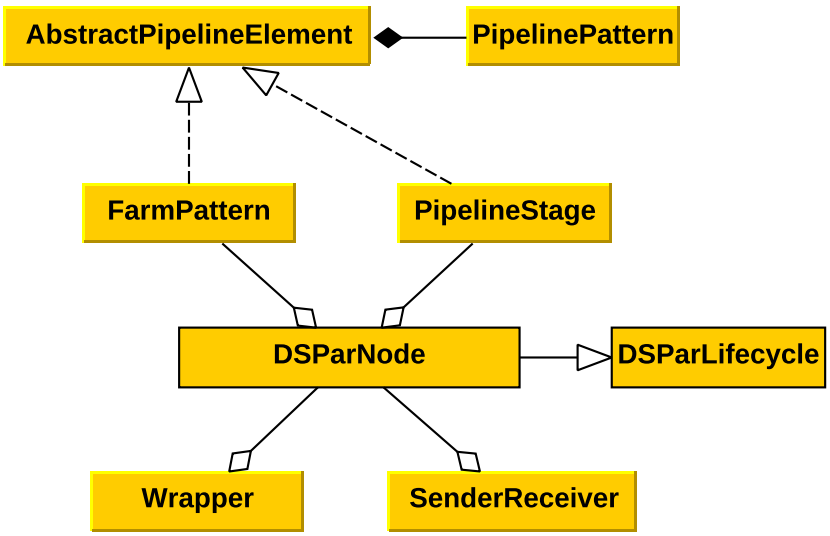

Figure 5.7 – Class relationships in DSPARLIB.

5.11 Performance tests

In this section, we present performance experiments. Applications written with DSPARLIB are compared with the same programs written with MPI directly, with no abstractions. The Mandelbrot and Prime numbers are data-parallel instead of stream processing applications. However, data parallel applications can also be expressed in a streaming fashion [Gri16]. We provide a brief description for all the used benchmark applications:

1. **BZIP2** is a compression library and program used in many Linux distributions. The MPIBZIP2 is a parallel version for distributed-memory architectures using Open MPI, which was based on the Pbzip [Gil04]. The parallelism strategy of MPIBZIP2 is master-worker, where the master splits the file in smaller blocks, and workers compress the blocks. The master collects the results and writes the final result to disk [GF17]. In DSPARLIB, the parallel implementation uses a Farm pattern with ordered constraints. The Emitter splits the file, the Workers compress the blocks, and the Collector writes the result to disk.

2. **Lane detection**: is a program that detects the road lanes on a video stream using OpenCV [GHDF17]. The computation is CPU intensive. The result is a new video stream where the road lanes are drawn frame-by-frame. We used a pre-existing handwritten MPI application [VRJ+20] to compare the performance against DSPARLIB, although we had to fix bugs in the implementation. In both versions, parallelism is achieved using stream parallelism. The MPI version implements a Farm-like[3] computation with ordering constraints implemented manually, while the DSPARLIB version uses the Farm pattern.

3. **Face recognition**: is a program that performs face recognition in a video stream using OpenCV [GHDF17]. A machine learning model is trained at the beginning of the program and is used to detect a specific face in the video. The result of the stream is a new video with visual indications where the faces were detected. There were no pre-existing handwritten MPI versions for this application, therefore, we developed it using a Farm-like strategy with ordering. The DSPARLIB version uses the Farm pattern with ordering enabled.

4. **Mandelbrot**: Computes a black and white image of the Mandelbrot Set [Gri16]. It does not write an image to the disk. Instead, it just computes it to show on the screen. We wrote the MPI version ourselves using a Farm-like strategy without ordering. We do not use a pre-existing Mandelbrot MPI application because we want to ensure the same

[3]The application implements the components of the Farm (Emitter, Worker and Collector) directly in the application code.

sequential code is used in all versions. The DSPARLIB version uses the Farm pattern with ordering disabled.

5. **Prime numbers**: Counts the number of prime numbers in a given range [GDTF15]. In this benchmark, we used the range 0 to 1.2 million. We used a pre-existing handwritten MPI program [GF17] that performs the same computation using a Farm-like strategy without ordering. The DSPARLIB version uses the Farm pattern with ordering disabled.

Our tests measured the throughput of the applications compared to the MPI versions. We also compare the round-robin and on-demand schedulers when they are available. We do not change the way the original applications measure the throughput. The throughput is measured by the formula *items*/*time*, where *items* is the total number of stream items transmitted and *time* is the execution time of the application. PBZIP2 compression is measured in *MB*/*second* using the size of the original file and the execution time.

All tests ran on LAD (Laboratório de Alto Desempenho) at PUCRS, in the Cerrado cluster. Each node has 2 Intel(R)Xeon(R) CPU E5-2620 v3 @ 2.40GHz (12 cores, 24 threads) with 24GB of RAM memory. The MPI implementation is Open MPI 1.4.5. The operating system is Ubuntu 16.04 64 bits with kernel 4.4.0-146-generic. The machines are connected via Gigabit Ethernet and InfiniBand QDR 4x (32GBit/s), and Open MPI automatically uses InfiniBand when available [4]. The application is compiled on GCC 9.3.0 using -O3 optimizations. OpenCV version is 2.4.13. We allocated the nodes 06 to 13 in the cluster.

We tested the output of all applications to check whether the results using DSPARLIB are the same as the sequential version. For instance, the Lane Detection and Person Recognition applications generates a video file. We compare the MD5 hash of the DSPARLIB result and the sequential result to certify that they are the same. For the PBZIP2 application, we decompress the compressed files produced by mpibzip2 and our version of PBZIP2 to check whether the decompressed files have the same MD5 hash. The Prime Numbers application prints the total amount of prime numbers in a range, therefore, we just check whether the sequential and parallel versions produce the same result. For the Mandelbrot test, we create a version that generates image files, and again we used an MD5 hash program to check the result.

With Open MPI, the user can configure how the processes will be allocated in the cluster using `hostfiles` and flags in the `mpiexec` command. Depending on how they are allocated, the performance characteristics might change. Therefore, we tested different process allocation configurations. We also set the maximum number of processes on each node to avoid hyper-threading. This way, each node runs 12 processes on 12 physical cores. We have 3 process allocation strategies:

1. **Isolate Collector and Emitter, fill node first** (`fillnode-ec-isolate`): This strategy allocates all available cores in the cluster node first, and will only use another node if

[4] `https://www.open-mpi.org/faq/?category=tcp#tcp-auto-disable`

the current node is fully allocated. The Collector and Emitter are allocated in different nodes (an entired node for them).

2. **Collector and Emitter in the same node, fill node first** (`fillnode-ec-samenode`): This strategy allocates all available cores in the node first, and will only use another node if the current node is fully allocated. The Collector and Emitter are isolated from the Workers, but they are placed in the *same node*.

3. **Isolate Collector and Emitter, round-robin allocation** (`roundrobin-ec-isolate`): This strategy allocates one process in each available node in a round-robin fashion. For instance: the Worker 2 is placed in node #2, Worker 3 is placed in node #3, and so on. The Collector and Emitter are isolated: each one runs in a separate node.

Our experiments were configured such as follows. For each application, we test different degrees of parallelism, from 4 to 72. We start at 4 and increment the parallelism degree by 4 after testing it 30 times for each process allocation strategy. This process is also repeated for each of the MPI configurations explained before. The process allocation strategy that had the best results for most of the applications was `roundrobin-ec-isolate`. Therefore, we will focus on showing results for this specific strategy. When necessary, we show performance results using different process allocation strategies.

5.11.1 Lane detection performance

Figure 5.8 shows the throughput in frames per second (FPS) in the Y-axis, and the number of Farm Workers (MPI processes) in the X-axis, also called degree of parallelism. The lines have error-bars to show the standard deviation. Four applications are being tested: Two pre-existing MPI applications that implement on-demand and round-robin scheduling. The `mpi-ond` label refers to the MPI on-demand version and `mpi-rr` refers to the MPI round-robin version. Likewise, `dspar-ond` and `dspar-rr` are the DSPARLIB on-demand and round-robin versions, respectively. The 4 applications require ordering in the Collector.

Our peak throughput was 214.55 frames per second with 32 Workers with DSPARLIB on-demand. The on-demand MPI version has similar performance until 28 Workers. The difference becomes more evident as more processes are used. At 32 Workers, the bottleneck in disk I/O starts to show. The program stop scaling.

In order to evaluate the disk impact, Figure 5.9 shows the performance of the application without writing the resulting video to disk. The performance gap between on-demand and round-robin scheduling becomes more evident. At 72 Workers, the program is still speeding up and computes 566.28 frames per second. Although we do not have results for MPI versions, it is expected a similar behavior.

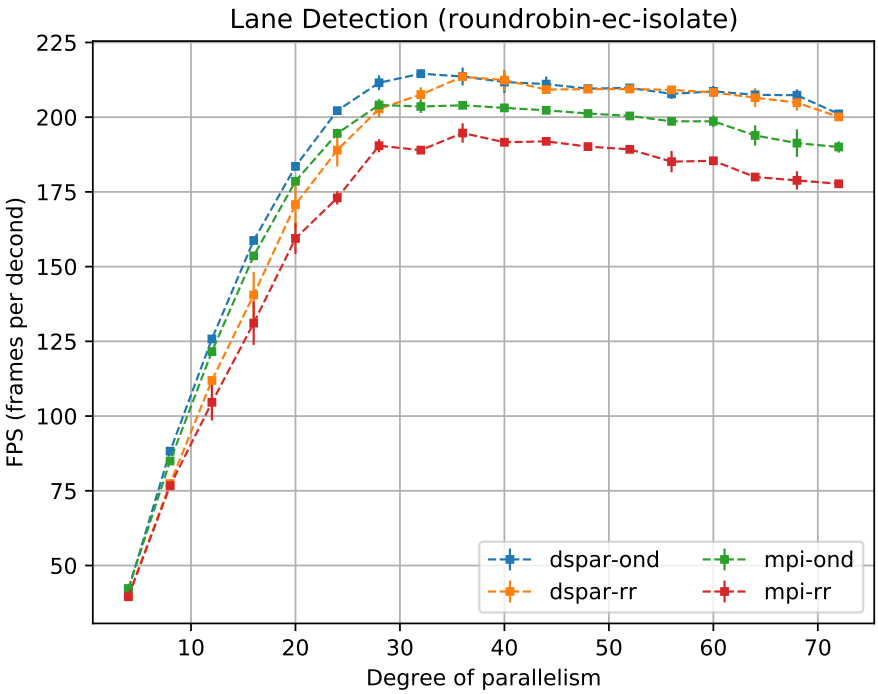

Figure 5.8 – Lane detection throughput in frames per second.

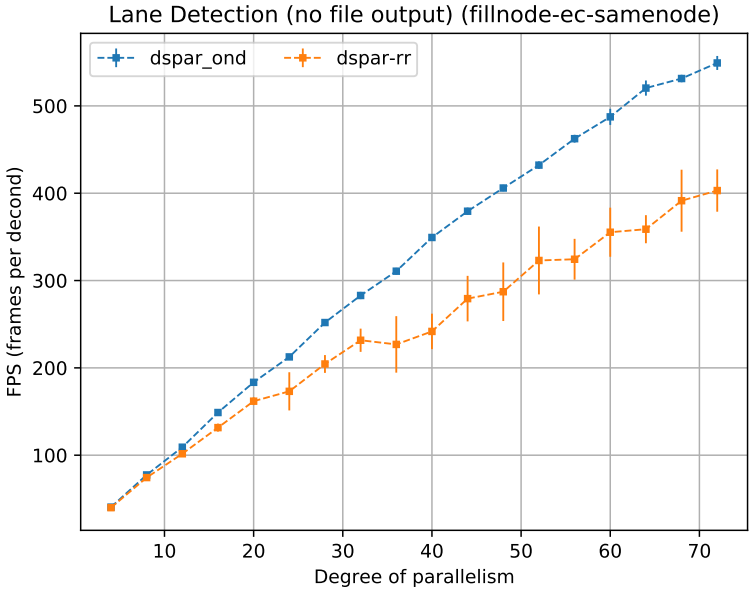

Figure 5.9 – Lane detection throughput in frames per second, no I/O.

The results in Figure 5.8 show that the DSPARLIB version is faster than the MPI version. We expected the MPI version to be faster because it can perform serialization more efficiently. Both versions, however, need to send 3 messages to transmit a video frame. The MPI version uses a header message of size `MPI_INT` (4 bytes) to communicate the frame number and 2 messages to send the frame data. DSPARLIB sends a bigger header message with 24 bytes and then 2 messages to send the frame data. Although DSPARLIB consumes more network bandwidth, it was not an issue if compared to the MPI version.

5.11.2 Face recognition performance

Figure 5.10 shows the throughput in frames per second (FPS) of the face recognition application. The application processes 450 video frames of size 640x480 pixels. In this test, frames are more expensive to compute, and we never reach a disk I/O bottleneck. At 72 Workers, the application is still scaling well in all versions. The MPI on-demand and DSPARLIB on-demand versions have similar performance, but DSPARLIB is slightly faster.

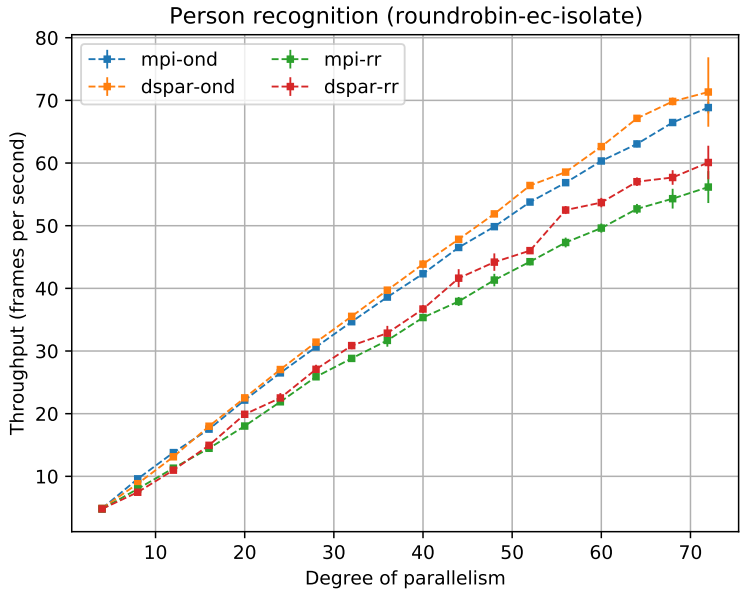

Figure 5.10 – Face recognition throughput in frames per second.

There is a possibility of bias in this test since we wrote the MPI versions ourselves. If a pre-existing MPI application exists, we would use it, but it was not the case for the face recognition application. DSPARLIB and MPI are sending and receiving similar messages, and the MPI version has an advantage since the header messages (message ID) are smaller. This is similar to lane detection, where header messages are smaller, but DSPARLIB is still faster. Therefore, there might be room for improvement in the MPI version.

5.11.3 BZIP2 performance

Instead of writing a new MPI streaming version of BZIP2, we use the already existing program mpibzip2. However, mpibzip2 is a master-worker application instead of a streaming application. Since the DSPARLIB version isolates the Emitter and Collector, we isolate the mpibzip2 master in a separate cluster node as well. We compress a 512MB MP4 video file. Since the MP4 file format is already compressed, the compression ratio achieved by the BZIP2 algorithm is small. We also slightly modified the mpibzip program to allow the

user to specify a different output file location. At the time of writing this document, the LAD cluster had some performance issues when writing files to the NFS. This change allows the benchmark to use the `/tmp` directory in a local filesystem instead of a network filesystem. We show both versions: with and without writing to the `/tmp` file. We did not change any compression or scheduling code in mpibzip2, only the output directory.

Figure 5.11 shows the performance of all versions. In mpibzip2, the master process is responsible for reading the data from the uncompressed file, and write the compressed data to the output file. This causes a networking and synchronization bottleneck since the master is responsible for a significant part of the network I/O and resources management. At 44 processes, the mpizip2 master reaches its maximum throughput. The DSPARLIB version has the advantage of having an extra MPI process for the Collector. In this test, the Collector is isolated in another cluster node. Consequently, I/O work is divided between two machines, and the performance continues to scale as more processes are added.

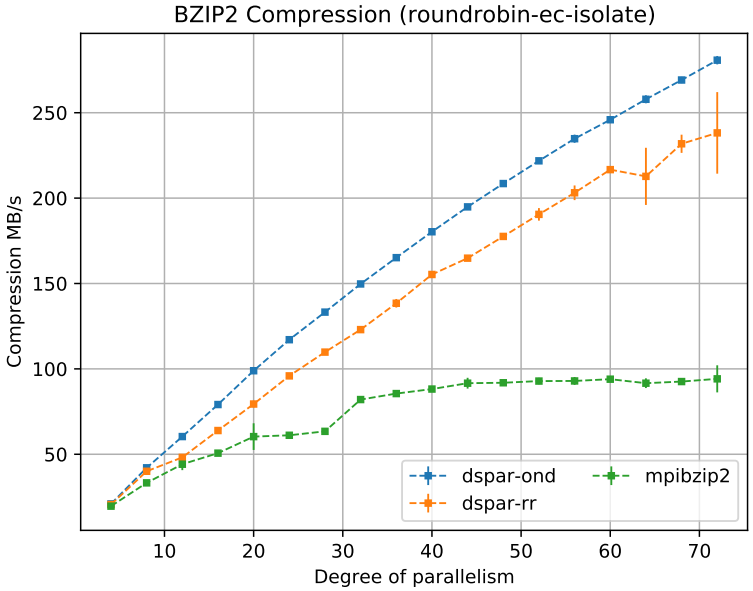

Figure 5.11 – BZIP2 compression throughput in MB/s.

In Figure 5.12 we show a test without changing the mpibzip2 program. We also configured the programs to write in the NFS file. The mpibzip2 is even slower than the previous benchmark on Figure 5.11, and has similar performance to DSPARLIB. Due to the networking issue, all versions got slower, thus requiring the modification to mpibzip2. Notice that DSPARLIB reaches only 80MB/s with 72 processes. while in Figure 5.11 we show that DSPARLIB on-demand reaches 275MB/s.

