International Journal of Parallel Programming (2022) 50:454–485
https://doi.org/10.1007/s10766-022-00737-2

DSParLib: A C++ Template Library for Distributed Stream Parallelism

Júnior Löff[1] · Renato B. Hoffmann[1] · Ricardo Pieper[1] · Dalvan Griebler[1,2] · Luiz G. Fernandes[1]

Received: 5 October 2021 / Accepted: 19 September 2022 / Published online: 29 October 2022

Abstract
Stream processing applications deal with millions of data items continuously generated over time. Often, they must be processed in real-time and scale performance, which requires the use of distributed parallel computing resources. In C/C++, the current state-of-the-art for distributed architectures and High-Performance Computing is Message Passing Interface (MPI). However, exploiting stream parallelism using MPI is complex and error-prone because it exposes many low-level details to the programmer. In this work, we introduce a new parallel programming abstraction for implementing distributed stream parallelism named DSParLib. Our abstraction of MPI simplifies parallel programming by providing a pattern-based and building block-oriented development to inter-connect, model, and parallelize data streams found in modern applications. Experiments conducted with five different stream processing applications and the representative PARSEC Ferret benchmark revealed that DSParLib is efficient and flexible. Also, DSParLib achieved similar or better performance, required less coding, and provided simpler abstractions to express parallelism with respect to handwritten MPI programs.

Keywords Parallel programming · Distributed systems · Stream processing · Parallel patterns · MPI

✉ Dalvan Griebler
dalvan.griebler@pucrs.br

[1] School of Technology, Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre, Brazil

[2] Laboratory of Advanced Research on Cloud Computing (LARCC), Três de Maio Faculty (SETREM), Três de Maio, Brazil

1 Introduction

Stream Processing is a computing model to perform real-time processing in live data streams [4]. Its importance is evidenced by the common necessity for filtering data sources that often produce data on the scale of millions of items per day. In stream processing, this data can be viewed as a potentially infinite flow of items. Moreover, the items that flow through the data stream can have a varied workload which may spike or slow down the quality of service. In some cases, on-the-fly processing may be required to leverage the data utility window fully. Therefore, adapting the stream processing systems to this dynamic environment while striving for efficiency is challenging. In fact, over the last few years, several improvements have been made toward distributed and scalable stream processing systems such as Apache Storm [5] and Apache Flink [9], which are written in high-level languages such as Java.

Thanks to ongoing advances, application programmers now have various options to choose from when developing parallel software. Languages such as Rust and C++ benefit from libraries that implement high-level parallel abstractions, like Rayon [33] and Intel Threading Building Block [34], and Rust-SSP [31, 32]. These solutions employ *parallel patterns* as API (Application Programming Interface) developed by experts in the field. They help the application programmer by hiding the complexity of parallelism while still delivering good performance and productivity solutions [10, 26, 27].

Clustered architectures introduce another level of communication hierarchy. They are characterized by multiple computational nodes connected via a communication network (e.g., Infiniband, wi-fi, etc.). In these cases, scaling the computation to satisfy the performance demand of modern stream processing applications requires distributed processing techniques. In High-Performance Computing (HPC), Message Passing Interface (MPI) is the state-of-the-art parallel API for implementing parallel C/C++ programs. However, using MPI is difficult for application developers since it exposes several low-level programming aspects. Therefore, programmers have to deal with exhaustive and error-prone parallelism concepts. Namely, data handling, process communication through message passing and synchronization models, fault tolerance, work scheduling, load balancing, and parallelism strategies. No framework or library in C++ provides all of these features, while in other programming domains, they are available. Therefore, as shown by our related work, programmers do not have reliable options that they can use. Moreover, porting all application codes from C++ to Java/Scala for using Apache Storm or Apache Flink may not be doable. Our work goes in this direction, in which we tackle this research gap and provide a library in C++ with better programmability aspects. However, we are not trying to compete against consolidated solutions (e.g., Flink and Storm); instead, our main contributions are to provide the first set of abstractions based on a structured parallel programming approach and evaluate our solution in terms of programmability and performance.

In summary, the goal of this research is helping to ease the process of developing distributed C++ parallel programs. We propose to handle the steep learning curve of MPI parallel programming by providing parallel patterns or algorithmic skeletons as higher-level abstractions. Parallel patterns are an already existing approach that uses structured parallel programming strategies to model common data flows that can easily be composed and comprehended by the programmers. In the state-of-the-art, similar approaches have been employed by FastFlow [1], GrPPI [12], eSkel [11], and Muesli [13]. In contrast, we provide a higher-level programming abstraction specifically targeting stream processing applications (see more details in Sect. 2). We also address the challenges inherent to distributed parallel programming. This work introduces programming abstractions to deal with data communication (serialization and message passing), synchronization between parallel processes via parallel patterns (protocols for message passing), and two scheduling algorithms for balanced and unbalanced workloads. Implementing these challenges in C++ is more complex than in languages such as Java and Python. Unlike these languages, data communication is significantly more difficult since there is no metadata about types available during runtime in C++. We summarize our contributions as follows:

- A high-level parallel programming abstraction for parallel and distributed stream parallelism implementation in C++ named DSParLib. Its main characteristics are transparent data communication, support for two different scheduling techniques, and abstracted synchronization between parallel processes.
- A parallel implementation of PARSEC's Ferret Benchmark [6] with DSParLib and its evaluation, as no distributed version is available.
- A comprehensive set of experiments for comparing our high-level abstraction (DSParLib) with respect to state-of-the-art handwritten MPI programs.

The remainder of this document is organized as follows. Section 2 discusses the related work. Then, Sect. 3 presents the implementation details of DSParLib and its API. Subsequently, Sect. 4 introduces the applications we used in our experiments and their respective parallelization techniques. Section 5 describes and analyses the experiments performed with different stream processing applications for comparing DSParLib with state-of-the-art handwritten MPI versions and evaluating pattern composition. Finally, Sect. 6 concludes with final remarks and future works.

2 Related Work

In this related work, we consider the parallel programming abstractions that support distributed stream parallelism in C++ targeting HPC systems. Therefore, we do not consider *Big Data* frameworks (e.g. Apache Flink, Akka Streams, Apache Storm, etc.) because they do not face the same programming language challenges. They are not related work, but we considered them background when designing our library. In contrast, our goal is to open research questions about HPC for leveraging distributed stream systems in a low-level programming language like C++.

FastFlow [2] is a high-level C++ pattern-based parallel programming library. FastFlow targets shared memory but also supports distributed computing. Our works are similar since both of them provide implementations for parallel patterns. Patterns are described in the literature long before the libraries implementing them were created [26]. DSParLib provides new implementations to the Pipeline and Farm patterns for stream processing, while FastFlow provides a wider set of patterns for different types of computation. The main differences between the implementations are that FastFlow supports multi-node systems using ZeroMQ as the transport layer and to create processes, while DSParLib uses MPI [1]. Also, DSParLib employs newer MPI specifications and dynamic process management to improve performance. Another difference is that programmers manually handle serialization in FastFlow while DSParLib provides abstractions to help programmers communicate data. For native C++ data types, DSParLib completely abstracts data communication, and in the case of custom data types, DSParLib provides high-level data abstractions via templates.

GrPPI (Generic Reusable Parallel Pattern Interface) [12] is a C++ library that implements composable and generic interfaces for parallel patterns. This means that the programmers only implements the parallel patterns once and chooses at compile time, which is the desired runtime that GrPPI should use. FastFlow, Intel TBB, and C++ Parallel STL (Standard Template Library) are available options. The support for distributed computing was studied by [24, 28] using MPI. However, the source code is unavailable, and we could not evaluate their solution.

Thrill [7] is an experimental C++14 framework for distributed stream processing. Instead of merging existing Big Data frameworks with HPC tools like others [3], their goal is to implement a new Big Data framework targeting HPC directly in C++. Networking in Thrill is done via TCP sockets and optionally via MPI. Currently, no delivery guarantees or fault-tolerance mechanisms are implemented. Thrill deals with data communication abstraction using the Cereal library [16], which provides a lightweight serializer that uses a similar interface to Boost archives [8].

The work of [25] provides a high-performance abstraction based on a **MPI hybrid** approach. The work is mainly concerned with supporting heterogeneous systems. The communication is achieved using both MPI and TCP sockets. The work also implements optimizations to reduce overheads, for example, using POSIX Threads instead of operating system processes. **MPI Lightweight** Stream [36] proposes a lightweight interface for MPI stream processing. The work focuses on communication optimizations and their functional correctness when using MPI for stream processing. They aim to create a library to support data flow via a directed acyclic graph (DAG). Likewise, they investigate employing a graph-coloring algorithm to properly label the DAG to arrange the `intra`- and `inter`-communicators for the Pipeline.

MPI Streams [29][30] proposes an extension to the Message Passing Interface standard for better supporting streaming applications. The work introduced a generic streaming communication model consisting of entities like producers, consumers, and message passing channels. Consumers support receiving data from multiple producers while computing them using a "first-come, first-served" approach. MPI Streams do not abstract message passing.