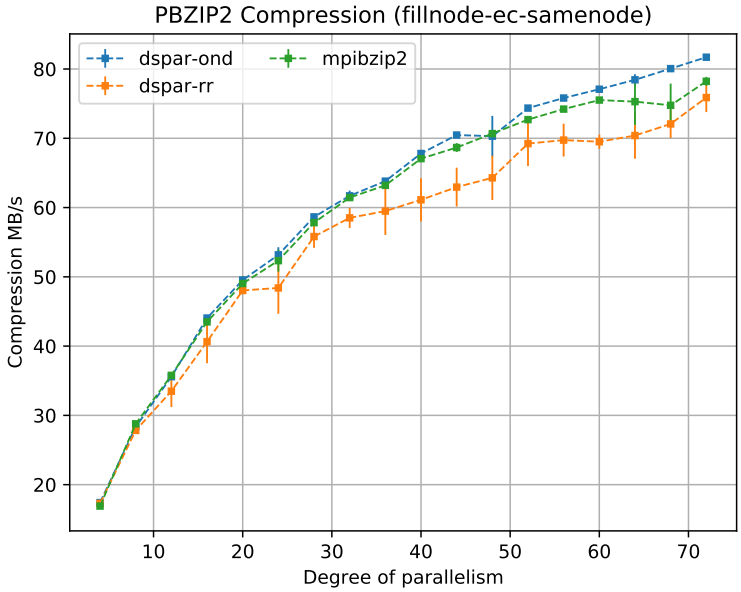

Figure 5.12 – BZIP2 compression throughput in MB/s, writing to an NFS filesystem.

5.11.4 Mandelbrot Set performance

The Mandelbrot application computes a 2000x2000 pixel image of the Mandelbrot set, using 10,000 iterations. The performance difference between the versions is significant as shown in Figure 5.13. The MPI program was written by ourselves and there might be room for improvement. The computation is done on a line-by-line basis in all versions, which is also similar to the shared-memory parallelism strategy proposed in [Gri16].

The differences between DSPARLIB and MPI in this case is large. This is a very small workload, and minimal measurement differences result in significant differences in the throughput graph. When we run the DSPARLIB program with dynamic process allocation, DSPARLIB creates a new communication group containing all the spawned processes. However, for the MPI versions, all processes live in `MPI_COMM_WORLD`. We added extra `MPI_barrier` to synchronize the start and finishing points of all processes in order to get an accurate measurement, but still observe a difference.

In fact, all benchmarks in this chapter show slightly better results for DSPARLIB. We could not prove that dynamic process allocation strategy provides better performance, because our experiments were not made to stress this fact. Although the results in this chapter suggest that Open MPI may perform some optimizations when dynamically allocating processes, further investigations on this aspect can explain precisely which are the main reasons that DSPARLIB is better than Open MPI.

Therefore, we also ran a test with a larger workload. The results are shown in Figure 5.14. The performance between MPI and DSPARLIB is similar, as the small measurement differences are less significant compared to the total test time. In this test, lines

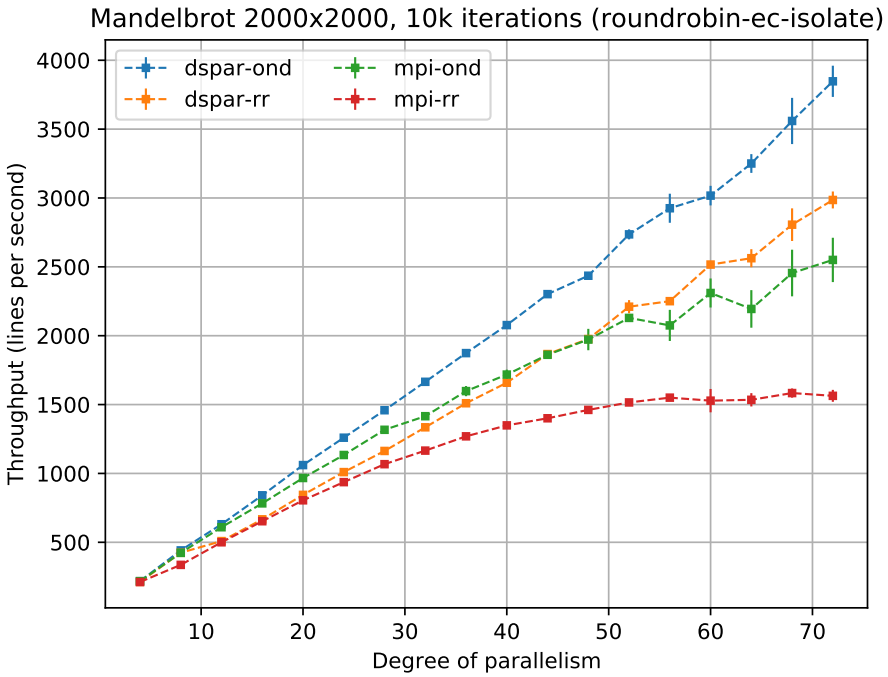

Figure 5.13 – Mandelbrot Set throughput, size 2000x2000.

near the middle of the Mandelbrot image are more expensive to compute than other regions, causing load balancing issues. The on-demand scheduling is better in this situation, resulting in better performance for the DSPARLIB and MPI on-demand implementations.

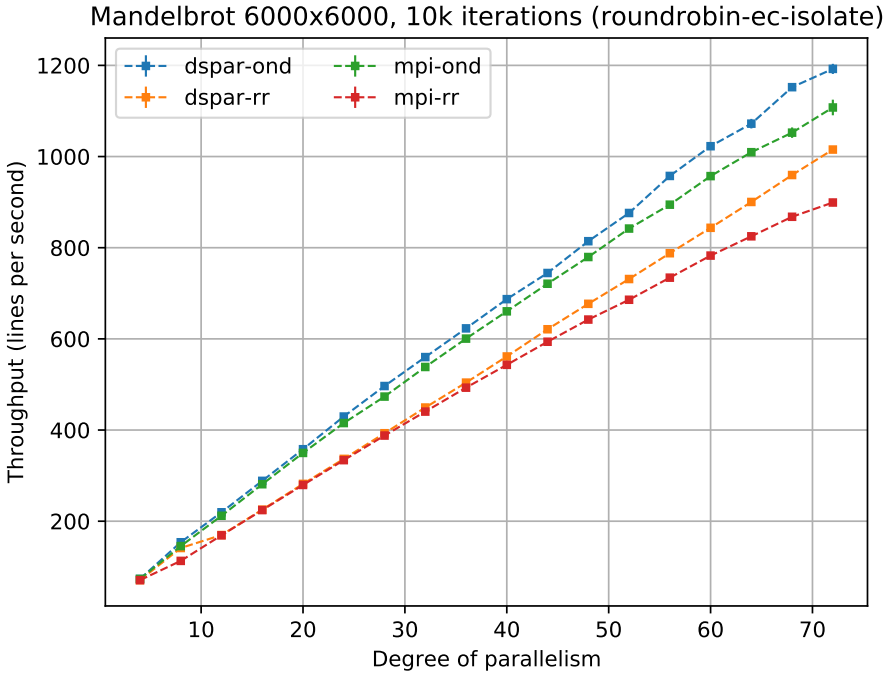

Figure 5.14 – Mandelbrot Set throughput, size 6000x6000.

5.11.5 Prime numbers performance

In this benchmark application, the performance difference is bigger than in the Mandelbrot. This is shown in Figure 5.15. The difference between the on-demand and round-robin versions is due to load balancing issues. The program checks all numbers between 1 and 1.2 million. An interesting characteristic of this test is the computation involved in checking whether a number is prime or not. Different numbers have different computation times, which results in load balancing issues.

The round-robin scheduling is not efficient in this scenario. This is because even numbers (trivially proven non-prime using `number % 2 == 0`) are always delivered to the same Workers. These Workers end up being starved, because their computation is trivial. Meanwhile, other works have to perform more work. Furthermore, if there are too many messages enqueued by MPI in a Worker, the Emitter must wait the Worker finish its current computation since `MPI_Send` is a blocking operation (assuming our understanding of the MPI standard `MPI_Send` is correct in Section 5.9). While the Emitter is blocked, other Workers must wait to receive new data items and stay idle. Using the on-demand scheduler, Workers ask for more work as soon as they are ready.

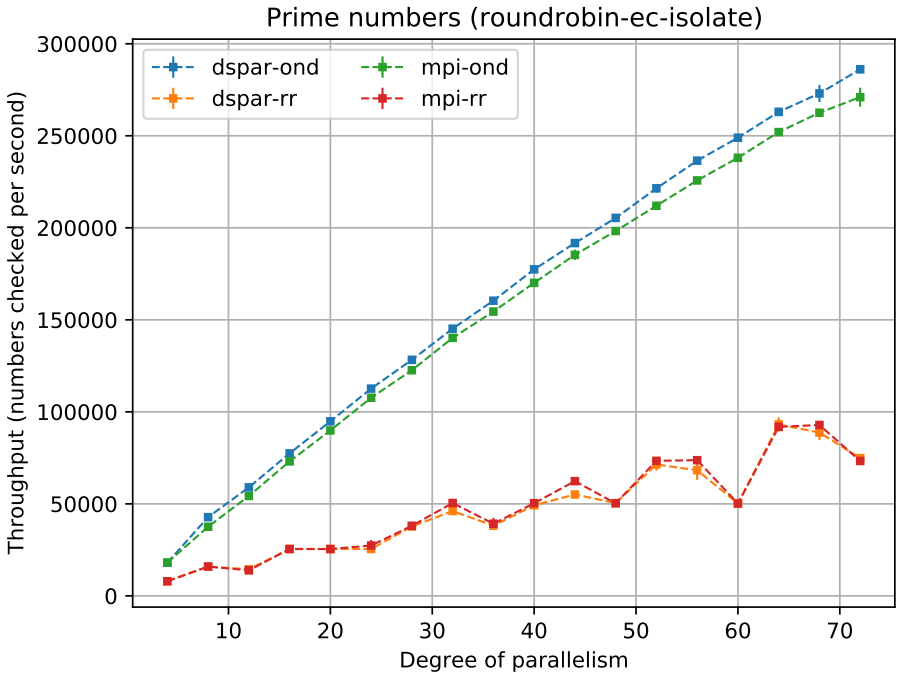

Figure 5.15 – Prime numbers throughput in numbers checked per second.

Another detail that can be observed in Figure 5.15 is the unstable behavior for the round-robin versions. To understand why it happens, consider the sequence of numbers a given process will receive under different degrees of parallelism. The Worker #2 will first receive the number 3. If there are 60 Workers, the sequence of numbers the Worker #2

will receive is 3, 63, 123, 183, 243, and so on. With 64 Workers, the sequence changes: it becomes 3, 67, 131, 195, 259, and so on. The first sequence never gives the process #2 a number divisible by 5, which can be quickly proven to be non-prime. Meanwhile, the Worker #3 will always receive even numbers, which are trivial to prove they are not prime. The level of difficulty is unevenly spread over Workers depending on the sequence. Each parallelism degree will result in a different sequence and different performance characteristics. The on-demand scheduler changes this pattern in a non-deterministic way and distributes the difficulty level among Workers.

5.11.6 Comparison of the process allocation strategies

The round-robin MPI process allocation (`roundrobin-ec-isolate`) resulted in the best performance for most of the applications. In this subsection, we will compare the performance results for the process allocation strategies. All applications are using DSPARLIB with on-demand scheduler shown in the previous subsections. Although `roundrobin-ec-isolate` resulted in best performance, it needs 2 extra cluster nodes. For brevity, we do not show all applications in this chapter, though they have similar behavior.

The Mandelbrot's results is shown in Figure 5.16. Observe that the `roundrobin-ec-isolate` is the best process allocation strategy across all degrees of parallelism. Since Workers are allocated using round-robin, this can reduce the competition for network resources among processes in the same machine. The difference between `fillnode-ec-isolate` and `fillnode-ec-samenode` is not significant, as can be seen by the error-bars.

The lane detection benchmark is shown in Figure 5.17. This program achieves the best performance at 32 Workers with `roundrobin-ec-isolate`. After 36 processes, the best performance is achieved with `fillnode-ec-isolate`, but it does not perform better than `roundrobin-ec-isolate` with 32 Workers.

For the prime numbers benchmark, the round-robin allocation is significantly better for up to 56 processes, as shown in Figure 5.18. This might be due to the way that `roundrobi-/isolate` is balancing the network load among more cluster nodes. The other configurations concentrate the load in a small number of nodes first. For more than 64 Workers, `fillnode-ec-isolate` offers better performance than `roundrobin-ec-isolate` , achieving more than 300,000 numbers checked per second. There is a small variation in performance for `fillnode-ec-isolate` and `roundrobin-ec-isolate`, especially for more than 64 processes.

For the face recognition, we do not show the performance plot for brevity. The the results are similar to the others: `roundrobin-ec-isolate` strategy is the best configuration for most degrees of parallelism. At 72 processes, the `fillnode-ec-isolate` wins by a small margin and the `roundrobin-ec-isolate` standard deviation is significant.

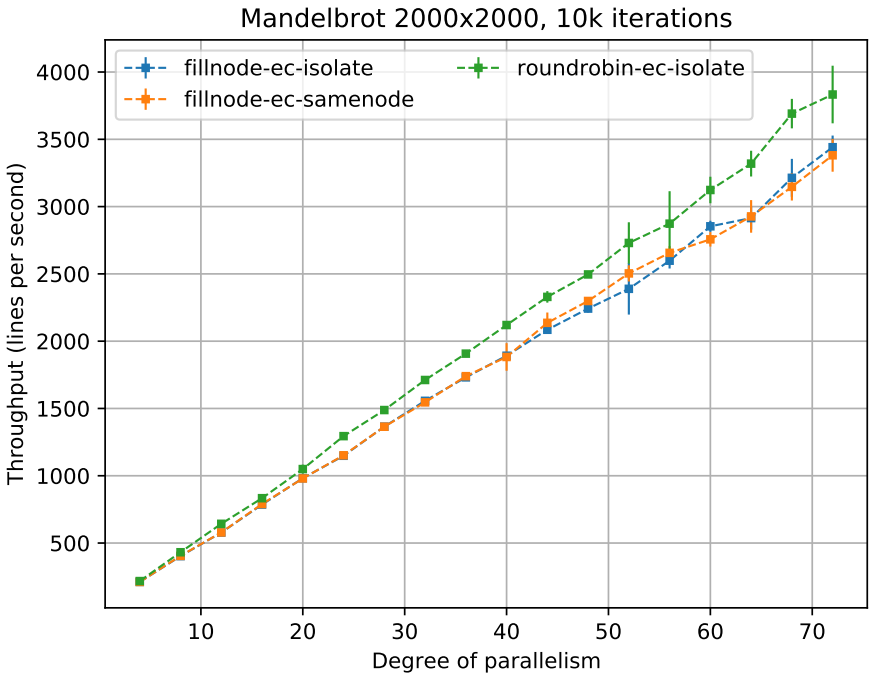

Figure 5.16 – Mandelbrot benchmark using different process allocation strategies.

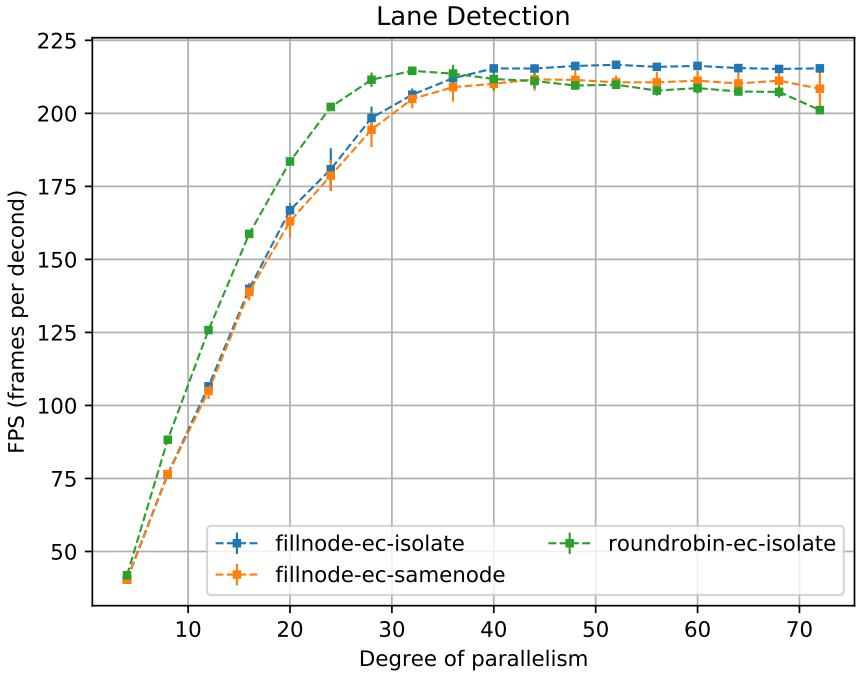

Figure 5.17 – Lane detection benchmark using different process allocation strategies.

5.11.7 Overall performance outcomes

In this section, we presented performance results for DSPARLIB compared to MPI versions. In all tests, DSPARLIB performs faster or as fast as handwritten MPI programs in the Cerrado cluster at LAD - Laboratório de Alto Desempenho PUCRS. Specifically, we

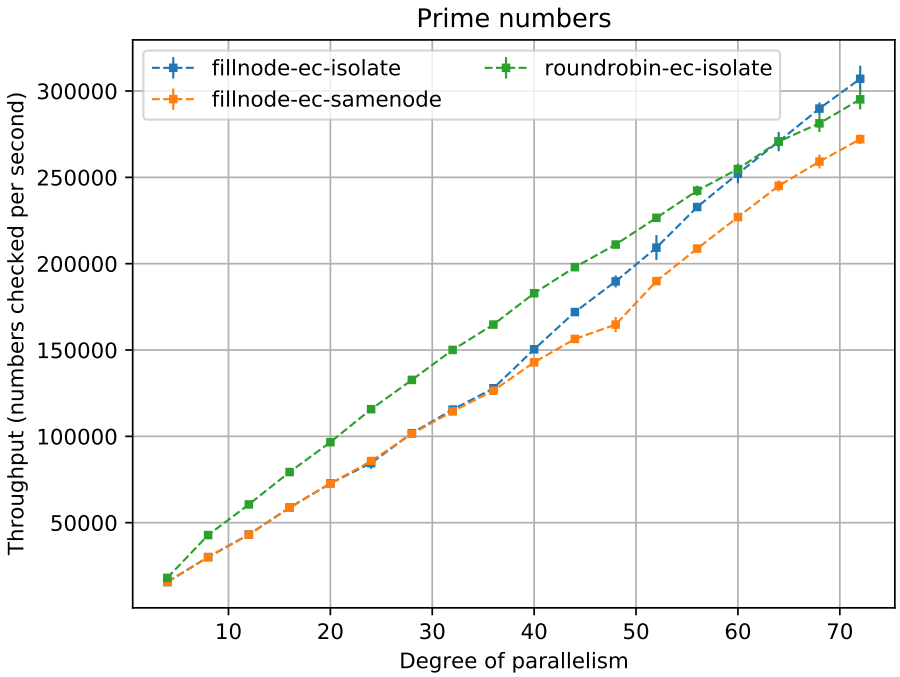

Figure 5.18 – Prime numbers benchmark using different process allocation strategies.

used the range of nodes `Cerrado06n` to `Cerrado13n`. Further testing could be done in other clusters or different machines in the Cerrado cluster. Specifically, it would be interesting to see whether DSPARLIB still has a performance advantage on a cluster connected via Ethernet instead of InfiniBand.

In some cases, DSPARLIB was faster than handwritten MPI programs with the same work scheduling algorithm. However, some results are *drastically* better. In Section 5.11.4 we described some variables in the tests that are yet to be investigated in more depth, to better understand why the measured performance is better. Another hypothesis is that the use of `MPI_BYTE` disables endian checking and conversion in Open MPI, as explained in Section 5.7. This could provide a performance advantage because Open MPI would perform less work. Meanwhile, all MPI handwritten applications use appropriate data types, such as `MPI_INT` or a user-defined `MPI_Datatype`. In this case, Open MPI may perform more work to encode and decode the data. However, we have not tested this hypothesis.

Furthermore, we show that the `roundrobin-ec-isolate` process allocation strategy had the best performance. We expected this to be the case for most degrees of parallelism. Moreover, when analyzing the performance of the `fillnode` configurations, separating the Emitter and Collector in separate cluster nodes seems to have limited effect, and requires more computing resources.

The round-robin scheduler is the default, because it is the simplest scheduler in DSPARLIB. For programmers that want to use DSPARLIB, they can try using the on-demand scheduler to increase performance. This scheduler has consistently better results compared to the round-robin scheduler for unbalanced workloads, being also easy to enable. For the

process allocation strategy, we suggest the programmer to test different configurations on their clusters, as each cluster may perform differently than ours.

5.12 Final Remarks

In this chapter, we described DSParLib, a new library for stream parallelism on MPI. Application programmers that are familiar with libraries such as FastFlow [ADKT17] or TBB (Thread Building Blocks) [Rei07] will find many similarities in DSParLib.

DSParLib can make distributed stream programming easier when compared to handwritten MPI. Furthermore, writing MPI programs can be error-prone. During performance testing, we had to fix bugs in handwritten MPI C++ programs due to memory and logic errors when controlling the stream. DSParLib handles the stream correctly and does not expose low-level details to the application programmer.

Additionally, DSParLib also provides on-demand scheduling to improve load balancing when networking is not a bottleneck. This type of scheduling requires effort to implement in handwritten MPI. With DSParLib, this can be easily enabled with a simple configuration on a Farm pattern. Therefore, application programmers have easier access to features that increase performance without having to deal with low-level implementation details.

Although the high-level DSParLib API does not offer all stream parallelism functionality implemented in other libraries (such as feedback loops and arbitrary computation graphs of FastFlow), it covers many use cases. Furthermore, the internal structure of DSParLib allows implementing more advanced features in the future.

To increase performance, some optimizations are still left to be implemented, such as reducing the message header size or sending the header together with the data in a single `MPI_Send` call. This can reduce the total number of MPI messages, which is important in clusters with limited networking capabilities. DSParLib currently lacks error handling and fault tolerance. Future works could implement ULFM (User-level fault mitigation) [BBH+13] on DSParLib, which is supported by Open MPI. The delivery guarantees of DSParLib are only as strong as the underlying MPI implementation. In the case of Open MPI, delivery of messages is not guaranteed[5].

[5] `https://www.open-mpi.org/faq/?category=ft#dr-support`

6. HIGH-LEVEL DISTRIBUTED STREAM PARALLELISM

Our goal is to extend the SPAR language [Gri16] to be used for distributed-memory architectures [GDTF17]. This chapter aims to describe the investigations, principles and implementations to make it possible for SPAR without changing the level of abstraction. In the previous chapter, we have described the DSPARLIB library for distributed stream parallelism. This chapter will describe how SPAR was modified to generate code for DSPARLIB in the compiler and language levels. To distinguish the shared and distributed-memory implementation, we will refer to "SPAR-multicore" or "multicore version" for the FastFlow code generation targeting shared-memory architectures. For distributed-memory architectures code generation with DSPARLIB, we will refer to "SPAR-distributed" or "distributed version". When we refer to processes executed by both versions, we will refer to them simply as SPAR.

Section 6.1 will describe the parts of SPAR language that will were changed to support serialization of dynamically allocated arrays. Section 6.2 will revisit the transformation rules and show that they have not been changed at all. We also address limitations in DSPARLIB API and how the code generation process was changed.

Section 6.3 presents details on how the code generation was implemented as well as refactors in the existing code. In section 6.5, we ran benchmarks of the code generated by SPAR-distributed and compared the results with programs written with DSPARLIB.

Moreover, we demonstrate that SPAR-distributed can be also used in more complex programs by parallelizing the Ferret application in Section 6.6, where we explored other challenges with serialization and how they can be alleviated with the C++ STL.

We assess the programmability of SPAR-distributed in Section 6.7 and present a table comparing the SLOC of applications in multiple versions (serial, MPI, DSPARLIB, and SPAR-distributed). Section 6.8 shows where SPAR-distributed still needs to be improved, exposing limitations that can be addressed in the future. Finally, we conclude this chapter in Section 6.9 by summarizing our findings.

6.1 New extensions to the SPAR Language

In this work, we intend to preserve the high-level programming abstraction of the language. However, the distributed-memory environment rises new challenges. In the previous chapters, we have described the problem of serialization. The DSPARLIB allows the user to specify how the data is sent to another process.

Stage communication on SPAR is expressed with the `spar::Input` and `spar::Output` annotations. The syntax for both annotations are the same. It accepts a list of variables that will be sent to another process. In SPAR-multicore it is trivial to communicate data

between stages, since it is a shared-memory architecture where we can just comunicate with a queue.

In SPAR-distributed, the actual bytes of the data must be copied to the network. In case of arrays, we need also inform the data sizes. Listing 6.1 shows the new `spar::Input` and `spar::Output` syntax in line 10. Although the new syntax is `array[size]`, it also supports arrays of many dimensions. Using this syntax, contiguously-allocated arrays (including C-style `char*` strings) can be serialized. This will not work for complex data structures with data layouts that are not understood by DSPARLIB. We will present a use case of SPAR-distributed in the Ferret application later (Section 6.6), showing how we serialized complex Ferret data structures. We will also present the shortcomings of the strategy we used.

```
1
2  [[spar::ToStream]] while (true) {
3      int size = 42;
4      char* array = new char[size];
5      char** array2d = new char*[size];
6      for (int i = 0; i< size; i++) {
7          array2d[i] = new char[10];
8      }
9      ...
10     [[spar::Stage, spar::Input(size, array[size],
11         array2d[size][10])]] {
12         ...
13     }
14 }
```

Listing 6.1 – New input/output syntax.

SPAR parses the `spar::Input` and `spar::Output` syntax using CINCLE's parsed tree structure. Currently, SPAR-multicore expects the first syntax node after the annotation name to be a list of identifiers, where the number of child nodes of each expression (each identifier) is 1 and it must be an identifier expression. Listing 6.2 shows the relevant part of the code. The number of child nodes is checked on line 1.

```
1 if (postfix_expression->childs_n == 1) {
2   cincle::node *id_expression = return_node_onlyleft_down(
       postfix_expression, NODE_TYPE_id_expression);
3   if (id_expression == NULL) {
4     std::cerr << "NULL ERROR when searching id_expression with:
         return_node_onlyleft_down(...);" << std::endl;
5     return false;
6   }
7   ...
8 }
```

Listing 6.2 – Parsing identifiers.

When the user enters the new syntax, each expression has 4 nodes. Figure 6.1 presents a visualization of the node tree for arrays with the size expression. The first `postfix_expression` node contains the name of the variable. The nodes `lbracket_token` and `rbracket_token` are the array bracket characters `[]`. The `expr_or_braced_init_list` will contain the size of the array as a literal expression (a constant number) or an identifier expression (a variable containing the size).

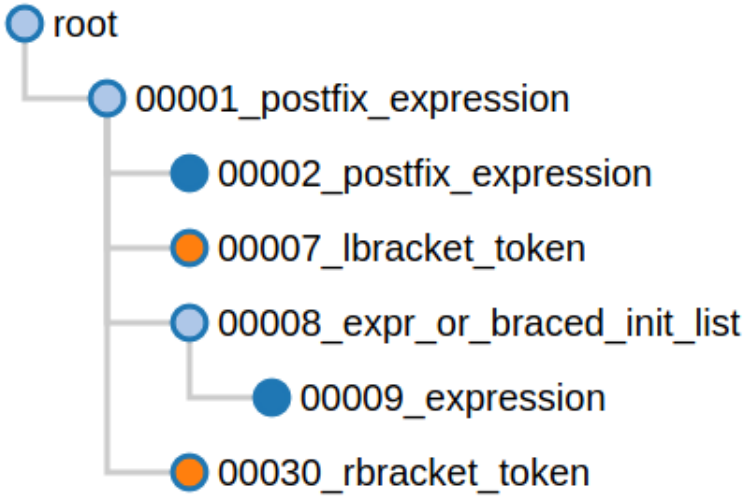

Figure 6.1 – Array node tree for `array[size]`.

If the programmer enters a 2D array `array[dim1][dim2]`, the tree needs to be navigated recursively. The first postfix expression will contain the first array. In Figure 6.2, the bracket tokens appear inside the first postfix expression too, which has 4 child nodes. We can extract the array data using a recursive method. Listing 6.3 shows a simplified version of the parsing method. In lines 4 to 7 we declare variables representing what the nodes are. Then, we call the method recursively on the postfix expression in line 10. We then check whether the node is a literal or identifier expression and add it to the `sizes` array in lines 12 to 19. When the recursion finds an expression that does not have 4 children nodes, the recursion ends. There are more checks in the actual code to ensure that each of the 4 children nodes has the types we expect.

The array `sizes` resulting from the code in Listing 6.3 is later used to generate a DSPARLIB `SenderReceiver`. This will be described in Section 6.3.6. The programmer must be careful on specifying code with this syntax. When using the array as an input, the user cannot deallocate it inside the stages. SPAR-distributed generates code that deallocates the array automatically when the item finishes its processing. However, it is possible to deallocate the array and then allocate a new one if necessary. If the programmer wants to deallocate the array manually, he must set the size variable to 0 as well. This way, SPAR-

distributed will not attempt to deallocate the array. However, this is not necessary. It is safer let SPAR-distributed to manage the memory allocation.

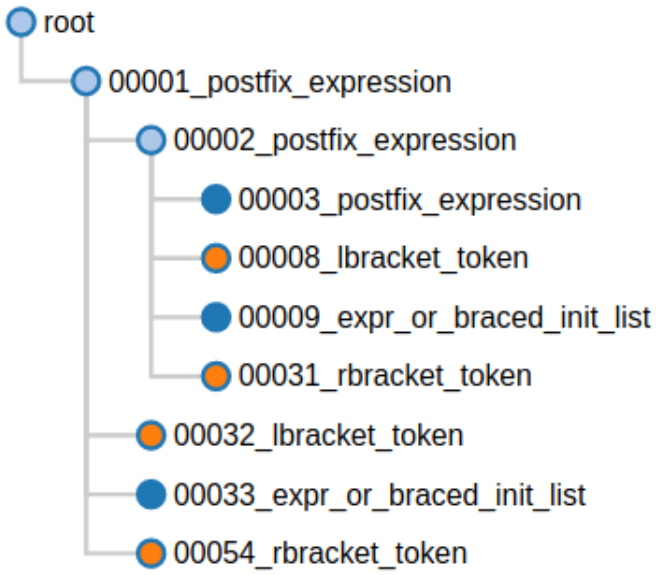

Figure 6.2 – 2D Array node tree for `array[dim1][dim2]`.

```
1  void parse_array(cincle::node *expression, std::vector<Declsize_info
       >& sizes) {
2      if (expression->childs_n != 4) return;
3
4      auto postfix = expression->node_down[0];
5      auto lbracket = expression->node_down[1];
6      auto between_braces = expression->node_down[2];
7      auto rbracket = expression->node_down[3];
8
9      //recurse
10     parse_array(postfix, sizes);
11
12     auto integer_literal = return_node_onlyleft_down(between_braces,
           NODE_TYPE_integer_literal);
13     if (integer_literal != NULL)
14         sizes.push_back(integer_literal);
15     else {
16         auto identifier = return_node_onlyleft_down(between_braces,
               NODE_TYPE_identifier);
17         if (identifier != NULL)
18             sizes.push_back(identifier);
19     }
20 }
```

Listing 6.3 – Parsing arrays.

6.2 Transformation rules

We do not changed the transformation rules already designed for SPAR-multicore. SPAR-distributed generates the same patterns as SPAR-multicore. However, FastFlow has a flexible API in which Emitters and Collectors of a Farm pattern can be removed. This is useful when there are many Farms (stages with `spar::Replicate`) connected in sequence on a Pipeline. At the moment being, DSPARLIB does not allow removing Emitters and Collectors. Therefore, we addressed it during the code generation. Section 6.3 explains how we dealt with this limitation. Since it is related to the data serizalization, this concerns more at the level of DSPARLIB and it will not require changes on the pattern-based transformation rules.

Recalling Rule 1 in Equation 6.1, which was introduced in Section 3.2. This is an annotation schema that has a `ToStream` region with a sequential code block, represented by $\square_0$, and one replicated stage, represented by $[[S_0, R_n]]$. This rule generates a Farm without a Collector, which is supported by FastFlow and not by DSPARLIB due to the API differences.

$$[[T_0]]\{\square_0, [[S_0, R_n]]\{\square_1\}\} \Rightarrow farm(E(\square_0), W(\square_1)) \tag{6.1}$$

If the stage S_0 has an output annotation, represented by O_i, SPAR-multicore will auto-generate a Collector stage, which can be seen in Rule 2 in Equation 6.2 represented by $C(\Psi)$. The auto-generated Collector $C(\Psi)$ has a specific purpose: make the Farm result available outside the `ToStream` region.

$$[[T_0]]\{\square_0, [[S_0, O_i, R_n]]\{\square_1\}\} \Rightarrow farm(E(\square_0), W(\square_1), C(\Psi)) \tag{6.2}$$

For SPAR-distributed, we also auto-generate another type of stage: a "placeholder" stage. A placeholder stage is a stage that receives a value and immediately emits the same value to the next stage. A placeholder Collector is a Collector that does nothing. If a new rule was necessary, it could be similar to the Equation 6.3. In this equation, Θ is an auto-generated placeholder Collector, different from an auto-generated Collector stage Ψ. In the future, DSPARLIB-distributed could implement support for $farm(E, W)$, where these rules will no longer needed. Therefore, the auto-generated placeholder stage is an implementation detail.

$$farm(E(\square_0), W(\square_1)) \Rightarrow farm(E(\square_0), W(\square_1), C(\Theta)) \tag{6.3}$$

We demonstrate how the auto-generated placeholders are used in more complex scenarios in Equation 6.4, where the annotation schema is transformed into a Pipeline with 3 stages and a Farm in the end. Because the last stage S_4 is replicated with the R_n annotation, S_4 must become a worker stage in a Farm pattern, due to Definition D2 in Table 3.1. SPAR uses the previous non-replicated stage S_3 as an Emitter. The last stage S_4 also the O_i annotation, therefore it needs a further auto-generated Collector $C(\Psi)$ to collect the results.

$$\begin{gathered}[[T_0]]\{\square_0, [[S_1, O_i]]\{\square_1\}, [[S_2, O_i]]\{\square_2\}, [[S_3, O_i]]\{\square_3\}, [[S_4, O_i, R_n]]\{\square_4\}\} \\ \Downarrow \\ pipe(\square_0, \square_1, \square_2, farm(E(\square_3), W(\square_4), C(\Psi)))\end{gathered} \tag{6.4}$$

Equation 6.5 presents a Pipeline with multiple Farms. In this case, all Farms in the middle of the Pipeline need Emitter and Collector placeholders Θ while the last Farm needs an auto-generated Collector Ψ. The last replicated stage S_6 uses S_5 as Emitter since it is not replicated, and a generated $C(\Psi)$ to collect the results.

$$\begin{gathered}[[T_0]]\{\square_0, [[S_1, O_i]]\{\square_1\}, [[S_2, O_i, R_n]]\{\square_2\}, [[S_3, O_i, R_n]]\{\square_3\}, \\ [[S_4, O_i, R_n]]\{\square_4\}\}, [[S_5, O_i]]\{\square_5\}\}, [[S_6, O_i, R_n]]\{\square_6\}\} \\ \Downarrow \\ pipe(\square_0, farm(E(\square_1), W(\square_2), C(\Theta)), farm(E(\Theta), W(\square_3), C(\Theta)), \\ farm(E(\Theta), W(\square_4), C(\Theta)), farm(E(\square_5), W(\square_6), C(\Psi)))\end{gathered} \tag{6.5}$$

6.3 Code generation

While the transformation rules did not change, a significant part of the code has been changed. This implementation is necessary to address the limitations in DSPARLIB more easily. Additionally, this process helped studying the code and understanding the compiler code generation internals. Differently from FastFlow API, DSPARLIB only supports creating wrappers using the `Wrapper` API as explained in Section 5.5. Support for using lambdas or functions directly is not implemented. SPAR-multicore generates stages using function wrappers and `ff_node_t` wrappers. This complicates code generation because, in some cases, a `ff_node_F` becomes necessary to wrap the function while Farm Workers can receive a plain C++ function directly. In summary, the SPAR-multicore code generator uses many FastFlow features that DSPARLIB does not support. The new SPAR-distributed code generation is performed in the following steps:

1. Process the transformation rules and determine pattern instances;
2. Generate C++ code to instantiate the wrappers;
3. Generate C++ code to instantiate the Farms and Pipeline stages;
4. Generate C++ code to create the Pipeline (pattern invocation);
5. Generate C++ code to synchronize process state.