In addition to MPI-based libraries that extended MPI to leverage stream processing applications, other C/C++ skeleton libraries support the standard MPI

Table 1 A comparison of C++ programming and runtime libraries

Work	Stream patterns	Runtime	Data communication	Built-in schedulers	Message passing
FastFlow [1]	Yes	ZeroMQ	Zero-copy	Yes	Implicit
GrPPI [12]	Yes	MPI*	Boost*	Yes	Implicit
MPI Streams [29]	No	MPI	Low-level MPI	No	Explicit
MPI Hybrid [25]	No	MPI	Low-level MPI	No	Explicit
MPI Lightweight [36]	No	MPI	Low-level MPI	No	Explicit
Thrill [7]	No	TCP, MPI	“Cereal” library	Yes	Implicit
eSkel [11]	Yes	MPI	Low-level MPI	No	Implicit
Muesli [13]	Yes	MPI	Boost	Yes	Implicit
Quaff [14]	Yes	MPI	Boost	No	Implicit
DSParLib (Ours)	Yes	MPI	Zero-copy	Yes	Implicit

specification. The work in [15] performed an extensive survey on skeleton libraries, and we describe some of these libraries here. Unfortunately, many of them are no longer being maintained. Also, for the most part, they do not tackle programmability aspects and do not propose high-level abstractions for data communication.

ESkel [11] is from the same authors of Algorithmic Skeletons [10]. ESkel is a C library based on MPI that implements the parallel patterns Pipeline, Farm, and Divide-and-Conquer. Users of eSkel must be familiar with basic MPI concepts as well. Message passing 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 data type.

Muesli [13] implements MPI support for the Farm pattern in a hybrid environment using MPI and OpenMP. Muesli provides parallel and distributed data structures such as distributed arrays and matrices. It also allows serializing arbitrary data types. The programmers can implement an abstract class `MSL_Serializable` on the type that needs to be serialized. The interface requires the programmers to copy the data to a buffer provided by Muesli. **Quaff** [14] is a C++ template-based skeleton library aiming at reducing the overhead of object-oriented abstractions. According to to [14], libraries like Muesli frequently introduce overheads due to virtual function calls. The work supports MPI but does not elaborate on the serialization process.

Table 1 presents a summary of our findings regarding C++ libraries for distributed stream processing. The results show that many libraries lack implementations for stream parallel patterns since they target data-parallel applications only. Data communication is often done with low-level MPI or external libraries such as Boost and Cereal. Message passing is generally done implicitly, where the programmers do not need to know how programs communicate with each other. However, for libraries that extend MPI, communication is explicit.

DSParLib fills these gaps by introducing a new implementation of stream patterns with high-level data communication based on C++ templates, allowing zero-copy data communication. Moreover, DSParLib has implicit communication and a built-in scheduler. In addition, DSParLib provides a high-level programming abstraction for low-level MPI details. For example, it does not unnecessarily expose the programmers to raw pointers or explicit message passing protocols. DSParLib also ensures strong type-safety since we check at compile-time, through C++ templates, for possible incorrect usage.

3 DSParLib

This section presents how we conceived DSParLib (an acronym for Distributed Stream Parallelism Library). DSParLib[1] is a programming abstraction for expressing or implementing stream parallelism on C++ applications targeting distributed architectures. DSParLib abstractions over MPI simplify parallel programming via high-level and easy-to-use API. Section 3.1 presents the design principles and implementation choices of DSParLib. Section 3.2 shows how users can compose different stream parallel patterns and semi-arbitrary pattern nesting. Section 3.3 presents the basic components of the building block development approach. Then, the remaining Sects. 3.4–3.7 summarize how DSParLib works underneath our high-level abstractions.

3.1 Design Principles

Our main design principles are focused on programmability and efficiency. Regarding the programmability, we want DSParLib to be a parallel programming abstraction that is simple to use for application programmers. Therefore, we developed it as a safe and high-level API that checks for incorrect usage at compile time. This is done mostly using built-in checking mechanisms and features introduced in the C++11 standard for inferring data types. Also, DSParLib strives not to expose application programmers to raw pointers while ensuring that data types are semantically correct using C++ templates instead of `void*` pointers. The efficiency design principle is aligned with the HPC domain that considers scalability and performance.

We implemented DSParLib in C++, a widely used programming language. Through the notion of building blocks, we designed an intuitive API for DSParLib that allows the programmers to "wrap" existing code with few refactorings. We state that refactoring sequential code and wrapping it via DSParLib 's building blocks should be accomplished with safety and simplicity. For example, DSParLib guarantees the functional correctness of the pipeline data types implemented by the programmers so that a stage receiving an integer is inter-connected with a stage sending integers. Likewise, the semi-arbitrary composition of parallel patterns and each

[1] Available in: https://github.com/GMAP/DSParLib.

inter-connected operator is checked to validate the resulting data flow. Checkers are helpful in avoiding simple mistakes introduced by programmers.

The runtime is MPI, which is the standard for HPC. Our design principles were based on newer MPI versions, starting from MPI-2. MPI-2 equips programmers with new features for creating more flexible but complex strategies that impose additional challenges when writing distributed parallel code. In our case, MPI-2 can be considered a different parallel programming paradigm from the MPI-1 standard because it allows the creation and communication with new processes. We mainly employ MPI-2 in our work because it supports dynamic process management. In stream processing, the dynamic characteristics of this domain (workload spikes and slowdown, infinite and heterogeneous data, etc.) impose additional challenges in efficiently exploiting distributed resources when executing a streaming application. For that, creating and removing MPI processes during execution time complies with our plan for supporting auto-adaptive and fault-tolerance strategies in the future.

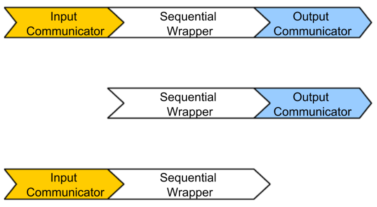
Furthermore, communication is a significant abstraction issue that needs to be addressed in C++. Instead of using a third-party library such as Boost or Cereal, DSParLib provides a new and lightweight strategy that abstracts message passing while allowing efficient and zero-copy data communication. Finally, the library is header-only, so there is no need to compile dynamic or static libraries separately. We also provide implementations of concepts commonly found in the literature, such as Pipeline and Farm, to provide ready-to-use parallel patterns [26]. We allow the semi-arbitrary composition of the Pipeline and Farm parallel patterns in which Farms can be included as Pipeline Stages. Currently, other pattern compositions are not supported.

3.2 Building Block Developing Approach

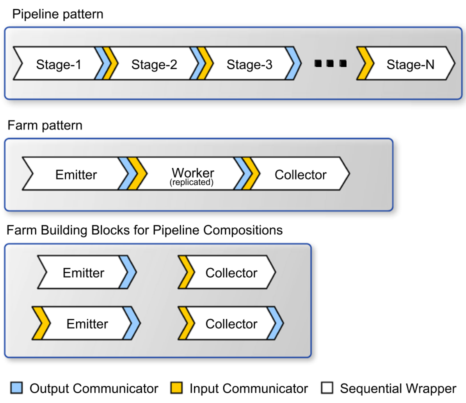
DSParLib is designed to develop parallel stream processing applications following the structured parallel programming paradigm. The main benefits are that many low-level parallelism details are hidden from application programmers. Developers are expected to implement parallel code using composable and reusable structures to design an assembly line for data stream processing. The building blocks terminology is popular in C++ among state-of-the-art structured parallel programming abstractions, such as Intel Threading Building Blocks (TBB) and FastFlow. We take inspiration from these libraries to design our DSParLib 's API.

In DSParLib, there are three possible compositions of building blocks, as illustrated in Fig. 1a. The basic component of a building block is the Sequential Wrapper (white block), which wraps the computational processing. More details about how this block works will be discussed later in Sect. 3.3. The other two blocks are the Input and Output communicators (yellow and blue), which are used to implement the message passing through the network. They can also be seen as blocks that wrap the mechanisms for sending and receiving data.

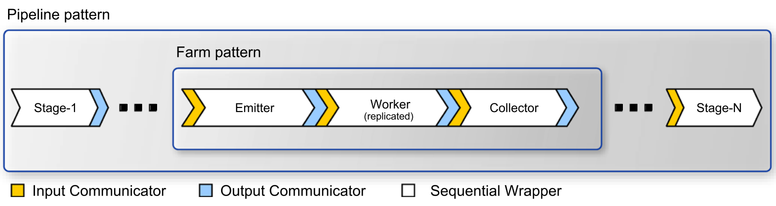
Each building block must have a Sequential Wrapper and optionally Input or Output communicator. Implementing data communicators depends on whether other building blocks precede or succeed the current building block. For example,



(a) DSPARLIB's building blocks.



(b) DSPARLIB's parallel patterns.



(c) Pattern composition/nesting in DSPARLIB.

Fig. 1 DSPARLIB composable and reusable building blocks

suppose a given building block is preceded by another one or a different parallel pattern. In that case, it must implement the Input communicator while the previous building block implements the Output communicator. The same happens if the building block is succeeded by another one or by a parallel pattern. Figure 1b, c illustrate the previous example and DSPARLIB inter-connected building blocks from a high-level point of view. Each couple of blue and yellow blocks is a network communication abstracted by DSPARLIB 's message passing mechanisms.

Figure 1b presents the available parallel patterns, which are Pipeline and Farm. For example, the Pipeline pattern starts with a Sequential Wrapper + Output communicator and ends with an Input communicator + Sequential Wrapper. Those are respectively representing the beginning and end of a stream. Additionally, all intermediate blocks implement both Input and Output communicators combined with Sequential Wrappers since they receive an input from the previous block and send an output to the next one. On the other hand, the Farm pattern is a particular case of the Pipeline containing three stages. Sometimes stages have intensive computations, and we want to replicate them to improve performance. We employ the Farm parallel pattern, where the intensive computation is assigned to the middle stage (Worker) that can be replicated. To maintain the functional correctness of the data flow, stream items require a scheduler (Emitter) to assign data for these parallel workers and a step that gathers the parallel Workers' results (Collector). The communication data flow moves from left to right.