The SPAR-multicore joins steps 1, 2, and 3. Initially, this was difficult to understand since rule processing is complex. We separated the rule processing and code generation parts to make it easier to understand and change. We will show it in the next sections. The SPAR-distributed code generation steps are represented in Figure 6.3. In yellow we highlight the steps executed by the SPAR compiler for all architectures and green are the steps of the SPAR-distributed version. We will explore each of the code generation steps deeper in the next sections.

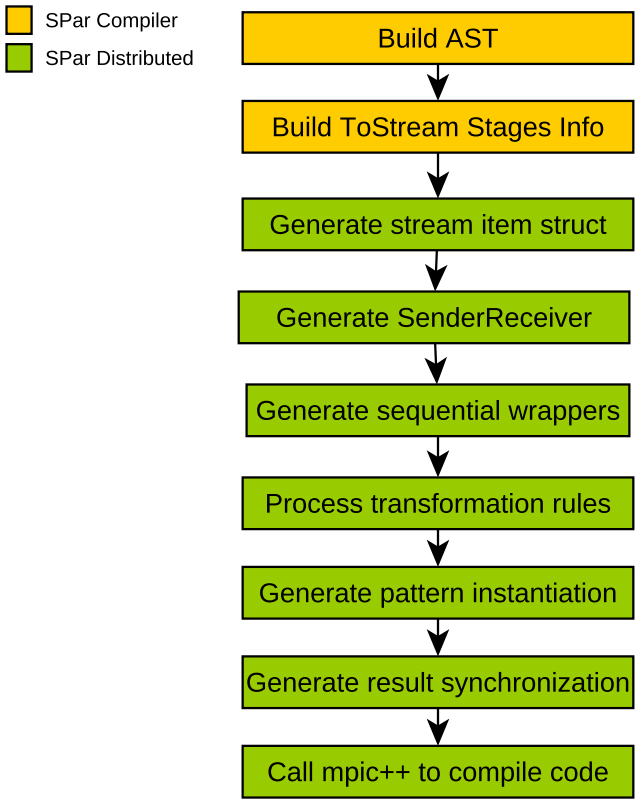

Figure 6.3 – SPAR-distributed compiler steps and dependencies.

6.3.1 Processing the transformation rules and determining pattern instances

Processing the transformation rules is the most important step. All the next steps enumerated in Section 6.3 depend on it. First, we address problems in the SPAR-multicore where code reuse was difficult.

The code generator in SPAR-multicore has 2 concerns: Determining the instantiation of the patterns (we will call it “rule processing”) and generating the C++ code. However, these concerns were not obvious while reading the internals of the compiler. For the SPAR-distributed version, we decided to refactor the code. This refactoring process enabled us to study the code and make it easier for us to understand it.

The SPAR-multicore transformation code loops through the stages (`spar::Stage` attributes) and generates C++ FastFlow code directly. Listing 6.4 presents an example of the current SPAR-multicore pattern code generation. It determines that a Farm must be instantiated (due to the `Replicate` annotation) in line 1 and immediately instantiates in line 4 it by concatenating the code to the `runtime_init` variable.

```
1 if (spar_region.Stage_list[i].replicate.nargs == 1) {
2   //initiating a farm
3   if (ordered_flag == true) {
4     runtime_init += "ff_OFarm<" + SPAR_STREAM_NAME + std::to_string(
          spar_region.id) + "> " + STAGE_REGION_NAME + std::to_string(
          spar_region.id);
5   }
6   ...
```

Listing 6.4 – Example of the SPAR-multicore’s pattern instantiation code.

SPAR-distributed generates code using a different approaches. First, SPAR-distributed performs rule processing by looping through all the stages, and builds a stage graph. The graph tells us how stages are connected to each other. SPAR-distributed only generates code after building the stage graph. This graph was helpful during development because it provided a simpler view of the connections between stages. The alternative is to generate code directly, but it was harder to understand as a beginner to the SPAR internals.

Each node in the stage graph is represented by a `struct` called `DSparStage`, shown in Listing 6.5. This `struct` stores the ID of the stage attribute (`spar:Stage`), type, replication expression (for instance: `spar::Replicate(5)`), wrappers (name of the `dspar::Wrapper` generated structs) and the relationship of this stage to other stages. If the stage is an Emitter or Collector of a Farm pattern, we store the IDs in the fields `emitter_of` and `collector_of`. If the stage has a `spar::Replicate` attribute, it becomes a Farm Worker. If the stage is a Farm Worker, the `emitter` and `collector` fields store IDs of the Emitter and Collector stages, of which this Worker stage is part of. Furthermore, if the stage is a Farm Worker, we

also set the `type` field as `TYPE_FARM`. Therefore, the connections between Farm components are stored bi-directionally. We also store flags to control whether the Emitter is the `ToStream` stage (`emitter_is_tostream`) or whether the Collector of this stage will be the last stage of the whole Pipeline (`collector_is_last_stage`).

```
1  #define TYPE_STAGE 1
2  #define TYPE_FARM 2
3
4  struct DSparStage {
5    int id;
6    int type; //1 - stage, 2 - farm
7    std::string replicas;
8    std::string wrapper_name;
9    std::string emitter_wrapper;
10   std::string collector_wrapper;
11   int emitter_of;
12   int collector_of;
13   int emitter;
14   int collector;
15   bool emitter_is_tostream;
16   bool collector_is_last_stage;
17 };
```

Listing 6.5 – DSparStage struct.

To process the rules, fist we initialize the stages. The initialization is done by the `get_stages` function shown in Listing 6.6. This initialization only sets the IDs of the stages in line 6. The other values are filled with default values that will be changed during rule processing. Each stage found in the annotated C++ code becomes an item in the vector `dspar_stages`. The `ToStream_Info` parameter in line 1 is generated in a previous stage of the SPAR compiler before actually invoking the architecture-specific transformations (SPAR-distributed or SPAR-multicore).

```
1  std::vector<DSparStage> get_stages(ToStream_Info spar_region) {
2    std::vector<DSparStage> dspar_stages;
3    for (unsigned long int i = 0; i < spar_region.nStages; ++i) {
4      DSparStage s;
5      s.id = i;
6      s.type = -1;
7      s.replicas = "";
8      s.wrapper_name = "NOT_DETERMINED";
9      s.emitter_of = -1;
10     s.collector_of = -1;
11     s.emitter = -1;
12     s.collector = -1;
13     dspar_stages.push_back(s);
14   }
15 }
```

Listing 6.6 – DSParStage vector instantiation.

The rest of the code in the `get_stages` function is responsible for processing the rules. The code is too big to show in this document, therefore, we only present a small fraction of it in Listing 6.7. It demonstrates how the `DSparStage` values are handled. We iterate over the stages of a `spar::ToStream` region in line 4. Then we check in line 5 whether the stage is the Worker of a Farm. In line 10, we set the `emitter_is_tostream` field as `true` because it is the first annotated block of the stream parallelism region and it is replicated. This field is named `emitter_is_tostream` because the code between the `spar::ToStream` annotation and the first `spar::Stage` annotation becomes the Emitter of a Farm pattern when the first stage is replicated.

```
1  for (int i = 0; i < spar_region.nStages; ++i) {
2  auto stage = spar_region.Stage_list[i];
3  auto &s = dspar_stages[i];
4  if (i == 0 && spar_region.nStages > 1) {
5    if (stage.replicate.nargs == 1) { //is a farm(E,W, C)
6     s.type = TYPE_FARM;
7     s.replicas = stage.replicate.decl_list[0];
8     s.wrapper_name = STAGE_REGION_NAME + std::to_string(spar_region.id
          ) + std::to_string(stage.id);
9     s.emitter_wrapper = TOSTREAM_REGION_NAME + std::to_string(
          spar_region.id);
10    s.emitter_is_tostream = true;
11   }
12   else { //is a stage
13    s.type = TYPE_STAGE;
14    s.wrapper_name = STAGE_REGION_NAME + std::to_string(spar_region.id
          ) + std::to_string(stage.id);
15   }
16   ...
17 }
18 ...
19 }
```

Listing 6.7 – Part of the rule processing code.

6.3.2 Generating C++ code to instantiate the wrappers

Before we instantiate the patterns, we need to instantiate the wrappers. Although we have not explained how the sequential code is moved to the wrappers yet, this process has already happened at this stage. The process of moving sequential code to wrappers will be explained later. Each wrapper has a name, and in this step, we just instantiate them. Listing 6.8 presents the code needed for implementing the wrapper instantiation. We iterate over the stages and generate the code.

```
1 for (auto stage: dspar_stages) {
2   auto name = stage.wrapper_name;
```

```
3   runtime_init += name + " " +name + "_wrapper;\n";
4 }
```

Listing 6.8 – Wrapper code instantiation.

6.3.3 Generating C++ code to instantiate the Farms and Pipeline Stages

After processing the rules and instantiating the wrappers, we can generate the Farm pattern and the Pipeline stages. We present a fraction of this code in Listing 6.9. In this example, we iterate over the stages and check whether the element is a Farm or a stage. In line 12, we check if the annotated stage block is not being used as a Collector or an Emitter of another Farm. If this is the case, we do not generate a Pipeline stage, because it will be part of a Farm pattern. For the Farm case in line 4, a Farm is always generated. We also check whether the stage has the `ToStream` block as an Emitter in line 8. In the actual code (not shown here), we also check whether we need to generate the placeholder stages. We also set flags such as whether the Farm is ordered, the scheduling algorithm (round-robin or on-demand), and the number of Worker replicas.

This strategy also removes code repetition. In the SPAR-multicore, the code that configures the Farm instance (ordering, scheduling, and replicas) is repeated in many parts. In SPAR-distributed this code is in a single place. In the future, these changes can be incorporated in SPAR-multicore too.

```
1  for (size_t i = 0; i < dspar_stages.size(); i++) {
2    auto stage = dspar_stages[i];
3
4    if (stage.type == TYPE_FARM) {
5     std::string farm_instance = "auto " + stage.wrapper_name + "_call
         = dspar::MakeFarm(\n\t";
6
7     //add emitter
8     if (stage.emitter_is_tostream) {
9      farm_instance += stage.emitter_wrapper + "_wrapper, " +
          serializer + ", \n\t";
10    } else { ...
11   ...
12   } else if (stage.type == TYPE_STAGE && stage.collector_of == -1 &&
        stage.emitter_of == -1) {
13     ...
14   }
15   ...
16 }
```

Listing 6.9 – C++ code generation for pattern instances.

6.3.4 Generating C++ code to create the Pipeline (pattern invocation)

After instantiating the Farms and Stages, we create the Pipeline object. The code in the SPAR-multicore detects whether there is only a single Farm (where there is no need for a Pipeline) or if a Pipeline is necessary. Moreover, SPAR-multicore uses a FastFlow feature called “freezing” to reuse threads when possible. In DSPARLIB, there is no “freezing” feature, and there is no overhead of using a Pipeline object even when having only a single Farm pattern. Therefore, we simply add the stages to the Pipeline. Listing 6.10 shows the code. An important corner case is when the first stage is not the Emitter of a Farm stage, for instance, a *pipe*(*stage*, *farm*(*E*, *W*, *C*)). In this case, the Emitter is the `ToStream` block as can be seen in line 6.

```
1  std::string pipeName = "_pipeline" + std::to_string(spar_region.id);
2  runtime_call += "\n\n";
3  runtime_call += "\tPipeline _pipeline" + std::to_string(spar_region.
       id) + ";\n";
4
5  if (dspar_stages[0].type == TYPE_STAGE && dspar_stages[0].emitter_of
       != -1) {
6      runtime_call += "\t" + pipeName + ".Add(&" + TOSTREAM_REGION_NAME
           + std::to_string(spar_region.id) + "_call);\n";
7  }
8
9  for (size_t i = 0; i < dspar_stages.size(); i++) {
10     auto stage = dspar_stages[i];
11     bool isFarm = stage.type == TYPE_FARM;
12     bool isStage = stage.type == TYPE_STAGE && stage.collector_of ==
           -1 && stage.emitter_of == -1;
13     if (isFarm || isStage) {
14         runtime_call += "\t" + pipeName + ".Add(&" + stage.
               wrapper_name + "_call);\n";
15     }
16 }
```

Listing 6.10 – C++ code generation for the Pipeline pattern.

The code generation process until this point is mostly based on the SPAR-multicore. However, there are specific problems in the SPAR-distributed that will be addressed in the next sections.

6.3.5 Result synchronization

When running an MPI program, there are parts of code that will execute in all processes and other parts of code that only runs in a specific processes. This is know as SPMD (Single Program Multiple Data). In our case, all code before the `[[spar::ToStream]]`

will run in all processes. This means that, when the code reaches the stream parallelism region, the state of the program in all processes should be the same. However, after the stream finishes, the result is only available in the Collector process. This is not the case for the SPAR-multicore since it is a single process with multple threads.

After the stream processing region, the state of the application has to be the same in all processes. The programmer can use the `spar::Output` in the `spar::ToStream` block to inform which variables are available after the streaming block. Listing 6.11 presents the use of this annotation in line 2, where the modifications to the `out` variable is available after the stream finishes. Consequently, we need to synchronize the state of the variables in the `spar::Output` of all processes. In SPAR-distributed, the process state synchronization can be done with a straightforward messages exchanging in MPI. The Collector sends the values of the variables to all other processes in the stream. Other processes wait for this synchronization to continue the execution. All processes have the same result after when the synchronization finishes.

```
1  int out = 0;
2  [[spar::ToStream, spar::Input(out), spar::Output(out)]]
3  while (1) {
4      int i = 0;
5      int j = 0;
6      [[spar::Stage, spar::Input(j), spar::Output(i)]] {
7          i = compute(j);
8          j++;
9      }
10     ..
11     [[spar::Stage, spar::Input(i, out)]] {
12         out++;
13     }
14 }
15 std::cout << out << std::endl;
```

Listing 6.11 – ToStream with Output annotation.

6.3.6 Generating data serialization operators and handling memory allocations

In SPAR-multicore and SPAR-distributed, all the variables in the `spar::Input` are aggregated into a single data structure to become the stream item. Consider the example in Listing 6.11. All the variables in `spar::Input` annotations (`out`, `i`, and `j`) are put into a single structure as exemplified in Listing 6.12. A single `SenderReceiver` is generated for the `struct`. In Listing 6.12, we also added the fields `arr` and `arrSize` for completeness. Sending the `struct` fields is quite trivial while handling arrays is more complicated. This section describes how we handle C-style dynamically-allocated arrays.

```
1 struct progname_spar0{
2   int out;
3   int i;
4   int k;
5   char* arr;
6   int arrSize;
7   bool __spar_should_deallocate;
8 }
```

Listing 6.12 – Example of the generated `struct`.

To serialize `progname_spar0` in Listing 6.12, we generate the `SenderReceiver` in Listing 6.13. Notice that we check for a `__spar_should_deallocate` flag in line 7, which is a flag that enables or disables automatic memory deallocation of arrays. The flag is set `true` when the data is received in line 19. After processing an item, the array is deallocated automatically *only if* the flag is set to `true` in line 8.

However, the Emitter does *not* set the deallocation flag as `true`. SPAR-distributed *cannot* automatically deallocate memory of arrays created in the Emitter. It might have been allocated with the `malloc` function (if the program uses C-style allocation) or with a custom memory allocator. Deleting data allocated with `malloc` using the `delete` operator may cause crashes in the program, therefore, we let the user handle the memory deallocation after the `ToStream` region.

```
1 class serializer_for_progname_spar0: public SenderReceiver<
    progname_spar0>{
2 public:
3 void Send(MPISender& sender,MessageHeader& msg, progname_spar0& data)
    override {
4   sender.SendTo(msg,data);
5   if(data.arrSize > 0) {
6     sender.SendTo(msg,data.arr,data.arrSize);
7     if(data.__spar_should_deallocate) {
8       delete[] data.arr;
9     }
10  }
11 };
12 progname_spar0 Receive(MPIReceiver& receiver,MessageHeader& msg)
    override {
13  progname_spar0 data;
14  receiver.Receive(msg, &data);
15  if(data.arrSize > 0) {
16    data.arr = new unsigned char [data.arrSize];
17    receiver.Receive(msg,data.arr,data.arrSize);
18  }
19  data.__spar_should_deallocate = true;
20 }
21 }
```

Listing 6.13 – Example of the generated serializer.

When an array is received by another process, we allocate the memory using the `new` operator, as can be seen on line 16 of Listing 6.13. After receiving the data, we pass the data to the `spar::Stage`. The user can modify this array in-place or reallocate another array, or even delete it using the appropriate `delete` operator. If the user deletes it for whatever reason (it is not needed), then the user must set the `arrSize` to `0`, which avoids a double-free error. The stage that received the data will also send the modified data to another process. After the data is sent, we deallocate the array in line 7.

We always set the `__spar_should_deallocate` on the receiving side. This is because SPAR-distributed allocated the arrays and we consider that SPAR-distributed owns the allocation intead of the programmer's code. In the case of multi-dimensional arrays, SPAR-distributed generates code that handles memory allocation and deallocation for all dimensions. The `__spar_should_deallocate` flag is only used by SPAR-distributed to control memory deallocations automatically. The application programmer cannot set this flag.

The first version of the serialization generator used to sent each field of the `struct` separately, resulting in 5 `MPI_Send` calls for the `struct` in Listing 6.12. For programs that need to exchange millions of stream items, the overhead becomes significant. We will present a discussion about the performance of serializers in Section 6.5.

6.3.7 Data serialization for C++ STL types

The SPAR language is based on C++. Therefore, it is important to support sending and receiving STL types such as `std::vector`. However, there are many STL types and we have not supported all of them in this work. For now, we support only `std::vector` and `std::array`. Support for more types can be implemented in DSPARLIB as needed. In this section, we briefly describe how `std::vector` is supported.