In contrast to the ready-to-use parallel patterns, DSParLib supports semi-arbitrary pattern composition. It allows nesting Farms into Pipeline stages, as depicted in Fig. 1c. Other compositions are currently not supported. By default, the Farm pattern depicted in Fig. 1b does not require the Input and Output communicators. However, when a Farm pattern is added as a Pipeline stage (Fig. 1c), it may require new data communicators depending on where it is positioned. For example, when a Farm is positioned as a middle Pipeline stage, extra Input and Output communicators are mandatory for communicating data. Otherwise, when a Farm becomes the first or last stage of the Pipeline, it may not require input or output communicators. Both situations are represented by the optional building blocks illustrated in the bottom-most part of Fig. 1b.

Parallel patterns may be reused and are suitable for complex stream processing applications, especially when they follow an object-oriented approach. Imagine that each building block from DSParLib can be seen as a programming function that receives input parameters (Input Communicator), applies a computational step (Sequential Wrapper), and returns the results (Output Communicator). The same benefits perceived by using the object-oriented programming paradigm are enforced via structured parallel programming paradigms rather than ad-hoc parallelism.

3.3 Sequential Wrappers

The concept of a sequential code wrapper was introduced in the previous Sect. 3.2. In other words, the Sequential Wrapper is a `class` that wraps the sequential code of the application. Then, the wrapped code can be managed in terms of DSParLib 's building blocks. This further allows the programmers to combine the building blocks using ready-to-use parallel patterns. The main benefit is that parallel patterns already abstract many parallelism complexities, so the programmers do not need to implement them from scratch, like schedulers, message passing queues, synchronizations, etc. Moreover, DSParLib's patterns have built-in features and optimizations that the programmers can seamlessly turn on and off.

In Listing 1, we show an example of sequential code wrappers implemented in DSParLib. They process the accumulation of square roots from 0 to 9. First, the `Stage1` generates the data (lines 1 to 7). Each new loop iteration (line 4) emits a new data item (line 5). Then, the `Stage2` receives the items and emits their square root as a double (lines 10 to 16). Any C++ computation or function could replace the computation from line 13. The `Process()` method executes over each stream item received. Finally, the `Stage3` accumulates all the results (lines 18 to 24). In DSParLib, the inputs from the previous building block are received using the `Process(inputs)` and the resulting outputs are scheduled using the `Emit(outputs)` function. In this way, the lower-level data communication and message passing implementations are abstracted from the programmers.

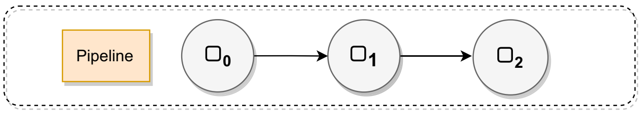


Fig. 2 Parallel activity graph example

```
 1 class Stage1: public wrapper <Nothing, int> {
 2 public:
 3   void Produce() override {
 4     for(int i=0; i<9; i++){
 5       Emit(i);
 6     }
 7   };
 8 };
 9
10 class Stage2: public wrapper <int, double> {
11 public:
12   void Process(int &i) override {
13     double square_root = sqrt(i);
14     Emit(square_root);
15   };
16 };
17
18 class Stage3: public wrapper <double, Nothing> {
19 public:
20   double total;
21   void Process(double &square_root) override {
22     total += square_root;
23   };
24 };
```

Listing 1: Example of DSParLib sequential wrappers.

Consider the parallel activity graph *pipe*($\square_0, \square_1, \square_2$), where $\square_n$ is a wrapped block of code. It represents a Pipeline with three sequential stages, as illustrated in Fig. 2. The `Stage2` wrapper in Listing 1 (line 10) can be in place of $\square_1$. However, it can not be in place of $\square_0$, nor $\square_2$. The reason is that this function expects an input and an output, which is not acceptable when it is the first or last stage of the Pipeline. DSParLib checks this type of semantics at compile time and reports inconsistencies to the programmers. To introduce the last stage, for instance, programmers must first adapt the sequential wrapper. The output type is then replaced by a special type `Nothing`. In Listing 1, we show a last stage example via `Stage3` (line 18). Note that the sequential wrapper types are `Wrapper<double, Nothing>` and it does not emit data items via `Emit()`.

3.4 Communicators and Data Communication

Frameworks and libraries for shared-memory architectures do not have to handle data communication because data can be accessed directly. In MPI or any other distributed parallel application, data must be copied between processes. This is a complex task, and different options are available for message passing programming models. For instance, in Java and . NET-based applications, data communication can be more transparent since these environments support runtime reflection (or introspection). This allows users of these frameworks to write distributed code that never handles message passing communication because frameworks will often handle it automatically.

DSParLib offers a layer of abstraction for simplifying this step without dealing with MPI low-level parameters (e.g., MPI types, tags, and communicators) or implementing its protocols to send and receive messages. In DSParLib, each data stream item is represented as a message. A message comprises one `header` and a data payload. The `header` contains the MPI rank of the process that sends the message, the MPI rank that receives the message, the type of a message, and a unique message identifier for each data item. When required, the identifier is used to automatically re-order the stream items in any sequential consumer wrapper. DSParLib employs a protocol that first sends a header message to establish communication between two processes (sender and receiver). Then, this channel can be used to receive multiple messages containing the actual data. The messages are non-blocking; after sending, the processes can resume other tasks.

From a programmers viewpoint, our solution provides two abstractions to simplify data communication: `MPISender` and `MPIReceiver`. We provide `Pack` and `Unpack` methods for implementing message passing to the sender and receiver, respectively. The programmers must call these methods in the correct order to serialize and deserialize the data. Data must be contiguously allocated in memory. This way, we provide zero-copy operations since data is sent *as it is* and thus does not involve the CPU. Additionally, DSParLib implements another programming abstraction on top of these features to simplify data communication, it is called `SenderReceiver`. The purpose of `SenderReceiver` is to provide an abstraction to serialize and deserialize data being sent/received through the network. When using native C++ data types, the `SenderReceiver` becomes a communicator object that automatically deals with data serialization. The following Section shows how to use this abstraction. When custom data types are sent through the network, the `SenderReceiver` abstraction can be overridden to implement the custom data type behavior.

Listing 2 showcases DSParLib 's `SenderReceiver` API when dealing with custom data types. Lines 1 to 5 describe a custom data type containing three fields. To implement data communication, the programmers implement a class extending `SenderReceiver` (line 6) and overrides its functions for `Send()` (line 7) and `Receive()` (line 12). Then, the programmers call `Pack()` for the sender and `Unpack()` for the receiver to each `struct` field in the same order while DSParLib handles the details and automatically creates and serializes the MPI communications using the aforementioned protocol. DSParLib 's templates can deal with up to 3-dimensional statically or dynamically allocated arrays or other contiguously allocated data.

```
struct CustomType{
 double value;
 size_t bytes;
 unsigned char * buffer;
};
class CustomAbstraction : public SenderReceiver <CustomType> {
 void Send(MPISender &sender,MessageHeader &msg,CustomType &
    data) override {
  sender.Pack(msg,data.value);
  sender.Pack(msg,data.bytes);
  sender.Pack(msg,data.buffer,size);
 }
 CustomType Receive(MPIReceiver &receiver,MessageHeader &msg)
    override {
  receiver.Unpack(msg,value);
  receiver.Unpack(msg,bytes);
  receiver.Unpack(msg,buffer,size);
  return CustomType(value,bytes,buffer);
 }
}
```

Listing 2: Example of Custom Type serialization.

If a sequential wrapper contains more than one `Emit()` in a single computational step (see Sect. 3.3), DSParLib will spawn messages using the same identifier. However, that constitutes an implementation error by the programmers, and the ordering may yield unpredictable results. A different aspect is that we implemented a special message informing the stream's end. When there is no more data, the producer informs all consumers to finish their execution and exit.

C++ compilers implement RTTI (Run-Time Type Information), which provides type information during runtime. We have investigated using this approach for DSParLib. Nonetheless, it does not work for automatic data communication purposes. RTTI only provides basic information about a type and does not support listing the fields of a `struct`. Furthermore, there is no standard approach to get the size of dynamically allocated data. The programmers need to intervene and manually specify the size of the allocations. Although we can not use more advanced techniques like RTTI, we implement a lightweight abstraction in DSParLib for simplifying data communication through message passing instead of using low-level MPI communicators.

3.5 Pipeline and Farm Patterns

In this section, we discuss the parallel patterns implemented and how they can be instantiated using C++11 features to improve programmability. It is possible to use C++ type inference and the `auto` keyword so that the programmers do not need to manually specify all the data types. Listing 3 shows how the programmers can use

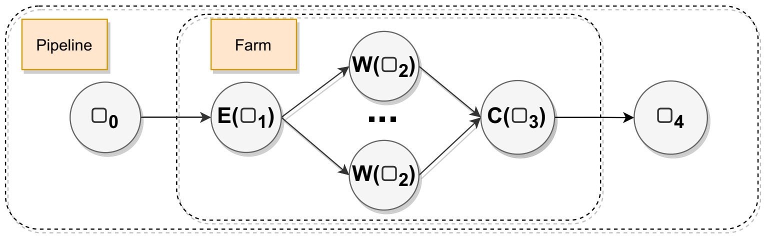


Fig. 3 Parallel activity graph example

DSParLib's building blocks to write parallel code. The example presents patterns composition using the following schema: $pipe\{\square_0, farm[E(\square_1), W(\square_2), C(\square_3)], \square_4\}$. The resulting parallel activity graph is illustrated in Fig. 3. In this example, the message is composed of the `double` data type. Since this is a contiguous memory type (data is placed in a single chunk of memory), the message passing communication is abstracted using `dspar::SenderReceiver<double>()` (line 2). Pipeline stages can be created as shown in lines 3 and 8, passing as parameters the sequential wrapper and the communicator mechanism. The Farm is created as shown in line 4, where the programmers inform the sequential wrappers and respective message passing mechanisms as parameters. Then, the final data stream can be modeled by building these blocks (sequential wrappers and patterns), as presented in line 9. The final parallel activity graph connects a Stage to a Farm pattern (with multiple blocks already inter-connected internally) and to the last stage. Finally, the method `Start()` (line 10) computes the complete parallel activity graph and schedules the MPI ranks. Each MPI rank will be responsible for a sequential wrapper.