For `std::vector`, the user does not need to inform the size of the vector with the new `spar::Input(arr[size])` syntax. This is because `std::vector` already has a size property, so DSPARLIB can use it. The user can also send and receive nested `std::vector` such as `std::vector<std::vector<std::vector<T>>>`. An example is presented in Listing 6.14. In line 3 a new 2D vector is created, and it can be passed to the `spar:Input` without using the new array syntax.

```
1 [[spar::ToStream]] while (1) {
2     std::vector<std::vector<int>> vec = get2DVector();
3     [[spar::Stage, spar::Input(vec), spar::Replicate(10)]] {
4         compute(vec)
5     }
6 }
```

Listing 6.14 – Sending C++ STL types in SPAR.

To support serialization of STL types in DSPARLIB, we add new methods to `MPISender` and `MPIReceiver`. Listing 6.15 shows how we implemented the sender. We omit the receiver implementation for brevity.

```
1  template <typename T>
2  void SendTo(const MessageHeader &header, std::vector<T> &vector) {
3      SendTo(header, vector.size());
4      if (vector.size() > 0) {
5          SendTo(header, vector.data(), vector.size());
6      }
7  }
8  template <typename T>
9  void SendTo(const MessageHeader &header, std::vector<std::vector<T>>
       &vectors) {
10     std::vector<size_t> sizes;
11     for (auto &vec : vectors) {
12         sizes.push_back(vec.size());
13     }
14     SendTo(header, sizes);
15     for (auto &vec : vectors) {
16         if (vec.size() > 0) {
17             SendTo(header, vec.data(), vec.size());
18         }
19     }
20 }
```

Listing 6.15 – Sending C++ `std::vector`.

Sending `std::vector<T>` is straightforward. First, we send the size, and if the array is not empty, we send the data in lines 3 to 5 of Listing 6.15. The receiver will also first receive the size, and if the size is larger than 0, it receives the array. However, sending nested `std::vector<std::vector<T>>` is more difficult. The naïve method just sends each one of the inner vectors individually, which causes a problem. The number of individual MPI messages is almost twice the number of arrays. For instance, if the user has a 1000x30 2D nested vector structure, DSPARLIB would send 1 message for the outer vector size, 1000 messages for each inner array size, and more 1000 messages to send each array. Instead, DSPARLIB sends 2 messages for the array sizes (line 14), and 1000 messages for each array (line 15). We reduced the number of messages from 2001 to 1002.

For 3D-nested vectors, we just call the 2D-nested vector `SendTo` function repeatedly without further optimizations. Since 3D-nested vectors may be less common, a more optimized implementation can be developed later. For `std::array`, sending and receiving is similar. The size of the array is present in the template argument, therefore, it is available at compile time. The function in Listing 6.16 shows how to capture the size template argument.

```
1 template <typename T, size_t I>
2 void SendTo(const MessageHeader &header, std::array<T, I> &array){
3   SendTo(header, array.data(), I);
4 }
```

Listing 6.16 – Sending C++ `std::array`.

This strategy allows to evolve serialization in DSParLib and SPar-distributed separately. Syntactical extensions can be developed for SPar-distributed without requiring changes to DSParLib. Meanwhile, serialization can be improved by updating DSParLib without changing the compiler. Another strategy to support all STL types is to use libraries like Cereal [GV17]. However, this was not implemented in DSParLib.

6.3.8 Generating sequential wrapper classes

We generate a DSParLib `Wrapper` for each annotated stage. Depending on the position of the stage, we generate a wrapper with different input and output types. We perform the following steps:

- The `ToStream` block is moved to an `Wrapper<Nothing, TOut>` stage.
- If the last stage is annotated with `Output` and has *Replicate*, then we move the sequential code to a `Wrapper<TIn, TOut>` and also automatically generate a Collector to get the results. This is because the stage is replicated and by definition *D*2 in Table 3.1, it must be a Farm Worker. By definition *D*0, it requires an auto-generated block to gather its results.
- If the last stage is annotated with `Output` and is not replicated, then we move the sequential code to a `Wrapper<TIn, Nothing>`
- If the last stage has no `Output` annotation and is not marked to be used as a Collector by the rule processing code described in Section 6.3.1, we move the sequential code to a `Wrapper<TIn, TOut>`.
- If the last stage has no output and is marked to be used as a Collector, we move the sequential code to a `Wrapper<TIn, Nothing>`.
- For all other cases, we move the sequential code to a `Wrapper<TIn, TOut>`.

To generate the code for Emitter `Wrapper<Nothing, TOut>` wrappers, we iterate over the variables needed by the `spar::Input` annotation and add them as `struct` fields as shown in the Listing 6.17. The `Produce` method is the code before the first `spar::Stage`

annotation. The variables passed to the `spar::Input` annotation in line 1 become struct fields in line 7 to 9.

```
1  [[spar::ToStream, spar::Input(hInfile, blockSize, bytesLeft)]]
2  while (bytesLeft > 0) {
3      ...
4  }
5  //Becomes:
6  struct pbzip_spar_ToStream_spar0: Wrapper<Nothing,
       pbzip_spar_struct_spar0> {
7    int hInfile;
8    int blockSize;
9    OFF_T bytesLeft;
10   void Produce() override {
11     while (bytesLeft > 0) {
12       ...
13     }
14   }
15 };
```

Listing 6.17 – Input annotation and `struct` fields. This example was taken from the code generated on PBZIP2 application.

In Listing 6.18 we generate code to emit stream data items by instantiating the data `struct` in line 1 and filling the memory space of the `struct` with 0 in line 2. This is necessary to prevent memory errors when declaring array size variables which were not initialized. An uninitialized value will contain whatever is in memory, which is likely an invalid value. Filling the memory with 0 also causes arrays sizes to be 0. The `SenderReceiver` will not send them if the sizes are not set after the `memset`.

```
1 pbzip_spar_struct_spar0 stream_spar;
2 memset(& stream_spar,0,sizeof(pbzip_spar_struct_spar0));
3 stream_spar.__spar_should_deallocate = false;
4 stream_spar.CompressedData = CompressedData;
5 stream_spar.outSize = outSize;
6 stream_spar.FileData = FileData;
7 stream_spar.inSize = inSize;
8 Emit (stream_spar);
```

Listing 6.18 – Setting the stream item fields in the auto-generated stage. This example was taken from the code generated on PBZIP2 application.

The auto-generated Collector is generated in the same way as the SPAR-multicore. When the Collector receives data, it copies the values to its fields. These fields are similar to the Emitter fields shown in Listing 6.18 based on the `spar::Output` annotation instead of `spar::Input`. We generate a Collector `Wrapper<TIn, Nothing>` if it is the last stage of the Pipeline or a `Wrapper<TIn, TOut>` in all other cases. In Listing 6.19, an example of a generated `Wrapper` is presented.

```
1  struct pbzip_spar_Stage_spar00: Wrapper< pbzip_spar_struct_spar0,
       pbzip_spar_struct_spar0> {
2  void Process(pbzip_spar_struct_spar0 & pbzip_spar_Input_spar)
       override {
3   {
4     pbzip_spar_Input_spar.outSize = (int)((pbzip_spar_Input_spar.
         inSize*1.01)+600);
5     pbzip_spar_Input_spar.CompressedData = new char [
         pbzip_spar_Input_spar.outSize];
6     ...
7   }
8   Emit (pbzip_spar_Input_spar);
9  }
10 }
```

Listing 6.19 – Generated stage pattern example. Example taken from generated code for PBZIP2 application.

This concludes the code generation section. A complete description of every minor change made to the SPAR-distributed compiler is not feasible. Our description gives an overall idea of the changes and implementations.

6.4 Compiler flags

Currently, SPAR-multicore has compiler flags that allow the programmer to customize the runtime behavior of the program. Some of these flags are specific to features supported by FastFlow. We attempt to reuse the same flags in SPAR-distributed whenever possible. In this section, we briefly describe compiler flags supported by SPAR-distributed.

- `-spar_cluster`: Enables code generation for DSPARLIB instead of FastFlow.
- `-spar_process_allocation_dynamic`: Enables dynamic process allocation. This enables the program to run using the process described in Section 5.4. Instead of passing all processes in the command line with `-np NUMBER_OF_PROCESSES`, DSPARLIB will spawn the necessary number of processes automatically. If this flag is not used, the program will abort with an error message when the number of processes passed in the `-np` flag is not sufficient.
- `-spar_blocking`: This option has no effect when used together with `-spar_cluster`.
- `-spar_ordered`: All Collectors of generated Farm patterns process items in the order they were produced. Supported by both SPAR-multicore and SPAR-distributed versions.

- `-spar_ondemand`: Enables the on-demand scheduler. The on-demand scheduler for DSPARLIB is currently enabled only for communication between Emitters and Workers. Messages sent from a Collector to another Emitter still uses round-robin scheduling. This flag is supported by both SPAR-multicore and SPAR-distributed versions.

6.5 Performance evaluation

This section evaluates the performance of handwritten DSPARLIB parallel applications (tested in Section 5.11) and the parallel applications generated by SPAR-distributed. In these tests, we ran each application 30 times for each degree of parallelism. All tests ran on LAD (Laboratório de Alto Desempenho) at PUCRS, in the Cerrado cluster. Each node has 2 Intel(R)Xeon(R) CPU E5-2620 v3 @ 2.40GHz (12 cores, 24 threads) with 24GB of RAM memory. The MPI implementation is Open MPI 1.4.5. The operating system is Ubuntu 16.04 64 bits with kernel 4.4.0-146-generic. The machines are connected via Gigabit Ethernet and InfiniBand QDR 4x (32GBit/s). The application is compiled on GCC 9.3.0 using -O3 optimizations. OpenCV version is 2.4.13. We allocated the nodes 06 to 13 in the cluster.

Our main research goals is not to add significant performance difference when automatically generating the parallel code. SPAR-distributed has to present performance results close to DSPARLIB for many types of workloads: video processing (lane detection), data compression (PBZIP2), and others (Mandelbrot and Prime Numbers). In fact, for the specific set of applications tested in this section, we will demonstrate that the performance difference between DSPARLIB and SPAR-distributed is negligible. For measuring purposes, small changes to the code generated by SPAR-distributed were necessary to precisely take the execution time only for the stream parallelism region. This is hard to do and guarantee in the annotated code because we need to print the measurements just in the Collector instead of in all other processes. The Mandelbrot [Gri16], Lane Detection [GHDF17], PBZIP2 [GDTF17] and Prime Numbers [GDTF15, Gri16] were annotated with SPar in previous works. However, small code modifications were required, which are discussed later in Section 6.7. The DSPARLIB versions are the same ones described in Section 5.11.

In the following performance graphs, we compare the applications' performance with on-demand and round-robin scheduling. The vertical axis represents the throughput of the application and the horizontal axis represents the degree of parallelism, i.e. the number of Worker replicas for the Farm pattern. All tests use round-robin MPI node allocation (`--by-slot` flag on `mpirun`). Emitter and Collector processes have dedicated nodes and are isolated from each other. The reader must not confuse *MPI node allocation* with *work scheduling*. We used the round-robin node allocation strategy in all tests and on-demand/round-robin work scheduling when specified. All graphs show the following labels:

- `dsparlib-ond`: Code written with DSPARLIB and uses the on-demand scheduling.

- `dsparlib-rr`: Code written with DSPARLIB and uses round-robin scheduling.
- `spar-ond`: Code generated with SPAR-distributed and uses on-demand scheduling with `-spar_ondemand` compiler flag.
- `spar-rr`: Code generated with SPAR-distributed and uses round-robin scheduling.

Figure 6.4 plots the Mandelbrot performance results. Using on-demand scheduling, the performance is the same up to 52 Worker replicas. For 56 processes or more, there is more standard deviation as shown by the error-bars. The round-robin scheduling is no so good as on-demand, which was also observed on the DSPARLIB tests.

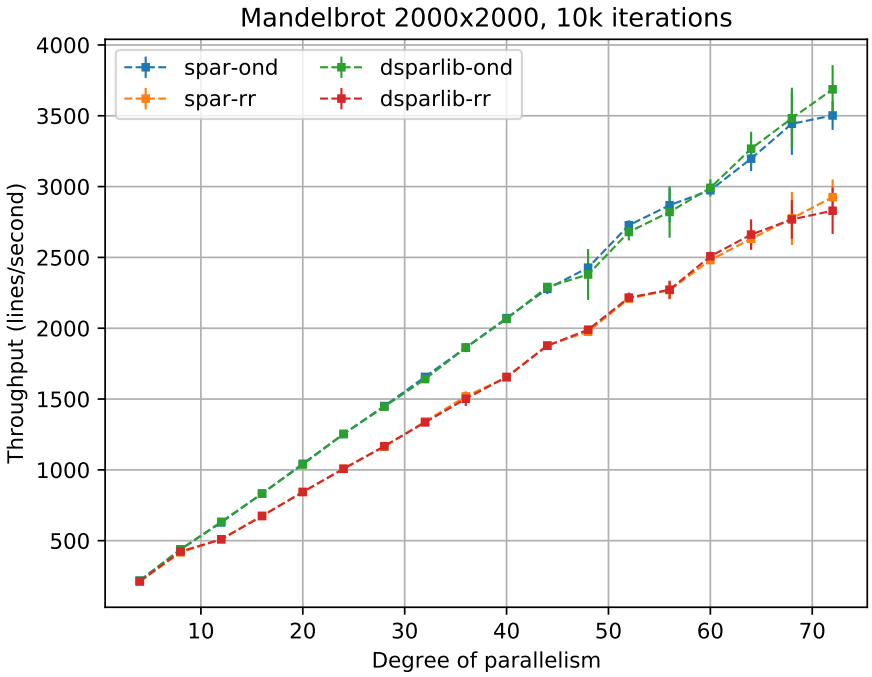

Figure 6.4 – Performance comparison between DSPARLIB and SPAR-distributed versions of the Mandelbrot Set application.

Figure 6.5 plots the PBZIP2 performance results. We compress a 512MB file at 275MB/s with 72 processes with the on-demand versions, and 250MB/s with the round-robin versions. SPAR did not cause a significant impact on the throughput.

Figure 6.6 plots the performance results for the Lane detection application. The behavior of both SPAR and DSPARLIB versions is nearly identical. Since the Lane Detection Farm Emitter only produces 1858 messages, one for each video frame, we were not concerned with serialization performance. Small microsecond latency variations resulting from more or less optimized serialization methods would not affect the result significantly for this application, as the majority of the time is spent in computation. For more than 30 processes, there are more standard deviation than observed in Section 5.11 because different cluster nodes were used when executing the tests.

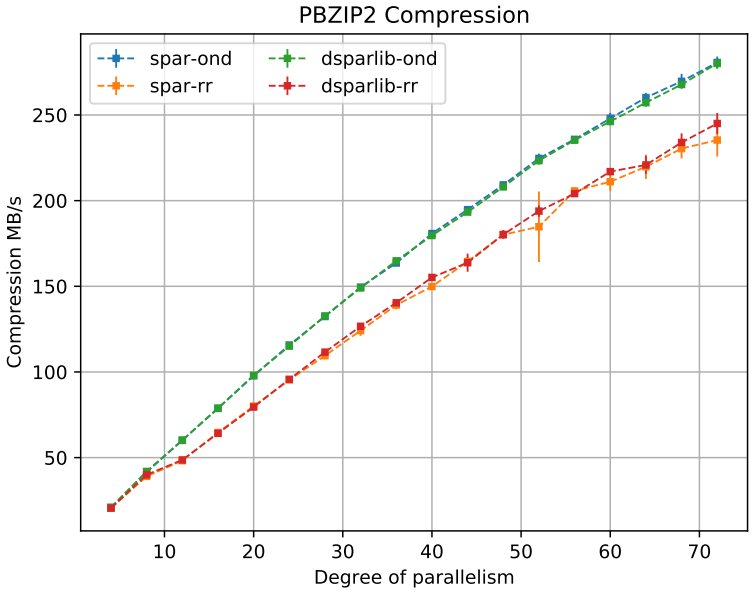

Figure 6.5 – PBZIP2 application performance comparison, reading the file from a NFS and saving to a local /tmp folder.

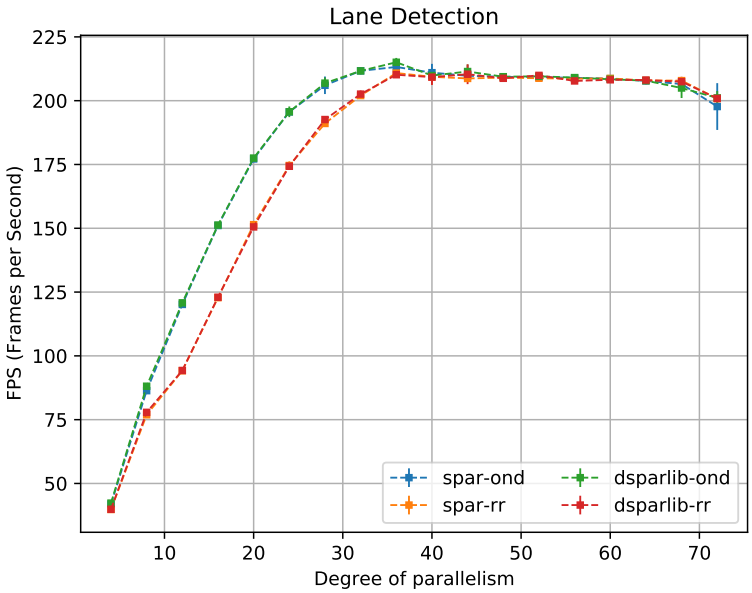

Figure 6.6 – Lane detection performance comparison between DSPARLIB and SPAR.

The prime numbers application is a case where millions of messages are sent between processes. Minor inefficiencies in serialization can add up and have a measurable impact on performance. Figure 6.7 shows this impact. Notice that the DSPARLIB version is faster. The graph shows only SPAR-distributed vs DSPARLIB on-demand and it rans the tests on the same nodes as the tests in Section 5.11.

Listing 6.20 shows a fraction of the annotated code for the Prime Numbers application. SPAR-distributed needs to serialize the fields `is_prime`, `i`, `n` and `t` passed in lines 3 and 7. In line 17, we show that SPAR-distributed generates 4 calls to `SendTo`. On the

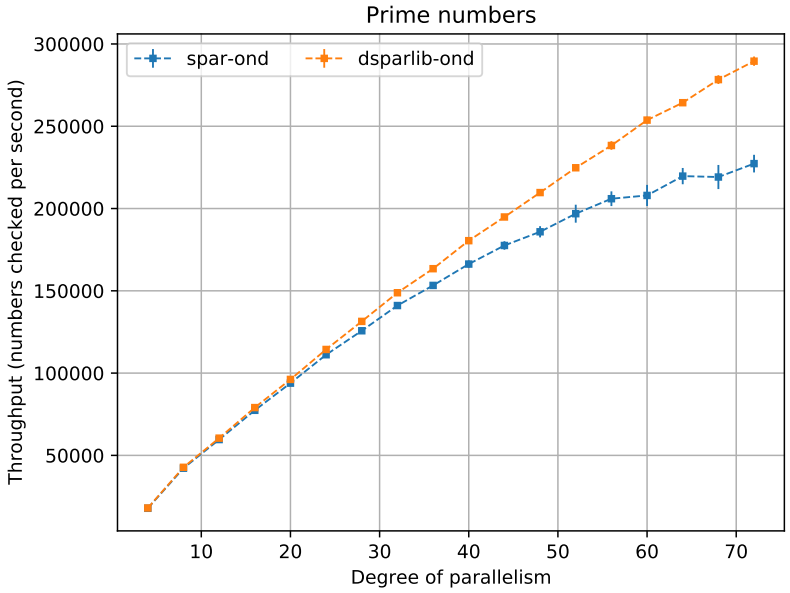

Figure 6.7 – Prime numbers performance comparison between DSPARLIB and unoptimized SPAR-distributed version.

other hand, the DSPARLIB version calls the `SendTo` only once. Our hypothesis is that each `MPI_Send` call incurs a small overhead.

```
1  //Annotated code:
2  int t = 0;
3  [[spar::ToStream, spar::Input(n, t), spar::Output(t)]]
4  for (int i = 2; i <= n; i++) {
5    bool is_prime;
6    [[spar::Stage, spar::Input(is_prime, i),
7        spar::Output(is_prime), spar::Replicate(workers)]] { ... }
8  }
9  //Generated Send method:
10 void Send(MPISender & sender,MessageHeader & msg,
       pn_spar_cluster_struct_spar0 & data) override {
11   sender.SendTo(msg,data.is_prime);
12   sender.SendTo(msg,data.i);
13   sender.SendTo(msg,data.n);
14   sender.SendTo(msg,data.t);
15 }
```

Listing 6.20 – Serialization for the Prime Numbers application before optimization.

We implemented an optimization in SPAR-distributed. Instead sending the `struct` field by field, we send the whole generated `struct` at once. This is possible because `structs` are contiguous blocks of memory. By sending the whole data block in one call, it reduces the number of messages `MPI_Send` calls, thus reducing the overhead. Listing 6.8 shows the optimized serialization. There is only one call to `SendTo` in line 3.

```
1 void Send(MPISender & sender,MessageHeader & msg,
     pn_spar_cluster_struct_spar0 & data) override {
2   sender.SendTo(msg,data);
3 }
```

Listing 6.21 – Serialization for the Prime Numbers application primes after optimization.

The result of this strategy is shown in Figure 6.8. The throughput of SPAR-distributed and DSPARLIB is the same for all degrees of parallelism. The round-robin versions perform poorly with similar throughput between SPAR-distribtued and DSPARLIB. Our strategy improved the throughput of the SPAR-distribtued version of the Prime Numbers application.

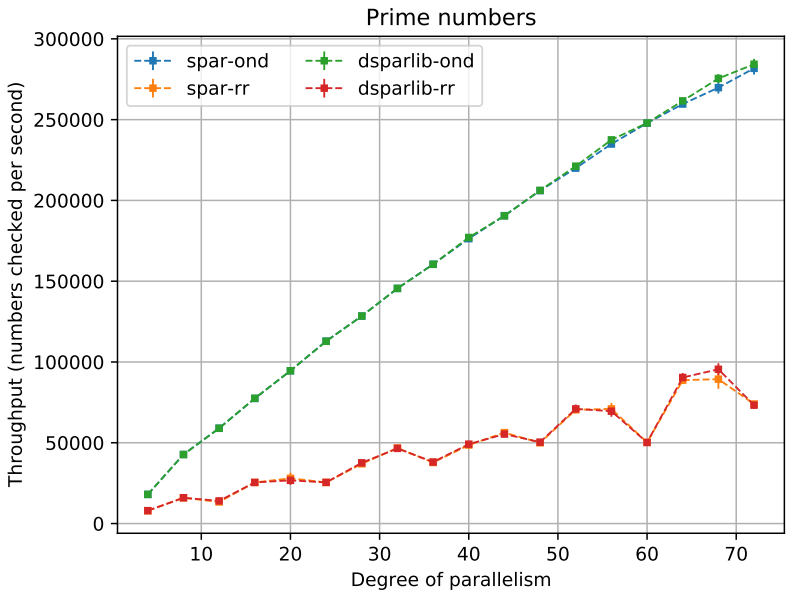

Figure 6.8 – Prime numbers application after optimizing DSPARLIB serialization.

These experiment has shown that SPAR-distributed can achieve close performance to the DSPARLIB handwritten codes. While there might be small overheads when compared to handwritten MPI and DSPARLIB, the applications tested in this section, can be considered with no significant performance differences. There are still a few optimizations in the code generation phase that can further improve the `SenderReceiver` performance, such as creating smaller data structures based on the `spar::Input` and `spar::Output` annotations.

6.6 Robust use case: SPAR-distributed for the Ferret application

In the previous sections, we explained how SPAR-distributed generates code and evaluated the performance compared to handwritten DSPARLIB versions. For these applications, we used existing annotated code with SPAR-multicore and used the new array syntax for the `spar::Input` and `spar::Output` annotations. In these applications, the array sizes

were readily available in the source code and they were only arrays with 1 dimension (for instance, `char*`). In this section, we present a case study for the Ferret application, where we parallelize the code using SPAR-distributed.

Ferret is part of the PARSEC benchmarks [LJW+06]. Ferret is like a search engine that finds images by similarity. Given a query image, it returns a set of images that have similar contents. Ferret has been annotated before with SPAR-multicore [GHDF18a]. In this section, we describe how the application can be parallelized with SPAR-distributed showing 3 versions with different annotation schemas.

There are 6 computation phases in the Ferret computation: Load the image queries, segment, extract, perform the query, rank, and collect the results. The Collector just prints the results in a file. For the first test, we create a single stage that encompasses the segment, extract, query, and rank stages. Figure 6.9 represents the topology graph built by SPAR, which is the normal form of the Farm pattern [AD99].

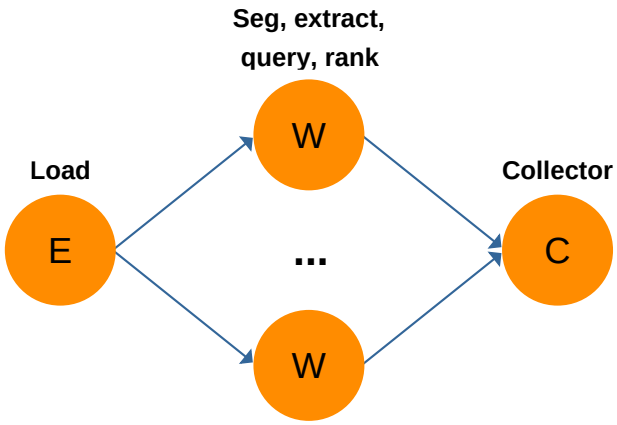

Figure 6.9 – Ferret - Normal form of the Farm pattern.

The Farm instantiation already provides challenging serialization problems on Ferret. We need to send the image RGB and HSV data from the Emitter to the Farm stage along with the file name. The user also needs to deserialize the data. In Listing 6.22 shows the `spar::Stage` annotation for the replicated stage in the Farm. The following explains how serialization is done for each variable:

- `name`: Since it is a C-style null-terminated string, we get the length using `strnlen` up to 2048 characters and we add 1 to the length for the null terminator character.
- `rgb` and `hsv`: We read Ferret's source code to find their sizes. The functions provided by Ferret's CASS framework (Content-Aware Search System) do not return the sizes for these arrays.

```
struct all_data *ret;
...
ret = file_helper(m_path);

size_t name_len = strnlen(ret->first.load.name, 2048);
char *name = ret->first.load.name;
name_len += 1;

int width = ret->first.load.width;
int height = ret->first.load.height;
int hsv_size = DEFAULT_SIZE * DEFAULT_SIZE * CHAN;
int rgb_size = DEFAULT_SIZE * DEFAULT_SIZE * CHAN;
unsigned char *hsv = ret->first.load.HSV;
unsigned char *rgb = ret->first.load.RGB;

cass_list_entry_t *cass_result = NULL;
cass_size_t result_size = 0;

[[spar::Stage,
    spar::Input(name[name_len], name_len,
                width, height,
                hsv[hsv_size], hsv_size,
                rgb[rgb_size], rgb_size),
    spar::Output(name[name_len], name_len,
                cass_result[result_size], result_size),
    spar::Replicate(NTHREAD)]]
{
    struct all_data ret2;
    ret2.first.load.name = name;
    ret2.first.load.width = width;
    ret2.first.load.height = height;
    ret2.first.load.HSV = hsv;
    ret2.first.load.RGB = rgb;

    ret2.second.seg.name = ret2.first.load.name;
    ret2.second.seg.width = ret2.first.load.width;
    ret2.second.seg.height = ret2.first.load.height;
    ret2.second.seg.HSV = ret2.first.load.HSV;

    //segment, extract, query, rank.

    cass_result = ret2.first.rank.result.u.list.data;
    result_size = ret2.first.rank.result.u.list.len;
}
```

Listing 6.22 – Annotated stage with SPAR-distributed for Ferret.

It is not straightforward to serialize Ferret's data structures. The array size information is only found inside the `.c` source code. Even if SPAR-distributed performed sophisticated analysis on the structs to find the sizes of the arrays, the `.c` files might not be available (for instance, having access to the header files and shared/static libraries instead of `.c` files).

We can construct a more complex topology by breaking the Ferret computation into a sequence of stages. Figure 6.9 depicts this topology. The green Collector and Emitter nodes (C and E) are placeholder nodes. Since the serialization code is complex, we will

show only the serialization for the query stage. The code is in Listing 6.23. The following explains how serialization is done for each value:

- `extract_dataset`: is the result of the extract stage, which type is `cass_dataset_t`. There are 2 array pointers `vec` and `vecset`. The `cass_dataset_t` has size information for both pointers.
- `result_lists_2d`: is a 2D array. However, the data structure in Ferret is a nested list. Each inner list might have different sizes. We need to allocate memory based on the list with the largest size. We explain the data structure in more detail later.

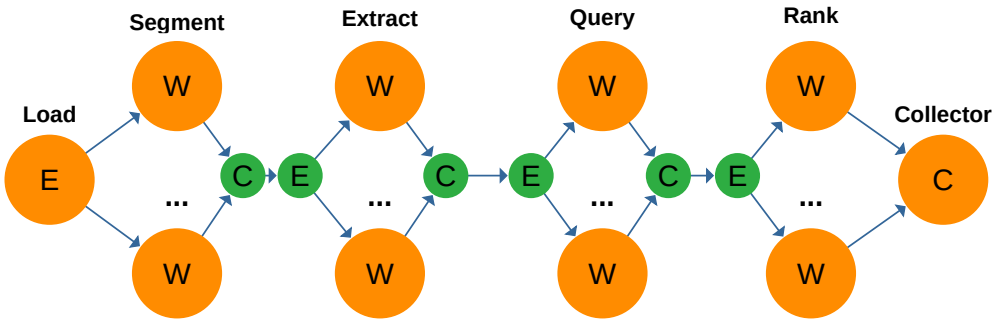

Figure 6.10 – Ferret - Pìpeline of Farms.

The problem of serializing Ferret data structures is due to non-contiguous data and nested data structures. The stage represented in Listing 6.23 performs a query in the Ferret database system. The result can be a list of results or a “list of lists”, which is like a 2-dimensional list.

```
1  [[spar::Stage,
2   spar::Input(
3    extract_dataset, extract_dataset_vec[extract_dataset_vec_size],
         extract_dataset_vec_size,extract_dataset_vecset[
         extract_dataset_vecset_size], extract_dataset_vecset_size),
4   spar::Output(
5    extract_dataset, extract_dataset_vec[extract_dataset_vec_size],
         extract_dataset_vec_size, extract_dataset_vecset[
         extract_dataset_vecset_size], extract_dataset_vecset_size,
         result_lists_2d_inc, result_lists_2d_len,
6    result_lists_2d[result_lists_2d_size],result_lists_2d_size,
7    result_lists_2d_raw_data[result_lists_2d_size][
         result_lists_2d_maxlist_size], result_lists_2d_maxlist_size
8   ),
9   spar::Replicate(NTHREAD)]] {
10   //deserialize data:
11   struct all_data ret2;
12   ret2.extract.ds = extract_dataset;
13   ret2.extract.ds.vec = extract_dataset_vec;
14   ret2.extract.ds.vecset = extract_dataset_vecset;
```

```
15
16    //perform query
17
18    //serialize the data:
19    auto lists = ret2.second.vec.result.u.lists;
20    result_lists_2d = lists.data;
21    result_lists_2d_inc = lists.inc;
22    result_lists_2d_len = lists.len;
23    result_lists_2d_size = lists.size;
24    result_lists_2d_maxlist_size = 0;
25    for (cass_size_t i = 0; i< result_lists_2d_size; i++) {
26      if (result_lists_2d[i].size > result_lists_2d_maxlist_size) {
27        result_lists_2d_maxlist_size = result_lists_2d[i].size;
28      }
29    }
30    result_lists_2d_raw_data = new cass_list_entry_t*[
         result_lists_2d_size];
31    for (cass_size_t i = 0; i< result_lists_2d_size; i++) {
32      result_lists_2d_raw_data[i] = new cass_list_entry_t[
           result_lists_2d_maxlist_size];
33      for (cass_size_t j = 0; j < result_lists_2d_maxlist_size; j++) {
34        result_lists_2d_raw_data[i][j] = result_lists_2d[i].data[j];
35      }
36    }
37  }
```

Listing 6.23 – Complex serialization in SPAR-distributed for Ferret.

This 2D list is not represented as a multidimensional C array. Instead, each level of these lists is represented by the struct shown in Listing 6.24. The `cass_result_t lists` field type is defined as `ARRAY_TYPE(cass_list_t)`. Listing 6.24 also shows the expansion of the `ARRAY_TYPE` macro.

The nested lists can have different sizes. In SPAR-distributed, we need to transform this structure into a simpler `cass_list_entry_t **` pointer, which is done in the end of Listing 6.23. Since the new array syntax does not support sending individual sizes for each of the lists, we need to find the largest list size, then allocate and send lists of that size. This approach can be inefficient if one of the lists is much larger than the rest. Fortunately, deserializing is much simpler, consisting of simply instantiating the result and looping through the `result_lists_2d_raw_data`, and then assigning the `struct` fields. The code is shown in Listing 6.25.

```
1  typedef struct {
2    uint32_t flags;
3    union {
4        bitmap_t bitmap;
5        cass_list_t list;
6        ARRAY_TYPE(bitmap_t) bitmaps;
7        ARRAY_TYPE(cass_list_t) lists;
8    } u;
9  } cass_result_t;
10
11 //This is the macro expansion of the cass_result_t lists field:
```

```
typedef struct {
  uint32_t flags;
  union {
        bitmap_t bitmap;
        cass_list_t list;
        ARRAY_TYPE(bitmap_t) bitmaps;
        struct {
            cass_size_t inc;
            cass_size_t size;
            cass_size_t len;
            struct {
                cass_size_t inc;
                cass_size_t size;
                cass_size_t len;
                cass_list_entry_t *data;
            } * data;
        } * lists;
  } u;
} cass_result_t;
```

Listing 6.24 – Ferret `cass_result_t struct`.

```
struct all_data ret2;
ret2.second.vec.result.u.list = result_list;
ret2.second.vec.result.u.list.data = result_list_data;
for (cass_size_t i = 0; i< result_lists_2d_size; i++) {
    result_lists_2d[i].data = result_lists_2d_raw_data[i];
}
ret2.second.vec.result.flags = CASS_RESULT_LISTS |
   CASS_RESULT_USERMEM;
ret2.second.vec.result.u.lists.data = result_lists_2d;
ret2.second.vec.result.u.lists.size = result_lists_2d_size;
ret2.second.vec.result.u.lists.inc = result_lists_2d_inc;
ret2.second.vec.result.u.lists.len = result_lists_2d_len;
ret2.second.vec.ds = &extract_dataset;
ret2.second.vec.ds->vec = extract_dataset_vec;
ret2.second.vec.ds->vecset = extract_dataset_vecset;
```

Listing 6.25 – Deserialization of Ferret query result.

In the first Farm example, it was not so difficult to find the memory allocation size required inside the source code of Ferret. However, for the query result, finding the sizes was significantly harder. We had to look deeper in the `.c` source code of the libraries to find the sizes. Due to the difficulty of serializing the query result, we can merge the query and rank stages. The topology shown in Figure 6.11 avoids many serialization issues, because all Ferret data structures that need to be done are simple pointers and sizes.

If the user must separate the query and rank, then another option to avoid most of the serialization complexity is to use `std::vector`. Listing 6.26 shows the same serialization performed in Listing 6.23, but simpler.

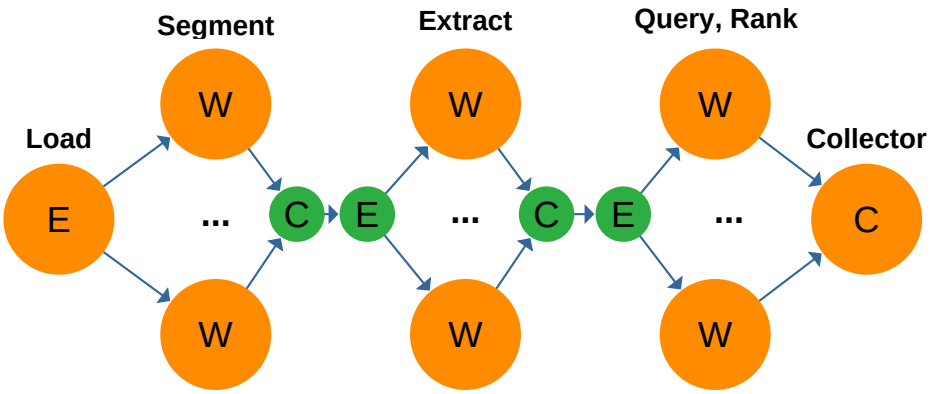

Figure 6.11 – Ferret - Pipeline of Farms, merging the query and rank operators.