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

Listing 3: Example of Pipeline and Farm composition.

The `dspar::Farm` pattern automatically infers the types and fails compilation if the programmers specify incompatible stages and communicators. The same is done for `dspar::Stage`. Regarding Farm's scheduling, the default option is round-robin. However DSParLib also implements on-demand scheduling, which can improve load balancing if the network is not a bottleneck, especially when Workers have a different computational load or when data stream items have unbalanced computational complexity. The messages are distributed on demand as soon as the Worker finishes the previous computation. Additionally, the `Farm` object supports the following customization options:

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

3.6 Runtime

This section explains how the runtime uses the wrapper classes to compute the stream activity graph. Also, we present the mechanisms that provide efficient scheduling protocols and ordering constraints in the runtime implementation. Figure 4 shows the relationship between the classes used in DSParLib. `DSParNode` inherits from `DSParLifecycle`. `FarmPattern` and `PipelineStage` create instances of `DSParNode` and run them based on their process rank. `AbstractPipelineElement` is used in the node allocation and will be explained in next Sect. 3.7. A `PipelinePattern` contains one or more instances of `AbstractPipelineElement` that can be of class type `PipelineStage` or `FarmPattern` (enables pattern composition).

Each DSParLib component internally implements an interface of type `DSParNode` that is executed by all MPI processes created during execution. `DSParNode` inherits from `DSParLifecycle`, which listens for messages coming from other processes and gathers information about important stream events such as the stream start, stream stops, and new messages. It executes actions according to the information captured. For example, capturing a stop signal message contains the information on whether the `DSParNode` must immediately finish execution or ignore the signal and wait for further instructions. This is useful for gatherers to track which Workers have stopped and only stop when the last Worker stops. Similarly, the start signal is part of the protocol to check whether a working node can immediately start receiving messages or ignore the signal and wait for further instructions.

Standard messages' payload do not require special treatment and are executed by the sequential wrapper implemented within the `Process()` method, as previously explained in Sect. 3.3. Nonetheless, if re-ordering is required, the messages' payload is not executed immediately. Instead, a precomputation determines if the payload should be processed or it should be stored for processing later. Since parallelism is non-deterministic in nature, some applications may require re-ordering so that the integrity of the output is guaranteed (e.g., order of frames in a video). The ordering mechanism is based on work described in [23] and uses a `std::priority_queue` for ordering the messages' payload in ascending order.

Concerning the scheduling protocols, DSParLib provides round-robin and on-demand scheduler protocols. The default scheduler is round-robin, which distributes

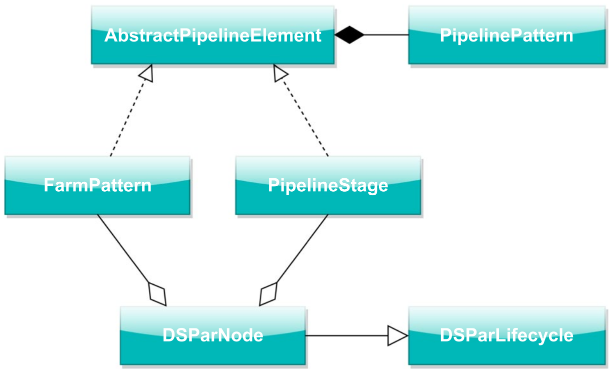


Fig. 4 Relationship of classes in DSParLib

balanced workloads among parallel processes using a circular order. We also implement on-demand scheduling. This way, data messages are only sent when the scheduler receives a demanding signal from a Worker. On the other side, Workers only demand new data when they finish their previous task. This protocol uses more messages in the network but can improve workload balancing. For example, if a process is faster than another, it demands more data to compute.

3.7 Planning Node Allocation

Before starting the stream processing execution, DSParLib needs to know which node must be executed by a given MPI rank. This section describes the process of allocating DSParLib nodes and MPI ranks.

First, we explain the Farm parallel pattern. By default, the Emitter and Collector are placed as neighbors (Emitter on rank 0, Collector on rank 1), and parallel workers go from 2 to $2 + \mathit{degree\_of\_parallelism} - 1$. If the default MPI process allocation is used, rank 0 and 1 will be placed in the same cluster node equipped with a multi-core processor. Considering the Emitter and Collector are network or disk I/O intensive, both processes may have degraded performance since they would compete for limited resources. However, the user can change it by providing their custom `hostfiles` with different allocation configurations.

When dealing with a Pipeline or Pipeline with Farm, DSParLib uses the following strategy. Each instance in DSParLib requires its nodes to implement the `AbstractPipelineElement` class from previous Fig. 4. This class stores information about the stage's position and how many parallel processes exist when the pipeline node is a Farm. We define the information used to provide processes' ranks as input and output offsets and the total number of processes. This information is later used to provide the correct rank for each process. By default, Pipeline stages will have their rank matching their position in the Pipeline. When combining a Pipeline with Farm, each Pipeline stage will be dislocated according to the Farm topology, depending on its degree of parallelism.

For instance, let us define the following parallel activity graph as example: $pipe\{\square_0, \square_1, farm[E(\square_2), W(\square_3), C(\square_4)], \square_5\}$. The activity graph example is illustrated in Fig. 5. In practice, Each Pipeline stage implements an `AbstractPipelineElement` where some of them are of type `FarmPattern` while others are of type `PipelineStage`. By default the `FarmPattern` communicates its input and output offsets as 0 and 1, and total number of processes as $2 + degree\_of\_parallelism$. Since there are other building blocks (stages $\square_0$ and $\square_1$) before the `FarmPattern`, these offsets are used to calculate the actual ranks where the Farm will be positioned. For example, if two `PipelineStage` instances precede the `FarmPattern`, the input offset 0 is summed to 2. Consequently, $\square_0$ and $\square_1$ are positioned at rank 0 and 1, while the `FarmPattern` has its Emitter on rank 2, Collector is rank 3, and parallel workers from ranks 3 to $3 + degree\_of\_parallelism - 1$. Finally, $\square_5$ is positioned at the end of the Pipeline. This means that DSParLib assigns the MPI ranks based on the final parallel activity graph.

4 Stream Processing Applications

In this section, we briefly describe the applications and their parallelizations using DSParLib. First, in Sect. 4.1 we discuss the parallelization methodology of five different stream processing applications: two of them are synthetic applications, and the remaining three are real-world streams processing applications. Then, in Sect. 4.2 we parallelize the PARSEC's Ferret benchmark with different parallel pattern compositions.

4.1 Parallelized Application with the Farm Pattern

We select five applications from the stream processing domain: two synthetic applications for stressing the schedulers with highly unbalanced workloads (Mandelbrot and Prime numbers); and three applications that represent real-world scenarios (Face Recognition, Lane Detection, and Bzip2). We briefly describe them:

1. **Mandelbrot** is a mathematical application that computes a fractal in the complex plane [17]. This application is naturally unbalanced. By default, it performs almost no I/O operations, but it can easily be modified to do so to mimic a typical stream processing application. We have implemented equivalent parallel versions for both programming abstractions (DSParLib and MPI). The parallel version implements a Farm pattern with re-ordering disabled. In the MPI version, the message passing step in this application uses default MPI data types. The data to be communicated is a 2D matrix which is sent in the shape of lines. In short, each Worker starts receiving a `line_id` (MPI_INT) for computation and finishes sending the resulting line (MPI_BYTE).
2. **Prime numbers** counts how many prime numbers are within a given range [18]. The algorithm checks if: for a given number n, there is any number between 2

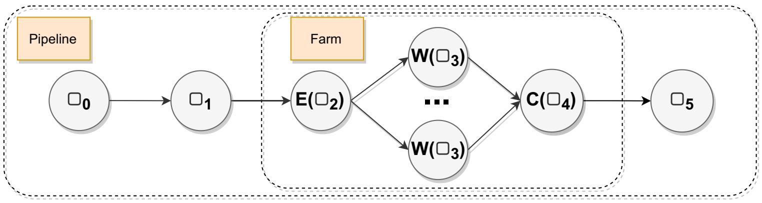


Fig. 5 Parallel activity graph example

and $n - 1$ that n is divisible. This synthetic application is highly unbalanced by default. We obtained a pre-existing handwritten MPI version [20]. We manually inspected it to guarantee that the code is similar to ours. The DSPARLIB and MPI versions were implemented using a Farm-like pattern with re-ordering disabled. The message passing in this application is straightforward because it only communicates integers.