We evaluate the performance of each topology. We do not have pure MPI or DSParLib versions of Ferret. We only compare the different topologies themselves generated by SPar-distributed. The scheduling is on-demand and the MPI process allocation strategy is round-robin (`roundrobin-ec-isolate`). Figure 6.12 plots the performance results. The `farm` line refers to the Farm topology represented in Figure 6.9. The `pipe-farm(4)` line refers to the topology where all stages have separated Farms as shown in Figure 6.10. The `pipe-farm(3)` line refers to the topology where the query and rank stages are joined as shown in Figure 6.11. Finally, `pipe-farm-stl(4)` is the same topology as Figure 6.10, which uses `std::vector` to serialize data.

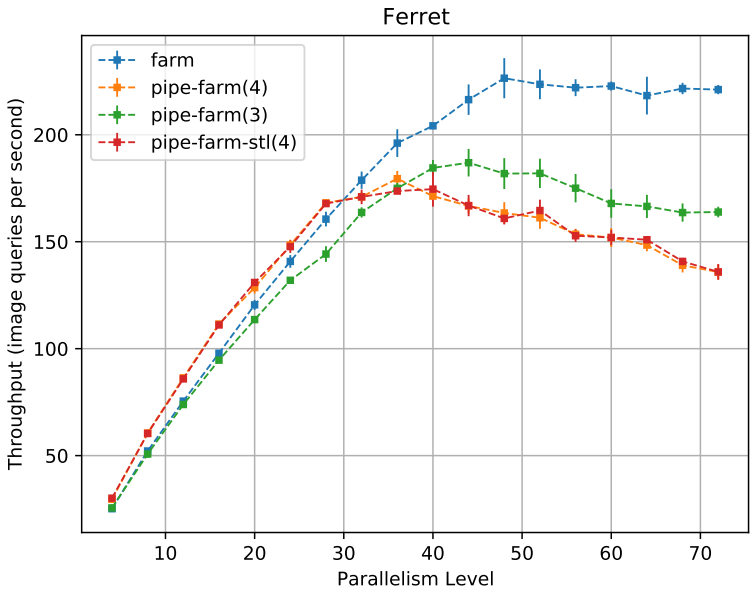

Figure 6.12 – Ferret Benchmark, using the `input-native` dataset.

The normal form of the Farm topology achieved the best result, scaling well up to 48 processes. For all implementations tested in Section 5.11, we avoid using hyper-threading

by specifying how many processes can be allocated in each node. However, the parallelism level is set per replicated stage (Farm). There are 4 Farm patterns in `pipe-farm(4)`, where we need 4 times more processes. For instance, At 72 processes, `pipe-farm(4)` needs 288 processes for all the Workers and 8 processes for Emitters and Collectors that placeholders. Therefore, the cluster nodes end up being oversubscribed. They have more processes running than there are logical cores available in all nodes. Furthermore, it is difficult to map complex pipelines into MPI machine files as explained in Section 5.10.

```
1  [[spar::Stage,
2    spar::Input(
3     extract_dataset, extract_dataset_vec, extract_dataset_vecset),
4    spar::Output(
5     extract_dataset, extract_dataset_vec, extract_dataset_vecset,
6     result_lists, result_lists_data, result_lists_2d_data),
7    spar::Replicate(NTHREAD)]] {
8    //deserialize data:
9    struct all_data ret2;
10   ret2.extract.ds = extract_dataset;
11   ret2.extract.ds.vec = extract_dataset_vec.data();
12   ret2.extract.ds.vecset = extract_dataset_vecset.data();
13   //perform query
14   //serialize
15   result_lists = ret2.second.vec.result.u.lists;
16   result_lists_data = std::vector<cass_list_t>(result_lists.data,
        result_lists.data + result_lists.size);
17   for (auto list: result_lists_data) {
18     result_lists_2d_data.push_back(
19       std::vector<cass_list_entry_t>(list.data, list.data + list.size
           ));
20   }
21 }
```

Listing 6.26 – Making serialization easier with C++ std::vector.

Moreover, there is more communication needed for 288 processes (216 processes for `pipe-farm(3)`), which may cause performance penalities. DSPARLIB does not use on-demand scheduling between Workers and Collectors. The Workers simply send their results to the Collector. This can also contribute for poor performance in both `pipe-farm(4)` and `pipe-farm(3)` since not all of the computation graph is using on-demand scheduling. Load balancing issues can also incur as some stages might run faster than others and load may vary on each stream data item.

Using `std::vector` for serialization in `pipe-farm-stl(4)` caused no significant performance difference compared to `pipe-farm(4)`. The strategy in `pipe-farm(4)` involves copying the array. In practice, the `pipe-farm-stl(4)` also copies the array because the `std::vector` constructor used in Listing 6.26 copies the data as well. Therefore, the overhead of both operations is similar.

6.7 Programmability analysis

In this section, we discuss the programmability of SPAR-distributed compared to DSPARLIB and pure MPI versions. Table 6.7 tabulates the number of source lines of code for each program without considering comments. Before measuring the lines of code, we deleted all empty lines and comments and applied automatic code formatting with clang-tidy. The Serial/Original version refers to versions written to run in a single thread. In the case of PBZIP, the original PBZIP program can run in multiple threads, but not distributed.

Table 6.1 – SLOC for 4 applications

	Mandelbrot	**Lane Detection**	**PBZIP2**	**Primes**	**Ferret**
Serial/Original	42	114	1388	35	242
MPI On-Demand	147	302	1880	329	N/A
MPI Round-Robin	97	297	N/A	324	N/A
DSPARLIB	161	264	1746	83	N/A
SPAR-multicore	45	117	1909	43	349
SPAR-distributed	**45**	**141**	**1911**	**43**	**376**

For the Mandelbrot application, both SPAR-multicore and SPAR-distributed versions have the same annotations. Notice that the DSPARLIB requires more lines of code than MPI. DSPARLIB has more lines because every stage wrapper is a class and each stage requires more code intrusion. In DSPARLIB, changing from round-robin to on-demand is just a single method call. If we analyze the benefits of DSPARLIB purely by source lines of code, there is no visual benefit for the Mandelbrot application. However, the MPI code exposes many details such as scheduling and serialization via MPI sends and receives, while in DSPARLIB it is hidden.

For the Lane Detection program, there is a significant difference between SPAR-multicore and SPAR-distributed due to the problems mentioned in Section 6.8. We need to add code that ensures the files are opened and closed by the correct processes. When we compare DSPARLIB and the MPI versions, DSPARLIB uses fewer lines of code.

The original PBZIP version is 1388 lines of code. The SPAR-multicore and SPAR-distributed versions have approximately the same number of lines of code. Their differences are due to the new array syntax. The MPI version is very different and uses a master-slave model to distribute the data and computation. The DSPARLIB version is derived from a previous version parallelized with FastFlow.

The prime numbers MPI program is 35 lines for the serial version. For the MPI versions, the code is much longer because there are more features such as command-line parsing. Both SPAR-multicore and SPAR-distributed use the same code with 43 lines.

Ferret does not have an MPI or DSParLib version. Due to that, we only measured the original, SPar-distributed, and SPar-multicore versions. Ferret uses more lines of code in the SPar-distributed version due to serialization issues explained in Section 6.6.

6.8 Limitations

Although we have demonstrated that SPar-distributed is effective in parallelizing stream processing programs, this section elaborates on the limitations of SPar-distributed. These limitations need a significant effort to be resolved.

In MPI, all processes need to call `MPI_Init(int* argc, char*** argv)`. Since MPI-2 does not require the arguments to be valid, the user can pass `NULL` to them. SPar-distributed passes `NULL` to these arguments when using *static process allocation*. However, with *dynamic process allocation*, we need to call `MPI_spawn` with the same `argc` and `argv` as the program was initialized.

```
1
2  void start() {
3    [[spar::ToStream]] {
4      ...
5      [[spar::Stage, spar::Input(x)]] {
6        ...
7      }
8    }
9  }
10
11 int main(int argc, char** argv) {
12   start();
13 }
```

Listing 6.27 – SPar-distributed process spawning problem.

Consider the code in Listing 6.27, and assume the code is compiled with the `-spar_process_allocation_dynamic` flag. The code poses a problem. The `[[spar::ToStream]]` does not have access to the `argc` and `argv`. There is no portable way to get the `argc` and `argv` outside the `main` function, consequently, SPar-distributed will fail when compiling the program if the code does not have the `argc` and `argv` variables in the current scope. The code in Listing 6.27 can be solved by passing the `argc` and `argv` variables to the `start` function.

In this case, SPar-distributed could attempt to find the `main` function and save the arguments in a global variable. After, DSParLib could use during process spawning. However, the compiler may have been run in a source file without a `main` function. In Linux, it is possible to register a function in the `".init_array"` section of the ELF binary, which runs before the `main` function and get the `argc` and `argv` arguments in this function. However, our

tentative to use it resulted in segmentation fault errors. Since there is no portable way to get these arguments outside `main`, this problem is left unsolved. For now, we assume there are `argc` and `argv` variables in the scope of the `ToStream` block. If not, then the code will fail in the compilation. We believe this problem can be solved but requires more effort.

Another problem is opening and closing files from different processes as well as managing file handles. We had this problem in the Lane Detection application. In Listing 6.28, the `open` function is executed in all *all processes* if not properly managed. If all processes are running in an NFS (network file system), every process will have access to the same files. This might be a problem. In this example, we open the file and let the Collector stage write data to the file. However, all processes will attempt to close the file as well.

```
1  FILE file;
2  int main(int argc, char** argv) {
3    file = open(...); /*Opens file in all processes*/
4    [[spar::ToStream]] {
5      ...
6      [[spar::Stage, spar::Input(x), spar::Output(x), spar::Replicate()
           ]] {
7        ...
8      }
9      [[spar::Stage, spar::Input(x)]] {
10         write(file, ...);
11     }
12   }
13   close(file);
14 }
```

Listing 6.28 – Opening and closing files.

In the Lane Detection program, the file is a `cv::VideoWriter` object. The destructor will attempt to write data to the file before closing it. Since all processes opened the file, every process will also write data to this file. In an NFS, it is the same file. Some video players (like VLC) can still reproduce the desired result even with unnecessary data, but this might cause problems for other file types.

```
1  VideoWriter result;
2  int main(int argc, char** argv) {
3    result.open("result.avi", ...);
4    [[spar::ToStream]] {
5      [[spar::Stage, spar::Input(x), spar::Output(image), spar::
           Replicate()]] { ... }
6      [[spar::Stage, spar::Input(image)]] {
7          result.write(image);
8      }
9    }
10 }
```

Listing 6.29 – Opening and closing files, OpenCV example.

To circumvent this problem, we currently keep track of which frame is being received and open the file on the first frame. The workaround is shown in Listing 6.30. The destructor of the object only writes data to the file if the file is opened. However, only the Collector will open the file and no extra invalid data will be written to the file. Currently, there is no way to express this kind of “local resource” in SPAR-distributed. Since the goal of this work is to add support for distributed computing in SPAR without changing the abstraction level as much as possible, this problem is also not solved. In these cases, the programmer must be aware of the distributed environment where the program will be executed.

```
1  VideoWriter result;
2  int frame = 0;
3  int main(int argc, char** argv) {
4
5    [[spar::ToStream]] {
6      ...
7      [[spar::Stage, spar::Input(x), spar::Output(image), spar::
           Replicate()]] {
8        ...
9      }
10     [[spar::Stage, spar::Input(image)]] {
11         if (frame == 0) {
12             result.open("result.avi", ...);
13         }
14         result.write(image);
15         frame++;
16     }
17   }
18 }
```

Listing 6.30 – Workaround for opening and closing files in multiple processes.

6.9 Final remarks

In this chapter, we introduced the distributed version of SPAR. We have successfully generated DSPARLIB code for real-world applications and tested the performance. We have shown that the same SPAR language can be used for different architectures such as shared and distributed-memory architectures.

Although there are some limitations to SPAR-distributed, especially when working with files as described in Section 6.8, we consider that the goals have been achieved. Many programs have been implemented with SPAR-distributed and complex applications such as PBZIP2 only required the user to specify array sizes using the new `spar::Input(arr[size])` syntax. It is possible to implement support for more C++ STL types later if needed.

SPAR-distributed present minimal performance differences for the tested workloads with respect to the handwritten DSPARLIB. Furthermore, we demonstrated in Section

6.6 that SPAR-distributed also works for more robust pipeline compositions. The performance degradation were observed when pipelines are complex due to performance issues in DSPARLIB regarding load balancing between sequences of Farm patterns. Future works can improve performance experiments as well as implement better load balancing for complex pipeline compositions.

7. CONCLUSION

In this work, we tackled the problem of providing distributed stream parallelism exploitation with higher-level programming abstractions. This topic is especially relevant as demands for stream processing grew in the last decade. Parallel programming abstractions for stream parallelism can improve productivity with small performance penalties for multicore machines when designing them properly [GDTF15, Gri16, GHDF18a, GHDF17]. However, such benefits for distributed-memory architectures are not clearly addressed by the state-of-the-art parallel programming abstractions such as MPI.

We introduced DSParLib, a new skeleton-based library for distributed structured stream parallelism. DSParLib can run in clusters with Open MPI installed. It offers higher-level and reusable pattern-based abstractions to increase developer's productivity compared to MPI. We demonstrated that it efficiently implements communication, serialization, and work scheduling abstractions, which enables application programmers to scale their stream processing applications with less effort compared to common MPI implementations. It is worth mentioning that serialization is often not flexible enough or even not studied in state-of-the-art libraries. DSParLib allows efficient zero-copy serialization using simpler programming abstractions. Also, DSParLib extensible features will allow the implementation of future improvements for performance optimizations and new abstractions.

Higher-level stream parallelism abstractions were successfully addressed for distributed-memory architectures by extending SPar's language and compiler roles. By using DSParLib as the runtime, the SPar's compiler source-to-source transformation rules did not require a redesign. We were able to preserve the easy of use SPar's language syntax and semantic that has been proven to be productive on shared-memory architectures. SPar compiler was able to generate parallel code automatically without performance penalties compared to handwritten DSParLib stream programs. Thanks to all these achievements, SPar has become a valuable option to application programmers and the first annotation-based language for expressing stream parallelism in C++ programs with support to distributed-memory architectures, avoiding significant sequential code refactoring to enable parallel execution on clusters.

Our work can be further improved in different ways. The following items present opportunities for continuous advancing this research area:

- **Runtime support for hybrid parallelism exploitation**: DSParLib currently supports distributed-memory only. We believe there is potential for more performance when using a hybrid approach, avoiding communication and serialization costs when possible. Any work towards hybrid support must keep the level of abstraction currently offered by DSParLib. We believe SPar can reap the benefits of this research without significant changes to the compiler if the abstractions are left unchanged. Furthermore, the

support to hybrid shared, distributed, or heterogeneous (GPUs - Graphics Processing Units, TPUs - Tensor Processing Units or FPGAS - Field-Programmable Gate Array) architectures is interesting for programmers efficiently exploit the available parallelism.

- **Fault-tolerance and delivery guarantees**: Robust applications need to be fault-tolerant. This is especially true in distributed environments, where there is more opportunity for hardware and software failures. While Open MPI has some fault-tolerance mechanisms, it does not guarantee delivery of messages. However, this is an important feature, especially in the context of big-data stream processing. We believe future works in this area turn DSPARLIB more reliable.

- **Autonomic computing research**: MPI allows dynamic processes creation at runtime, opening the door for autonomic computing research in DSPARLIB. There is ongoing research on autonomic computing for SPAR [VGS+18] focused on shared-memory architectures. Future works can explore autonomic computing for distributed-memory architectures with DSPARLIB and SPAR.

- **Language and runtime support for windowing parallel patterns**: Future works can investigate whether it is possible to create annotations for windowing patterns [DMM17] in SPAR. This would require substantial research effort in language design and runtime support. Such annotations need to maintain the same level of abstraction currently found in SPAR while targeting different parallel architectures, which may be quite challenging.

- **Data-parallel patterns and arbitrary nesting**: DSPARLIB currently supports nesting only Farm patterns inside a Pipeline. Some computations are better expressed as a data-parallel computation using patterns such as Map and Reduce. Support for data-parallel patterns would make DSPARLIB a more robust library. Future works can also implement support for arbitrary nesting of these patterns in DSPARLIB.

REFERENCES

[ABW06] Arasu, A.; Babu, S.; Widom, J. “The CQL Continuous Query Language: Semantic Foundations and Query Execution”, *The VLDB Journal*, vol. 15–2, June 2006, pp. 121–142.

[ACD+12] Aldinucci, M.; Campa, S.; Danelutto, M.; Kilpatrick, P.; Torquati, M. “Targeting Distributed Systems in FastFlow”. In: Euro-Par Parallel Processing Workshops, 2012, pp. 47–56.

[AD99] Aldinucci, M.; Danelutto, M. “Stream Parallel Skeleton Optimization”. In: International Conference on Parallel and Distributed Computing and Systems, 1999, pp. 955–962.

[ADKT17] Aldinucci, M.; Danelutto, M.; Kilpatrick, P.; Torquati, M. “Fastflow: High-Level and Efficient Streaming on Multicore”. Hoboken, United States: John Wiley & Sons, Ltd, 2017, chap. 13, pp. 261–280.

[AGT14] Andrade, H. C. M.; Gedik, B.; Turaga, D. S. “Fundamentals of Stream Processing: Application Design, Systems, and Analytics”. New York, United States: Cambridge University Press, 2014, 1st ed., 558p.

[Akk19] Akka.NET Project. “Streams Introduction | Akka.NET Documentation”. Source: https://getakka.net/articles/streams/introduction.html, July 2019.

[Apa19] Apache Storm. “Apache Storm”. Source: https://storm.apache.org, July 2019.

[BAJ+16] 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: IEEE International Conference on Big Data, 2016, pp. 172–183.