3. **Face recognition** is a program that recognizes faces in a video stream using OpenCV [21]. A lightweight model is trained before execution, and later used to detect faces. To our knowledge, this code has no pre-existing handwritten MPI versions. Therefore, we developed a Farm-like parallel version with re-ordering enabled for both MPI and DSPARLIB versions. Implementing message passing in this application is challenging because data is communicated using an OpenCV data type. So we investigated the source code and inspected the OpenCV to find the primitive data types of the data object. Finally, we manually send and receive each one of the members separately using the appropriate data types, and then we reconstruct the OpenCV data type.
4. **Lane detection** is a program that detects the limits of road lanes for autonomous vehicles based on a video stream using OpenCV [21]. We use a pre-existing handwritten MPI application [35]. The MPI version employs a Farm pattern, and we manually implemented an ordering strategy. On the other hand, the DSPARLIB version uses the Farm pattern and our library's built-in ordering. Similar to Face Recognition, this application communicates data using the same OpenCV object (`cv:Mat`). So we implemented the same strategy for the message passing step. We have checked the strategy used in the MPI version (implemented by others), which is similar to ours.
5. **Bzip2** is a compression library used in some Linux distributions. The MPIBzip2 is a parallel version using OpenMPI. The parallelism strategy of MPIBzip2 is like a master-worker [20] pattern, where the master splits the file into smaller blocks and collects them while the workers perform the compression. In DSPARLIB version, the parallel implementation uses a Farm pattern with re-ordered enabled. The main difference between MPI and ours is that the master task is split into Emitter and Collector. Concerning the message passing, since we are dealing with the compression phase, the blocks read from the original file have a default size of 900Kb. Therefore, the communication starts by sending and receiving messages with a known amount of bytes. However, considering compression rate varies,

the new size must be known for passing messages after compressing a block of data.

For each application, we additionally implemented different versions using two scheduling policy strategies: on-demand and round-robin. The former expects the Workers to request new tasks once they are free, and the latter uses a round-robin distribution fashion that distributes balanced workloads among the Workers in a circular order. The on-demand scheduling can improve performance when dealing with unbalanced workloads, as will be shown in the experimental section. Listing 4 shows the optimization options available in DSParLib. Instead of implementing the scheduler or the re-ordering strategies from scratch, programmers can inform DSParLib which features they want to enable.

```
auto c = dspar::SenderReceiver<...>();
auto farm = dspar::Farm(Emitter, c, Worker, c, Collector);
farm.SetCollectorIsOrdered(true/false);
farm.SetOnDemandScheduling(true/false);
farm.SetWorkerReplicas(degree_of_parallelism);
farm.Start();
```

Listing 4: DSParLib Farm template used in all applications.

Using DSParLib required significantly less effort than writing a distributed parallel code with MPI. Most of the development work regards the code refactoring from the application into the Farm pattern and implementing the data communication (marshaling and unmarshalling data). Because of that, during the conception of DSParLib we focused on data message passing abstractions while balancing the trade-off between programmability and flexibility aspects for writing stream parallel codes.

4.2 Ferret with Different Parallel Patterns

The Ferret application belongs to the PARSEC benchmark suite and is used for detecting similarities between video, audio, and image files [6, 22]. The Ferret application contains an original parallelization targeting multi-cores using the Pthreads library. In this version, the authors have parallelized Ferret using a Pipeline parallel pattern. There are six pipeline stages, and two of them are responsible for loading and collecting the data. In comparison, the other four stages are responsible for computational processing: Segmentation, Extraction, Vectorization, and Ranking.

We have based our parallelization on the original Pthreads version. Therefore, the first distributed version we implemented with DSParLib is a pipeline containing four computational stages, as shown in the bottom part of Fig. 6. Note that the four computational stages are stateless, meaning they do not maintain previous states and do not have data dependencies. Therefore, we can replicate the stages to increase the degree of parallelism up to the maximum degree available on a target machine. This version was the most difficult one to implement because message passing in Ferret is complex. The complete `struct` used for communicating data has more than 20 members, varying from integers, pointers using 1 or 2 dimensions, and custom data types such as Ferret's CASS types (Content-Aware Search System). Other complexities rely on non-contiguous data and nested data structures.

To conceive this parallel activity graph in Ferret, our parallel implementation uses the composition of Pipeline and Farm parallel patterns. We created Farms to express the parallel Worker, as represented in Listing 5. Note that the Farms have a different number of parameters. Since the parallelization strategy employs a Pipeline of Farms, all Farms are implemented accordingly to their graph position. For example, the Farm 0 communicates with Farm 1. Therefore, the Farm 0 adds an extra communicator at the end, while Farm 1 adds an extra communicator at the beginning. Also, all Farms implement at least one Emitter or Collector stage using an empty stage. We implemented our solution with this strategy to maintain the functional correctness of the data flow. The Farm parallel pattern, as mentioned in previous Sect. 3.2, must implement a scheduler and a gatherer, namely Emitter and Collector. For example, a replicated stage cannot directly communicate with another replicated stage without a proper scheduling protocol. We plan to implement optimized strategies in the future, like the all-to-all communication model. This means that each Worker from the previous stage has multiple messages passing queues to each Worker from the subsequent stage. For now, our strategy is compliant with the Farm parallel pattern, having a clear notion of Emitter, Worker, and Collector. Sometimes applications have natural Emitter and Collector stages, like the `Load` and `Collect` computations of Ferret. On the other hand, when no sequential application code fits in (Ferret internal stages), we use empty stages with no computational processing. These are *empty stages* because they do not process anything. Rather, they simply forward the messages received from the previous stage.

```
auto comm = dspar::SenderReceiver<... >();
EmptyStage <task>  E, C;

auto farm_0 = dspar::Farm(Load, comm, Seg, comm, C, comm);
auto farm_1 = dspar::Farm(comm, E, comm, Extract, comm, C, comm);
auto farm_2 = dspar::Farm(comm, E, comm, Vect, comm, C, comm);
auto farm_3 = dspar::Farm(comm, E, comm, Rank, comm, Collect);

Pipeline pipe;
pipe.Add(&farm_0);
pipe.Add(&farm_1);
pipe.Add(&farm_2);
pipe.Add(&farm_3);
```

Listing 5: Example of Pipeline and Farm composition.

Alternatively, we have implemented another similar parallel and distributed version with DSParLib. The most difficult part of the message passing is communicating the `Vect` stage results to the next `rank` stage. To get the correct sizes

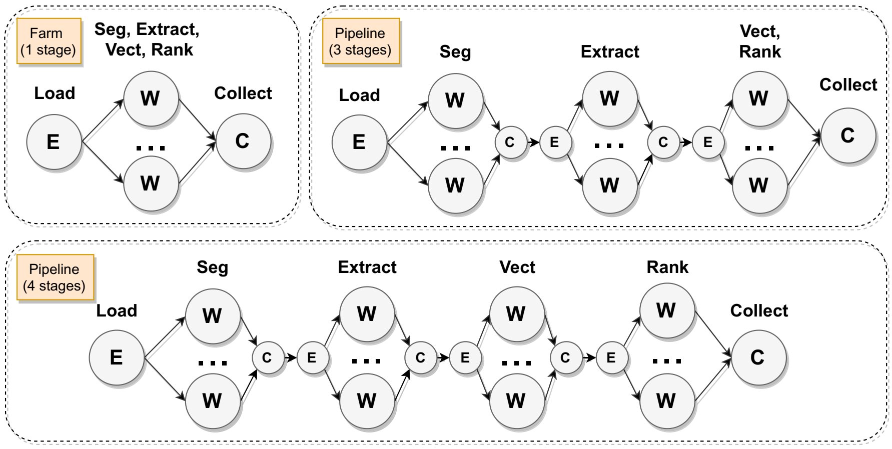


Fig. 6 The parallel activity graph of the Ferret application's parallel versions

of the amount of memory allocated for each data struct, we must look deep into Ferret's source files. Therefore, in our second version, we combine the `Vect` and `rank` stages into a single one. The resulting parallel activity graph can be seen on the top right-hand side of Fig. 6.

Finally, we developed one last version that implements the Farm parallel pattern without composition, as illustrated in the top left-hand side of Fig. 6. This version contains a single computational stage (Worker) obtained by merging the Segmentation, Extraction, Vectorization, and Ranking stages into a single one. The message passing in this version is significantly simpler than other DSPARLIB versions. The reason is that we only communicate the data items from the Emitter to the Worker and later from Worker to Collector. Intermediate data are not sent over the network because the computation stays in the same node and is performed locally using shared memory. As discussed by the authors from [22], Ferret's stages are not well balanced, showing some drawbacks regarding parallelism. If the computing stages are replicated using the same factor, the unbalancing problem remains, and resource usage is not optimized, resulting in performance losses. This will be shown in Sect. 5.2 when we discuss Ferret's results.

5 Experiments

In this section, we experimentally evaluate the performance and programmability aspects of DSPARLIB concerning MPI, which is the *de-facto* parallelism abstraction for HPC. Section 5.1 aims to evaluate DSPARLIB performance with Farm-like parallel versions and compare the results to MPI handwritten versions. In Sect. 5.2, we extended our performance evaluation using the PARSEC Ferret benchmark as a use case to embody characteristics of modern stream processing applications. Therefore, we evaluate the strategies designed to parallelize Ferret's complex activity graph.

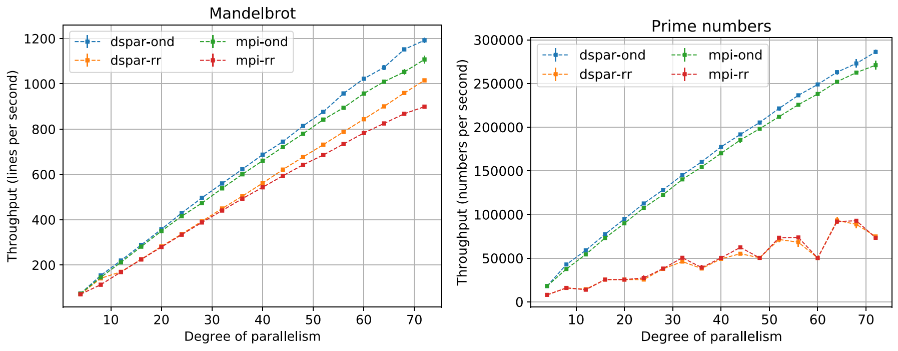
The experiments were executed on a cluster using eight computing nodes. Each node was equipped with 2 Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz (totaling 12 cores and 24 threads) with 24GB of RAM memory. The nodes were connected via Gigabit Ethernet and InfiniBand QDR 4x (32GBit/s). The operating system was Ubuntu 16.04 64 bits with kernel 4.4.0-146-generic. The MPI version was OpenMPI 1.4.5. The applications were compiled with GCC 9.3.0 using -O3 optimizations. OpenCV version was version 2.4.13.

The throughput is measured using *items/time*, where *items* is the total number of stream items transmitted and *time* is the total execution time. Bzip2 is an exception; it is measured in *MB/time* using the original file's size and execution time. Regarding the plotted graphs, the x-axis is the degree of parallelism, while y-axis represents the performance metric (*items/time* or *size/time*). The results plotted were obtained from the arithmetic mean of 30 executions performed for each configuration. The standard deviation is plotted using error bars, which may not be visible when the value is negligible. 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.

5.1 Performance Evaluation with Farm-Like Applications

In this section, our goal is to assess the performance of DSParLib concerning handwritten MPI codes. We investigated the experimental results and observed that DSParLib 's programming model could be slightly more efficient than standard MPI implementations. Although the parallelism strategy is equivalent between the versions, we cannot ensure they are identical because of particular communication protocols and contrasting programming models. The significant difference between the versions is that DSParLib uses dynamic process allocation from MPI-2, while handwritten MPI codes generate static processes via MPI-1. In DSParLib, the program is initiated with a single MPI process that will later create and allocate the other processes via MPI-2 functions like `MPI_Comm_spawn` and `MPI_Intercomm_merge`. Other differences are regarding the number of messages sent or received by each version, which is not always the same, especially for the parallel versions we obtained from the literature. Besides that, performance variations between versions may result from low-level MPI optimizations. Regarding total number of processes, Figs. 7 and 8 simply add 2 processes (one for Emitter and one for Collector) to the degree of parallelism. Furthermore, we individually explain each application in the remainder of this section.

The handwritten MPI programs we implemented do not use the MPI-2 programming model during the parallelization. Other works [20, 35] do not implement their applications using MPI-2 to support dynamic process management, which highlights the challenges inherited from this programming model. In short, some key differences exist regarding implementing MPI programs using different programming models. In MPI-1, all processes are created at the beginning and in a single communication group (*intracommunication*) as `MPI_COMM_WORLD`. In MPI-2, a single parent process can dynamically initialize children processes



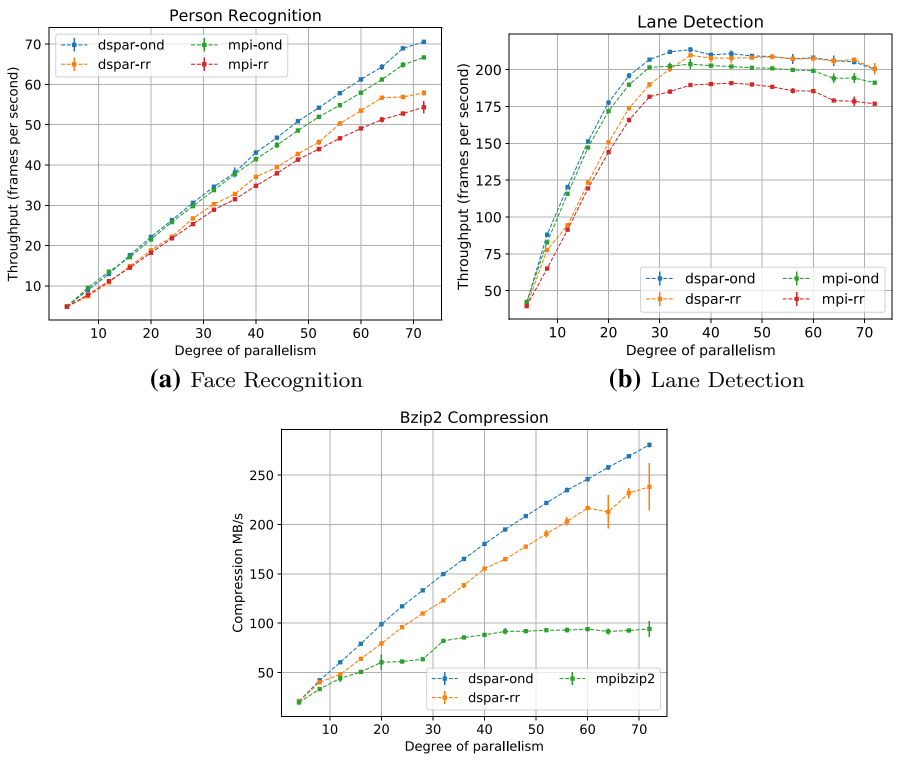
(a) Mandelbrot **(b)** Prime numbers

Fig. 7 Performance evaluation of Mandelbrot and Prime numbers

during execution time. However, they form a new group of processes since they create their MPI_COMM_WORLD. Consequently, the MPI program now must use *intercommunicators* to bind a communication context between these two groups of processes recognized as local and remote. Adding or removing processes becomes very difficult during the application execution because other features must be considered. Some examples are reasoning about where processes are physically allocated and executed, rearranging multiple communicators, and the relationship of each process's role in the parallel activity graph.

The tests were executed by varying the degree of parallelism using multiples of 4, starting from 4 up to the maximum number of available resources. For the first five Farm-like applications, we set the maximum number of processes on each node to avoid hyper-threading. Each node runs 12 processes on 12 physical cores. Also, our experimental analysis considered different process allocation strategies (e.g., isolate Emitter and Collector, allocate all available cores first, etc.). For brevity, we only report the allocation strategy with the highest overall performance in all versions (MPI and DSparLib). This strategy isolates the Emitter and Collector: Each runs in a separate node (nodes #0 and #1). Then, the other processing nodes are allocated to available nodes in a round-robin fashion. For example, Worker 2 is placed in node #2, Worker 3 is placed in node #3, and so on. We have verified the functional correctness of our parallel versions using checksum in the output and MD5 hash comparison.

Mandelbrot computes a 6000 × 6000 pixel image of the Mandelbrot set, using 10,000 iterations. For all versions, computation is performed line-by-line, similar to the shared-memory parallelism strategy proposed in [17]. The performance between MPI and DSPARLIB is similar as shown in Fig. 7a. However, at its peak, DSPARLIB has up to 7.6% higher throughput than MPI. In this test, lines near the middle of the Mandelbrot image are more computationally intensive 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. When we run the DSPARLIB program with dynamic process allocation, DSPARLIB creates a new communication group containing



(a) Face Recognition **(b)** Lane Detection

(c) Bzip2

Fig. 8 Performance of face recognition, lane detection, and Bzip2

all the spawned processes. However, for the MPI versions, all processes live in `MPI_COMM_WORLD`.

Prime numbers tests the prime property of all numbers between 0 and 1.2 million. In this benchmark application, the performance difference between scheduler types is bigger than in Mandelbrot. This is shown in Fig. 7b. The main problem is due to an unbalanced load. Using the on-demand scheduler helps balance the workload as Workers ask for more items on demand. Another detail that can be observed in Fig. 7b is the unstable behavior of the round-robin scheduling versions. The reason is that the degree of parallelism directly interferes with the scheduling. For example, a degree of parallelism 2 implies that one Worker always receives even numbers while the other receives odd numbers. Checking if an even number is a prime is simpler than an odd one. In Prime numbers, the difference between the abstractions is smaller. DSParLib has up to 5.5% higher throughput than MPI in the peak performance.

Face Recognition processes a 15-second video input. In this application, it is worth mentioning that all frames are computationally expensive. Therefore, Face Recognition never reached the disk I/O bottleneck in our tests. We could not find such an application implemented with MPI in the literature, so we implemented it ourselves. Figure 8a shows the throughput in frames per second (FPS), where the application continues scaling until the maximum degree of parallelism. At its peak performance, DSParLib has 5.8% higher throughput than MPI. The MPI on-demand and DSParLib on-demand versions have similar performance, but DSParLib is slightly faster. DSParLib and MPI are sending and receiving similar messages, and the MPI version have an advantage since the header messages are smaller. This is similar to lane detection, where header messages are smaller; however, DSParLib is still faster.

Lane Detection computes over a video input containing 1858 frames of size 640x480 pixels. Figure 8b shows the throughput in frames per second (FPS). The peak throughput is 214.55 frames per second with 32 Workers (DSParLib on-demand). After that, it reaches the bottleneck for disk I/O, and the program stops scaling. The on-demand MPI version has similar performance until 28 Workers. The difference becomes more evident as the number of processes increases. Both versions 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 slightly consumes more network bandwidth, results show that our version achieves better performance with respect to handwritten MPI.

Bzip2 has an existing parallel MPI version called Mpibzip2. It uses a master-worker approach instead of a native stream processing pattern. Since the DSParLib version isolates the Emitter and Collector, we also isolated the Mpibzip2 master in a separate cluster node. We tested with a 512MB MP4 video file. Figure 8c shows the performance of all versions. Mpibzip2 achieves lower performance compared to DSParLib. The maximum throughput of DSParLib is 2.98 times higher than the handwritten MPI program. The main reason is networking and synchronization bottlenecks because the master process is responsible for both I/O operations: reading and writing data. At 44 processes, the MpiBzip2 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.