[BBH+13] 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. 27–3, August 2013, pp. 244–254.

[BCC+97] Blackford, L. S.; Choi, J.; Cleary, A.; D’Azevedo, E.; Demmel, J.; Dhillon, I.; Dongarra, J.; Hammarling, S.; Henry, G.; Petitet, A.; Stanley, K.; Walker, D.; Whaley, R. C. “ScaLAPACK Users’ Guide”. Philadelphia, United States: Society for Industrial and Applied Mathematics, 1997, 1st ed., 345p.

[CHR+03] Consel, C.; Hamdi, H.; Réveillère, L.; Singaravelu, L.; Yu, H.; Pu, C. “Spidle: A DSL Approach to Specifying Streaming Applications”. In: Generative Programming and Component Engineering, 2003, pp. 1–17.

[CKE+15] Carbone, P.; Katsifodimos, A.; Ewen, S.; Markl, V.; Haridi, S.; Tzoumas, K. “Apache Flink™: Stream and Batch Processing in a Single Engine”, *IEEE Data Eng. Bull.*, vol. 38, December 2015, pp. 28–38.

[Col89] Cole, M. I. “Algorithmic Skeletons: Structured Management of Parallel Computation”. Glasgow, United Kingdom: University of Glasgow, 1989, 1st ed., 201p.

[Col04] Cole, M. “Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming”, *Parallel Computing*, vol. 30–3, March 2004, pp. 389–406.

[CPHP87] Caspi, P.; Pilaud, D.; Halbwachs, N.; Plaice, J. A. “LUSTRE: A Declarative Language for Real-Time Programming”. In: ACM SIGACT-SIGPLAN Symposium on Principles of programming languages, 1987, pp. 178–188.

[Cur18] Curtis, J. “A Comparison of Real Time Stream Processing Frameworks”, Master’s Thesis, Dublin Institute of Technology, 2018, 106p.

[DGS+16] Danelutto, M.; Garcia, J. D.; Sanchez, L. M.; Sotomayor, R.; Torquati, M. “Introducing Parallelism by Using REPARA C++11 Attributes”. In: Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, 2016, pp. 354–358.

[DM98] Dagum, L.; Menon, R. “OpenMP: an industry standard API for shared-memory programming”, *Computational Science & Engineering, IEEE*, vol. 5–1, January 1998, pp. 46–55.

[DMM17] De Matteis, T.; Mencagli, G. “Parallel Patterns for Window-Based Stateful Operators on Data Streams: An Algorithmic Skeleton Approach”, *International Journal of Parallel Programming*, vol. 45–2, April 2017, pp. 382–401.

[dRADFG17] del Rio Astorga, D.; Dolz, M. F.; Fernández, J.; García, J. D. “A generic parallel pattern interface for stream and data processing”, *Concurrency and Computation: Practice and Experience*, vol. 29–24, May 2017, pp. 1–12.

[Dro17] Drocco, M. “Parallel Programming with Global Asynchronous Memory: Models, C++ APIs and Implementations”, Ph.D. Thesis, University of Torino, Italy, 2017, 140p.

[DS00] Danelutto, M.; Stigliani, M. “SKElib: Parallel Programming with Skeletons in C”. In: Euro-Par Parallel Processing Workshops, Bode, A.; Ludwig, T.; Karl, W.; Wismüller, R. (Editors), 2000, pp. 1175–1184.

[DT16] Denis, A.; Trahay, F. “MPI Overlap: Benchmark and Analysis”. In: International Conference on Parallel Processing, 2016, pp. 258–267.

[EK14] Ernsting, S.; Kuchen, H. “A Scalable Farm Skeleton for Hybrid Parallel and Distributed Programming”, *International Journal of Parallel Programming*, vol. 42–6, December 2014, pp. 968–987.

[Fas19] FastFlow. “Fastflow Architecture”. Source: http://calvados.di.unipi.it/dokuwiki/doku.php/ffnamespace:architecture, July 2019.

[Fli19] Flink. “Apache Flink 1.8 Documentation: Distributed Runtime Environment”. Source: https://ci.apache.org/projects/flink/flink-docs-release-1.8/concepts/runtime.html, July 2019.

[For12] Forum, M. “MPI: A Message-Passing Interface Standard”, Technical Report, University of Tennessee, Knoxville, United States, 2012, 852p.

[FSCL06] Falcou, J.; Sérot, J.; Chateau, T.; Lapresté, J. “Quaff: Efficient C++ design for parallel skeletons”, *Parallel Computing*, vol. 32–7, September 2006, pp. 604–615.

[Gar18] García, J. D. “GRPPI text capitalization example”. Source: https://github.com/arcosuc3m/grppi/blob/doc03b/samples/farm/capitalize/main.cpp, July 2019.

[GDTF15] 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.

[GDTF17] Griebler, D.; Danelutto, M.; Torquati, M.; Fernandes, L. G. “SPar: A DSL for High-Level and Productive Stream Parallelism”, *Parallel Processing Letters*, vol. 27–01, March 2017, pp. 1–20.

[GF17] Griebler, D.; Fernandes, L. G. “Towards Distributed Parallel Programming Support for the SPar DSL”. In: International Conference on Parallel Computing, 2017, pp. 563–572.

[GFB+04] Gabriel, E.; Fagg, G. E.; Bosilca, G.; Angskun, T.; Dongarra, J. J.; Squyres, J. M.; Sahay, V.; Kambadur, P.; Barrett, B.; Lumsdaine, A.; Castain, R. H.; Daniel, D. J.; Graham, R. L.; Woodall, T. S. “Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation”. In: Recent Advances in

Parallel Virtual Machine and Message Passing Interface, Kranzlmüller, D.; Kacsuk, P.; Dongarra, J. (Editors), 2004, pp. 97–104.

[GHDF17] Griebler, D.; Hoffmann, R. B.; Danelutto, M.; Fernandes, L. G. “Higher-Level Parallelism Abstractions for Video Applications with SPar”. In: International Conference on Parallel Computing, 2017, pp. 698–707.

[GHDF18a] 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. 47–1, February 2018, pp. 253–271.

[GHDF18b] Griebler, D.; Hoffmann, R. B.; Danelutto, M.; Fernandes, L. G. “Stream Parallelism with Ordered Data Constraints on Multi-Core Systems”, *Journal of Supercomputing*, vol. 75–8, July 2018, pp. 4042–4061.

[Gil04] Gilchrist, J. “Parallel Compression with BZIP2”. In: International Conference on Parallel and Distributed Computing and Systems, 2004, pp. 559–564.

[Gri16] Griebler, D. “Domain-Specific Language & Support Tool for High-Level Stream Parallelism”, Ph.D. Thesis, Faculdade de Informática - PPGCC - PUCRS, Porto Alegre, Brazil, 2016, 243p.

[GSV+18] Griebler, D.; Sensi, D. D.; Vogel, A.; Danelutto, M.; Fernandes, L. G. “Service Level Objectives via C++11 Attributes”. In: Euro-Par Parallel Processing Workshops, 2018, pp. 745–756.

[GV17] Grant, W. S.; Voorhies, R. “Cereal - A C++11 library for serialization”. Source: https://github.com/USCiLab/cereal, July 2019.

[GVL10] González-Vélez, H.; Leyton, M. “A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers”, *Software: Practice and Experience*, vol. 40–12, November 2010, pp. 1135–1160.

[GVS+19] Griebler, D.; Vogel, A.; Sensi, D. D.; Danelutto, M.; Fernandes, L. G. “Simplifying and implementing service level objectives for stream parallelism”, *Journal of Supercomputing*, vol. 76, June 2019, pp. 4603–4628.

[HBB+18] Hirzel, M.; Baudart, G.; Bonifati, A.; Della Valle, E.; Sakr, S.; Akrivi Vlachou, A. “Stream Processing Languages in the Big Data Era”, *ACM SIGMOD Record*, vol. 47–2, December 2018, pp. 29–40.

[Her19] Heron. “Heron Documentation”. Source: https://apache.github.io/incubator-heron/docs/concepts/architecture/, July 2019.

[HGDF20] Hoffmann, R. B.; Griebler, D.; Danelutto, M.; Fernandes, L. G. “Stream Parallelism Annotations for Multi-Core Frameworks”. In: Brazilian Symposium on Programming Languages, 2020, pp. 48–55.

[Hof20] Hoffmann, R. B. “Stream Parallelism Annotations for Autonomic OpenMP Code Generation”, Technical Report, School of Technology - PPGCC - PUCRS, Porto Alegre, Brazil, 2020, 55p.

[HSG17] Hirzel, M.; Schneider, S.; Gedik, B. “SPL: An Extensible Language for Distributed Stream Processing”, *ACM Transactions on Programming Languages and Systems*, vol. 39–1, March 2017, pp. 1–39.

[Hü15] Hüske, F. “Juggling with Bits and Bytes”. Source: https://flink.apache.org/news/2015/05/11/Juggling-with-Bits-and-Bytes.html, July 2019.

[KBF+15] 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: ACM SIGMOD International Conference on Management of Data, 2015, pp. 239–250.

[KHAL+14] Kaiser, H.; Heller, T.; Adelstein-Lelbach, B.; Serio, A.; Fey, D. “HPX: A Task Based Programming Model in a Global Address Space”. In: International Conference on Partitioned Global Address Space Programming Models, 2014, pp. 1–11.

[KNR11] Kreps, J.; Narkhede, N.; Rao, J. “Kafka: A Distributed Messaging System for Log Processing”. In: NetDB Workshop, 2011, pp. 1–7.

[LB07] Lin, C.-k.; Black, A. P. “DirectFlow: A Domain-Specific Language for Information-Flow Systems”. In: European Conference on Object-Oriented Programming, Ernst, E. (Editor), 2007, pp. 299–322.

[LGFd+19] López-Gómez, J.; Fernández Muñoz, J.; del Rio Astorga, D.; Dolz, M. F.; Garcia, J. D. “Exploring stream parallel patterns in distributed MPI environments”, *Parallel Computing*, vol. 84, May 2019, pp. 24–36.

[LGMF15] Ledur, C.; Griebler, D.; Manssour, I.; Fernandes, L. G. “Towards a Domain-Specific Language for Geospatial Data Visualization Maps with Big Data Sets”. In: ACS/IEEE International Conference on Computer Systems and Applications, 2015, pp. 1–8.

[LGMF17] Ledur, C.; Griebler, D.; Manssour, I.; Fernandes, L. G. “A High-Level DSL for Geospatial Visualizations with Multi-core Parallelism Support”. In: IEEE Computer Society Signature Conference on Computers, Software and Applications, 2017, pp. 298–304.

[Lig19] LightBend. “Streams - Akka Documentation”. Source: https://doc.akka.io/docs/akka/current/stream/index.html, July 2019.

[LJW+06] Lv, Q.; Josephson, W.; Wang, Z.; Charikar, M.; Li, K. “Ferret: a toolkit for content-based similarity search of feature-rich data”, *Operating Systems Review - SIGOPS*, vol. 40, January 2006, pp. 317–330.

[Lö20] Löff, J. H. “Aumentando a Expressividade e Melhorando a Geração de Código Paralelo para o Paradigma de Paralelismo de Stream em Arquiteturas Multi-core”, Technical Report, School of Technology - PPGCC - PUCRS, Porto Alegre, Brazil, 2020, 76p.

[Mic20] Microsoft. “Introduction to PLINQ”. Source: https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/introduction-to-plinq, September 2020.

[MIEH06] Matsuzaki, K.; Iwasaki, H.; Emoto, K.; Hu, Z. “A Library of Constructive Skeletons for Sequential Style of Parallel Programming”. In: International Conference on Scalable Information Systems, 2006, pp. 1–13.

[Mis17] Misale, C. “PiCo: A Domain-Specific Language for Data Analytics Pipelines”, Ph.D. Thesis, University of Torino, Italy, 2017, 169p.

[MKI+04] Matsuzaki, K.; Kakehi, K.; Iwasaki, H.; Hu, Z.; Akashi, Y. “A Fusion-Embedded Skeleton Library”. In: Euro-Par Parallel Processing Workshops, Danelutto, M.; Vanneschi, M.; Laforenza, D. (Editors), 2004, pp. 644–653.

[MMP10] Mancini, E. P.; Marsh, G.; Panda, D. K. “An MPI-Stream Hybrid Programming Model for Computational Clusters”. In: IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, 2010, pp. 323–330.

[MnDdRA+18] Muñoz, J. F.; Dolz, M. F.; del Rio Astorga, D.; Cepeda, J. P.; García, J. D. “Supporting MPI-distributed Stream Parallel Patterns in GrPPI”. In: European MPI Users’ Group Meeting, 2018, pp. 1–10.

[MRR12] McCool, M.; Robison, A. D.; Reinders, J. “Structured Parallel Programming: Patterns for Efficient Computation”. Waltham, United States: Elsevier, 2012, 1st ed., 432p.

[MSM04] Mattson, T.; Sanders, B.; Massingill, B. “Patterns for Parallel Programming”. Addison-Wesley Professional, 2004, 1st ed., 355p.

[NiF19] NiFi. “Apache NiFi”. Source: https://nifi.apache.org/index.html, July 2019.

[NPP+17] Noghabi, S. A.; Paramasivam, K.; Pan, Y.; Ramesh, N.; Bringhurst, J.; Gupta, I.; Campbell, R. H. “Samza: Stateful Scalable Stream Processing at LinkedIn”, *Proc. VLDB Endow.*, vol. 10–12, August 2017, pp. 1634–1645.

[Omp20a] OmpSs. “OmpSs-2@Cluster programming model”. Source: https://github.com/bsc-pm/nanos6/blob/master/docs/cluster/Cluster.md, August 2020.

[Omp20b] OmpSs. “The OmpSs Programming Model”. Source: https://pm.bsc.es/ompss, August 2020.

[Ope20] OpenMP. “Open Multi-Processing API specification for parallel programming”. Source: https://www.openmp.org/, August 2020.

[Ora20] Oracle. “java.util.stream (Java Platform SE 8)”. Source: https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html, September 2020.

[PC13] Pop, A.; Cohen, A. “OpenStream: Expressiveness and Data-Flow Compilation of OpenMP Streaming Programs”, *ACM Trans. Archit. Code Optim.*, vol. 9–4, January 2013, pp. 1–25.

[PGF19] Pieper, R.; Griebler, D.; Fernandes, L. G. “Structured Stream Parallelism for Rust”. In: Brazilian Symposium on Programming Languages, 2019, pp. 54–61.

[PMG+17] Peng, I. B.; Markidis, S.; Gioiosa, R.; Kestor, G.; Laure, E. “MPI Streams for HPC Applications”, *CoRR*, vol. 1708.01306, November 2017, pp. 75–92.

[PML+15] Peng, I. B.; Markidis, S.; Laure, E.; Holmes, D.; Bull, M. “A Data Streaming Model in MPI”. In: ExaMPI Workshop at the International Conference on High Performance Computing, Networking, Storage and Analysis, 2015, pp. 1–10.

[Ray20] Rayon. “Rayon - Rust”. Source: https://docs.rs/rayon/1.4.0/rayon/, September 2020.

[Rei07] Reinders, J. “Intel Threading Building Blocks”. Sebastopol, United States: O’Reilly & Associates, Inc., 2007, 1st ed., 332p.

[Roc20] Rockenbach, D. A. “High-Level Programming Abstractions for Stream Parallelism on GPUs”, Master’s thesis, School of Technology - PPGCC - PUCRS, Porto Alegre, Brazil, 2020, 163p.

[Spa19] Spark. “Spark Streaming Programming Guide”. Source: https://spark.apache.org/docs/latest/streaming-programming-guide.html, July 2019.

[Sto19] Storm. “Understanding the Parallelism of a Storm Topology”. Source: https://storm.apache.org/releases/2.0.0/Understanding-the-parallelism-of-a-Storm-topology.html, July 2019.

[Str06] StreamIt. “StreamIt Cookbook”. Source: http://groups.csail.mit.edu/cag/streamit/papers/streamit-cookbook.pdf, July 2019.

[Thr19] Thrill. “Thrill word count simple example”. Source: https://github.com/thrill/thrill/blob/master/examples/word_count/word_count_simple.cpp, July 2019.

[TKA02] Thies, W.; Karczmarek, M.; Amarasinghe, S. P. “StreamIt: A Language for Streaming Applications”. In: International Conference on Compiler Construction, 2002, pp. 179–196.

[TKG+01] Thies, W.; Karczmarek, M.; Gordon, M. I.; Maze, D. Z.; Wong, J.; Hoffman, H.; Brown, M.; Amarasinghe, S. “StreamIt: A Compiler for Streaming Applications”, Technical Report, Massachusetts Institute of Technology, Cambridge, United States, 2001, 11p.

[TS06] Tanenbaum, A. S.; Steen, M. v. “Distributed Systems: Principles and Paradigms”. Upper Saddle River, United States: Prentice-Hall, Inc., 2006, 2nd ed., 702p.

[VGF21] Vogel, A.; Griebler, D.; Fernandes, L. G. “Providing High-Level Self-Adaptive Abstractions for Stream Parallelism on Multicores”, *Software: Practice and Experience*, vol. 51–6, January 2021, pp. 1194–1217.

[VGS+18] Vogel, A.; Griebler, D.; Sensi, D. D.; Danelutto, M.; Fernandes, L. G. “Autonomic and Latency-Aware Degree of Parallelism Management in SPar”. In: Euro-Par Parallel Processing Workshops, 2018, pp. 28–39.

[VRJ+20] 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: *High Performance Computing Systems*, Cham: Springer, 2020, *Communications in Computer and Information Science (CCIS)*, vol. 1171, pp. 102–116.

[WR09] Wagner, A.; Rostoker, C. “A lightweight stream-processing library using MPI”. In: IEEE International Symposium on Parallel & Distributed Processing, 2009, pp. 1–8.

[ZCF+10] Zaharia, M.; Chowdhury, M.; Franklin, M. J.; Shenker, S.; Stoica, I. “Spark: Cluster Computing with Working Sets”. In: USENIX conference on Hot topics in cloud computing, 2010, pp. 10–10.

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Graduação
Av. Ipiranga, 6681 - Prédio 1 - 3º. andar
Porto Alegre - RS - Brasil
Fone: (51) 3320-3500 - Fax: (51) 3339-1564
E-mail: prograd@pucrs.br
Site: www.pucrs.br