5.2 Performance Evaluation of Ferret with DSParLib

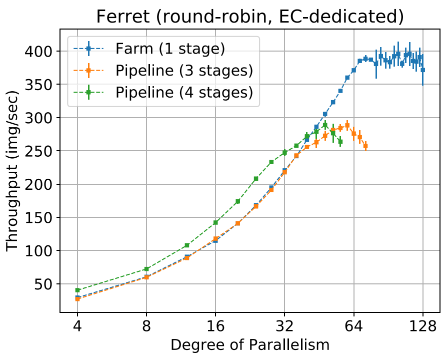
In this section, our goal is to assess the performance of DSParLib using the semi-arbitrary pattern composition of Pipeline and Farm. Ferret application exhibits complex data communication through message passing and data flows. We implemented three different distributed and parallel versions according to the parallel activity graphs illustrated in Fig. 6. Unlike the previous experimental setup, to evaluate Ferret we used our system's maximum available computing resources. Effectively, all cores, including logical ones, are used. Additionally, we configured different MPI process allocation strategies to study their impact on dynamic process allocation. We conducted tests using the round-robin and fill-node allocation strategies combined with different Emitter and Collector configurations:

- `round-robin, EC-dedicated`: This configuration uses a strategy that allocates one process in each available node in a round-robin fashion. Both Emitter and Collector have dedicated nodes.
- `fill-node, EC-dedicated`: This configuration uses a strategy that allocates processes in the same node and only goes to the next node when it is fully allocated. Both Emitter and Collector have dedicated nodes.
- `fill-node, EC-shared`: This configuration uses a strategy that allocates processes in the same node and only goes to the next node when it is fully allocated. Both Emitter and Collector share the same node.

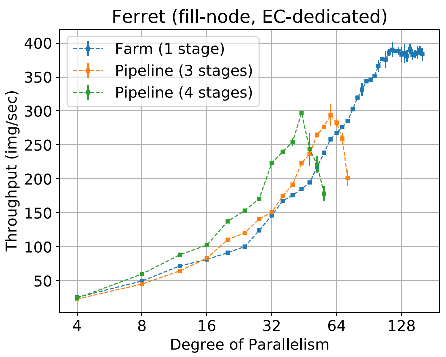
We tested the PARSEC Ferret benchmark with the default native workload composed of 3.500 images. The performance results are shown in Fig. 9. We depicted all results until they peaked and began decreasing performance. There are two different explanations: (1) the workload size limits further scaling when reaching higher degrees of parallelism, which explains the limited scalability observed in Farm implementations; (2) both pattern composition versions (Pipeline) stop scaling long before that since they have reached a point where they are using all available cores. For example, at the degree of parallelism 2, there are actually 12 $(6 + 2 * 3)$ processes running for the Pipeline with 3 stages and 16 $(8 + 2 * 4)$ processes for the Pipeline with 4 stages.

To help understand the results from the graphs, we prepared Table 2 showing the best throughput measured in each version and configuration. As expected, the Farm version scales better than the other versions and only stops scaling when the workload size becomes a problem. In the best-case scenario, the Farm version has up to 37% higher throughput than the other versions. For both the Pipeline versions, we observed that the maximum throughput is equivalent between themselves and varies less than 1%. Considering that the Pipeline (3 stages) version communicates less data because we have merged the last two stages avoiding a significant amount of communication, we expected this version to be slightly faster. Nonetheless, these positive results highlight that the overhead introduced by our data message passing abstraction is negligible.

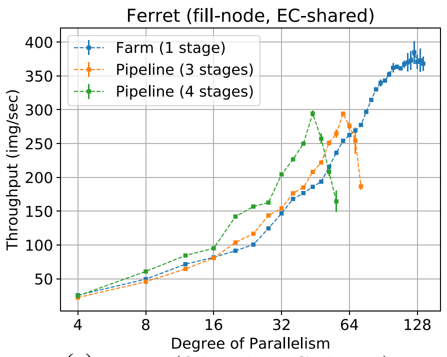
Concerning the different process allocation strategies, both Table 2 and graphs from Fig. 9 show us that the maximum throughput is similar among all versions. But if we look at the graphs from Fig. 9, we observe that the `round-robin, EC-dedicated` configuration is smoother than the others. Both experiments using the `fill-node` allocation strategy show serious performance losses after reaching the peak throughput. Therefore, we concluded that it is better to use the `round-robin, EC-dedicated` configuration as default in DSParLib while allowing the programmers to use custom configurations.



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



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



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

Fig. 9 Performance of Ferret using DSPARLIB and different process allocation strategies

5.3 Programmability

In this section, we discuss programmability aspects of parallel and distributed implementations for DSPARLIB with respect to handwritten and manually optimized MPI programs. We measured SLOC (Source Lines of Code) for each of the five applications described in Sect. 4.1. We do not consider Ferret for this comparison because we do not have a handwritten pure MPI parallelization. The SLOC metric does not consider blank lines and comments, only valid programming language syntax. With SLOC, it is possible to infer a general idea about the level of code intrusion that each programming abstraction requires for introducing parallelism. The goal is to measure and compare the extra number of code lines needed for parallelizing a sequential application. It is worth noting that this metric has its limitations. For example, SLOC is not representative enough to express which parallel programming abstraction has the best programmability characteristics. Other aspects need to be taken into account. For example, a reasonable approach could measure the total development time, programmer experience, application context, etc.

Table 2 Best throughputs measured in Ferret application

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

Table 3 compares the measured SLOC for DSParLib and handwritten MPI versions. The Sequential entry represents the sequential application version. MPI has two entries since each version requires different implementation and synchronization details. DSParLib represents both versions, where On-Demand and Round-Robin are the same, requiring only an extra parameter that is either enabled or disabled.

Mandelbrot is the only application where DSParLib requires more lines of code. The explanation is that DSParLib wraps each different building block in a class, resulting in more code. Yet, comparing DSParLib with MPI On-Demand shows that the difference is as small as 13.3%. Prime numbers indicate a higher difference between our abstraction and MPI, where DSParLib is up to 6x less intrusive than MPI. For MPI versions, the SLOC is higher because it requires low-level implementation details that make the source code more verbose. Face recognition also showcases a higher difference, where MPI SLOC is 91.1% higher than DSParLib. In Lane Detection and PBzip2, the handwritten MPI programs are 25.3% and 37.4% respectively more intrusive than DSParLib.

These results highlight one of the significant advantages of DSParLib. Our abstraction can simplify many low-level parallel details intrinsic to MPI programming. During the experiments, we had to fix a series of bugs in handwritten MPI C++ programs implemented by other experts, mainly due to memory and logic errors when dealing with message passing. DSParLib introduces a high-level API that makes it simpler to express parallelism while hiding message-passing communication details. Moreover, DSParLib implements a strong type-check system that looks for incorrect usage and wrong semantics at compilation time.

To give the reader an idea of the challenges inherited from stream processing applications parallelized via MPI, we extracted a code region from the Person Recognition application. Note that this slice of code (Listing 6) only shows the worker scope, while the operator's scheduling and sink strategies are hidden. Their strategy can become intricate when implementing efficient schedulers or ordering constraints. The worker logic and MPI functions are described between lines 5 and 28.

Table 3 Comparison of applications Source Lines of Code (SLOC)

Versions	Mandelbrot	Prime numbers	Face Recognition	Lane Detection	PBzip2
Sequential	42	35	126	114	1388
MPI on-demand	147	329	293	302	1880
MPI round-robin	97	324	296	297	–
DSParLib	161	83	215	264	1746

Lines 8 and 9 are part of the on-demand scheduling. Once the Emitter acknowledges the *demand* signal, lines 11 to 17 perform the MPI functions to receive data. Line 18 abstracts the person recognition computation. Then, lines between 20 and 26 send the result to the Collector.

In MPI, the programmers are responsible for manually implementing the data communication. They must understand MPI specifications and deal with low-level parallelism and architecture aspects, such as manually implementing the scheduler while providing the correct information about various data types, MPI message options, and data sizes without crashing the program. It is worth noting that MPI's parallelism strategy is mixed-up with the application business logic. It is challenging to debug and maintain code, especially in large applications.

```
int main(int argc, char **argv){
 if (isEmitter){
  /* abstracted scheduling logic */
 }
 else if (isWorker){
  while (true){
   // request work
   MPI_Send(&demand,1,MPI_INT, emitterRank,DEMAND_MSG, comm);
   MPI_Probe(emitterRank,MPI_ANY_TAG, comm,&status);
   // Receive Data
   MPI_Recv(&msgId,1,MPI_INT, emitterRank,ID_MSG,comm,&status);
   MPI_Recv(&nframe,1,MPI_INT, rank,DATA_MSG, comm,&status);
   MPI_Recv(&rows,1,MPI_INT, rank,DATA_MSG, comm,&status);
   MPI_Recv(&cols,1,MPI_INT, rank,DATA_MSG, comm,&status);
   MPI_Recv(&type,1,MPI_INT, rank,DATA_MSG, comm,&status);
   MPI_Recv(&size,1,MPI_INT, rank,DATA_MSG, comm,&status);
   MPI_Recv(data, size,MPI_BYTE, rank,DATA_MSG, comm,&status);
   /* abstracted computation */
   // Send result
   MPI_Send(&messageId,1,MPI_INT, collectorRank,ID_MSG,comm);
   MPI_Send(&data.nframe,1,MPI_INT, rank,DATA_MSG, comm);
   MPI_Send(&mat.rows,1,MPI_INT, rank,DATA_MSG, comm);
   MPI_Send(&mat.cols,1,MPI_INT, rank,DATA_MSG, comm);
   MPI_Send(&type,1,MPI_INT, rank,DATA_MSG, comm);
   MPI_Send(&size,1,MPI_INT, rank,DATA_MSG, comm);
   MPI_Send(mat.data, size,MPI_BYTE, rank,DATA_MSG, comm);
  }
 }
 else if (isCollector){
  /* abstracted sink logic */
 }
 MPI_Finalize();
}
```

Listing 6: Slice of Person Recognition's parallel code implemented with MPI.

On the other hand, employing a structured parallel programming approach via parallel patterns improves programmability. For comparison, the parallelization on the same stream processing application (Person Recognition) is similar to Listing 4. Instead of manually implementing the message passing strategy, DSParLib provides programming abstractions that hide communication complexities, and the programmer's concern move towards *how to use and interconnect* DSParLib building blocks or parallel patterns. Likewise, on-demand scheduling and ordering in DSParLib are enabled with a single function call.

6 Conclusion

In this paper, we introduced DSParLib, a parallel programming abstraction for expressing distributed stream parallelism in C++ programs. It is based on established parallel pattern concepts and building blocks developing approach. DSParLib offers ready-to-use abstractions to model and parallelize stream processing applications. By default, it supports round-robin and on-demand schedulers and supports built-in item re-ordering. The programmers can use these features simply by informing DSParLib which ones they wish to enable. Also, we implemented abstractions for message passing communication and synchronization.

To assess the performance of DSParLib, we executed a set of experiments over different applications. For comparison, we have extracted MPI handwritten programs from other authors or implemented by experts. The results revealed that DSParLib achieves better performance with respect to MPI versions. We investigated the effects and concluded that the improved performance is due to the better parallelism strategies implemented in DSParLib. In some cases, we achieve a throughput at most 6.3% higher than handwritten MPI. Regarding programmability, DSParLib generally requires the lowest amount of extra lines of code for expressing stream parallelism. The results show that DSParLib requires up to 6x less extra lines of code than handwritten MPI programs.

Additionally, the experiments performed on Ferret show that DSParLib can be used to efficiently implement complex stream processing applications with semi-arbitrary pattern composition. In future work, we plan to use DSParLib as a runtime library for SPar [19], which is a higher-level abstraction based on the C++-11 code annotation mechanism. It has a source-to-source compiler that may generate parallel code with calls to DSParLib. Finally, we intend to support more parallelism optimizations, self-adaptive strategies, and fault-tolerance mechanisms.

Acknowledgements We would like to acknowledge the support of LAD-PUCRS, GMAP research group, and PUCRS university. This research is partially funded by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, FAPERGS 05/2019-PQG project ParAS (N^o 19/2551-0001895-9), FAPERGS 10/2020-ARD project SPar4.0 (N^o 21/2551-0000725-7), Universal MCTIC/CNPq call 28/2018 project SParCloud (N^o 437693/2018-0), and MCTIC/CNPq call 25/2020 (N^o 130484/2021-0)

References

1. Aldinucci, M., Campa, S., Danelutto, M., Kilpatrick, P., Torquati, M.: Targeting Distributed Systems in Fastflow. In: Proceedings of the 18th International Conference on Parallel Processing Workshops, Euro-Par'12, pp. 47–56. Springer, Berlin (2013)
2. Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: Fastflow: High-Level and Efficient Streaming on Multicore, chapter 13, pp. 261–280. Wiley (2017)
3. Anderson, M., Smith, S., Sundaram, N., Capotă, M., Zhao, Z., Dulloor, S., Satish, N., Willke, T.L.: Bridging the gap between HPC and big data frameworks. Proc. VLDB Endow. **10**(8), 901–912 (2017)
4. Andrade, H.C.M., Gedik, B., Turaga, D.S.: Fundamentals of Stream Processing: Application Design, Systems, and Analytics, 1st edn. Cambridge University Press, New York (2014)
5. Apache Storm.: Apache Storm. https://storm.apache.org, July 2019. Accessed 16 May 2021
6. Bienia, C., Kumar, S., Singh, J. P., Li, K.: The PARSEC benchmark suite: characterization and architectural implications. In: 17th International Conference on Parallel Architectures and Compilation Techniques, PACT '08, pp 72–81. ACM, Toronto (2008)
7. 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: 2016 IEEE International Conference on Big Data (Big Data), pp. 172–183 (2016)
8. Boost committee: Boost C++ library: Serialization. https://www.boost.org/doc/libs/1_79_0/libs/serialization/doc/index.html (2004)

9. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache Flink^{TM}: stream and batch processing in a single engine. IEEE Data Eng. Bull. **38**, 28–38 (2015)
10. Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. University of Glasgow, Glasgow (1989)
11. Cole, M.: Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. Paral. Comput. **30**(3), 389–406 (2004)
12. del Rio Astorga, D., Dolz, M.F., Fernández, J., García, J.D.: A generic parallel pattern interface for stream and data processing. Concurr. Comput.: Pract. Exp. **29**(24), 1–12 (2017)
13. Ernsting, S., Kuchen, H.: A scalable farm skeleton for hybrid parallel and distributed programming. Int. J. Parallel Program. **42**, 968–987 (2013)
14. Falcou, J., Sérot, J., Chateau, T., Lapresté, J.: Quaff: efficient C++ design for parallel skeletons. Parallel Comput. **32**(7), 604–615 (2006). (Algorithmic Skeletons)
15. González-Vélez, H., Leyton, M.: A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. Softw.: Pract. Exp. **40**(12), 1135–1160 (2010)
16. Grant, W. S., Voorhies, R.: Cereal—a C++11 library for serialization. https://github.com/USCiLab/cereal, 2017
17. Griebler, D.: Domain-specific language & support tool for high-level stream parallelism. PhD thesis, Faculdade de Informática—PPGCC - PUCRS, Porto Alegre, Brazil (2016)
18. Griebler, D., Danelutto, M., Torquati, M., Fernandes, L. G.: An Embedded C++ domain-specific language for stream parallelism. In: Parallel Computing: On the Road to Exascale, Proceedings of the International Conference on Parallel Computing, ParCo'15, pp. 317–326. IOS Press, Edinburgh (2015)
19. Griebler, D., Danelutto, M., Torquati, M., Fernandes, L.G.: SPar: a DSL for high-level and productive stream parallelism. Parallel Process. Lett. **27**(01), 1740005 (2017)
20. Griebler, D., Fernandes, L. G.: Towards Distributed Parallel Programming Support for the SPar DSL. In: Proceedings of the International Conference on Parallel Computing, ParCo'17, pp. 563–572. IOS Press, Bologna (2017)
21. Griebler, D., Hoffmann, R. B., Danelutto, M., Fernandes, L. G.: Higher-level parallelism abstractions for video applications with SPar. In: Proceedings of the International Conference on Parallel Computing, ParCo'17, pp. 698–707. IOS Press, Bologna (2017)
22. Griebler, D., Hoffmann, R.B., Danelutto, M., Fernandes, L.G.: High-Level and Productive Stream Parallelism for Dedup, Ferret, and Bzip2. Int. J. Parallel Program. **47**(1), 253–271 (2018)
23. Griebler, D., Hoffmann, R.B., Danelutto, M., Fernandes, L.G.: Stream Parallelism with Ordered Data Constraints on Multi-Core Systems. J. Supercomput. **75**(8), 4042–4061 (2018)
24. 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 Comput. **84**, 24–36 (2019)
25. Mancini, E.P., Marsh, G., Panda, D.K. An MPI-stream hybrid programming model for computational clusters. In 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, pp. 323–330 (2010)
26. Mattson, T., Sanders, B., Massingill, B.: Patterns for Parallel Programming, 1st edn. Addison-Wesley Professional (2004)
27. McCool, M., Robison, A.D., Reinders, J.: Structured Parallel Programming: Patterns for Efficient Computation. Elsevier, Waltham, MA (2012)
28. 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: Proceedings of the 25th European MPI users' group meeting, EuroMPI'18, pp. 17:1–10. ACM, New York (2018)
29. Peng, I. B., Markidis, S., Gioiosa, R., Kestor, G., Laure, E.: MPI streams for HPC applications. In: New Frontiers in High Performance Computing and Big Data, number 30 in Advances in Parallel Computing, pp. 75–92 (2017)
30. Peng, I. B., Markidis, S., Laure, E., Holmes, D., Bull, M.: A data streaming model in MPI. In Proceedings of the 3rd Workshop on Exascale MPI, ExaMPI '15. Association for Computing Machinery, New York (2015)
31. Pieper, R., Griebler, D., Fernandes, L. G.: Structured stream parallelism for rust. In: 23rd Brazilian Symposium on Programming Languages (SBLP), SBLP'19, pp. 54–61. ACM, Salvador (2019)
32. Pieper, R., Löff, J., Hoffmann, R.B., Griebler, D., Fernandes, L.G.: High-level and efficient structured stream parallelism for rust on multi-cores. J. Comput. Lang. **65**, 101054 (2021)
33. Rayon: Rayon—Rust. https://docs.rs/rayon/1.4.0/rayon/, September 2020. Accessed 16 May 2021

34. Reinders, J.: Intel Threading Building Blocks, 1st edn. O'Reilly & Associates Inc, Sebastopol (2007)
35. 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, volume 1171 of Communications in Computer and Information Science (CCIS), pp. 102–116. Springer, Cham (2020)
36. Wagner, A., Rostoker, C.: A lightweight stream-processing library using MPI. In: 2009 IEEE International Symposium on Parallel Distributed Processing, pp. 1–8 (2009)