Journal of Computer Languages 73 (2022) 101160

Contents lists available at ScienceDirect

Journal of Computer Languages

journal homepage: www.elsevier.com/locate/cola

Combining stream with data parallelism abstractions for multi-cores

Júnior Löff [a], Renato B. Hoffmann [a], Dalvan Griebler [a,b,*], Luiz G. Fernandes [a]

[a] School of Technology, Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre, 90619–900, Brazil
[b] Laboratory of Advanced Research on Cloud Computing (LARCC), Três de Maio Faculty (SETREM), Três de Maio, 98910–000, Brazil

ARTICLE INFO

Keywords:
Programming language
Parallel programming
Stream parallelism
Data parallelism
Parallelism abstractions
Stream processing

ABSTRACT

Stream processing applications have seen an increasing demand with the raised availability of sensors, IoT devices, and user data. Modern systems can generate millions of data items per day that require to be processed timely. To deal with this demand, application programmers must consider parallelism to exploit the maximum performance of the underlying hardware resources. In this work, we introduce improvements to stream processing applications by exploiting fine-grained data parallelism (via Map and MapReduce) inside coarse-grained stream parallelism stages. The improvements are including techniques for identifying data parallelism in sequential codes, a new language, semantic analysis, and a set of definition and transformation rules to perform source-to-source parallel code generation. Moreover, we investigate the feasibility of employing higher-level programming abstractions to support the proposed optimizations. For that, we elect SPar programming model as a use case, and extend it by adding two new attributes to its language and implementing our optimizations as a new algorithm in the SPar compiler. We conduct a set of experiments in representative stream processing and data-parallel applications. The results showed that our new compiler algorithm is efficient and that performance improved by up to 108.4x in data-parallel applications. Furthermore, experiments evaluating stream processing applications towards the composition of stream and data parallelism revealed new insights. The results showed that such composition may improve latencies by up to an order of magnitude. Also, it enables programmers to exploit different degrees of stream and data parallelism to accomplish a balance between throughput and latency according to their necessity.

1. Introduction

Parallel hardware resources are ubiquitous. To fully exploit it, all new and legacy computing systems must employ concurrency and parallelism strategies. Nonetheless, writing parallel code while maintaining the correctness of applications may be a convoluted endeavor. The challenges of writing parallel systems are well documented in the literature [1–4]. Fortunately, academic and industry efforts have been conducted revolving around this issue for a long time, yielding fruitful and varied approaches to improve programmability aspects via abstractions. Parallel programming abstractions are organized into at least three layered abstraction levels, each one building on top of their lower level abstraction counterpart.

First, at the bottom or lowest level, programmers are equipped with low-level synchronizations, communicating queues, and other mechanisms specific to each architecture. The programmer manually coordinates computation using such mechanisms for a specific application and target hardware system [5,6]. In the second level are the structured parallel programming models. Most of them provide fundamental operators and parallel patterns that hide many parallelism intricacies such as load-balancing, communication, reductions, etc. Instead of lower-level mechanisms, the programmer may find ready-to-use building blocks (e.g., Map, flatMap, Reduce, Filter, etc.) and discern their semantics: *How and when to use?* [1,7,8]. At the highest parallel programming level, also known as higher-level parallelism abstractions [9,10] are the domain-specific languages (DSL) that ease the burden of writing parallel programs. Rather than manually designing parallelism strategies, the programmer specifies information about the data flow using a high-level language. This language leverages expert domain knowledge with a cleaner interface. Then, compilers, interpreters, and translators inspect the programmer's information and automatically assemble efficient parallel code. This is done employing building-blocks from the second parallelism layer that resembles the exact data flow from the programmer's information.

Our work focus on parallel programming abstractions at highest level. We employ SPar as use case to prototype our optimizations. Moreover, we provide extensions along with improvements to its language and compiler. SPar [10] is a domain-specific language (DSL) for expressing stream parallelism via code annotations. It provides five

* Correspondence to: School of Technology, Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre, 90619–900, Brazil
E-mail addresses: junior.loff@edu.pucrs.br (J. Löff), renato.hoffmann@edu.pucrs.br (R.B. Hoffmann), dalvan.griebler@pucrs.br (D. Griebler), luiz.fernandes@pucrs.br (L.G. Fernandes).

https://doi.org/10.1016/j.cola.2022.101160
Received 28 February 2022; Received in revised form 27 July 2022; Accepted 23 September 2022
Available online 3 October 2022

attributes for application programmers to annotate sequential code while the compiler selects the parallel pattern algorithms and handles complicated aspects of parallelization. Before this work, SPar was limited to coarse-grained stream-only parallelism. Modern stream processing systems may benefit from fine-grained internal data-parallelism computation to improve resource utilization and increase scalability. We highlight real-time constrained applications from computer vision, high-definition image processing, intelligent vehicles, and others. In these cases, performance may be improved by combining stream with data parallelism.

To the best of our knowledge, no previous work in the literature, except ours [11], has combined stream with data parallelism using code annotations. Therefore, in this work, we investigate the feasibility of introducing high-level and efficient combined stream with data-parallel abstractions using SPar as proof of concept. We provide an experimental evaluating using stream processing applications from different domains to determine which classes of applications may benefit from such composition of stream and data parallelism. We contribute for increasing SPar's language expressiveness, a new algorithm for parallel code generation, a prototype implementation in the SPar compiler, and performing performance analysis and trade-offs with representative applications.

Our investigations started in [11]. In this work, we provide a more mature research and significant improvements, presenting new insights with a consolidation of the experiments towards the composition of stream and data parallelism as well as future directions. In short, the content of this article newly adds: (1) a thorough explanation of the algorithms and low-level strategies that were designed for the compiler to support stream and data parallelism composition; (2) a new discussion regarding parallelization of applications with SPar combining stream and data parallelism. We explain SPar annotated code with high-level domain-oriented annotations and provide a discussion in terms of the automatically generated data flow using parallel patterns; (3) all experiments were executed in a different computer machine and collected additional metrics that lead to new insights. They revealed that the composition of stream and data parallelism can be used by stream processing systems to provide a balance between throughput and latency; (4) We assess the performance in a new stream processing application from the computer vision domain. It applies a neural network that trains a model using live data flowing through a stream for improving the model.

The remainder of this paper is organized as follows. Section 2 presents SPar's background and expands on our motivation towards the composition of stream and data parallelism. Section 3 introduces a high-level strategy for automatic data-parallel code generation. We explain parallelization blueprints on different stream and data-parallel applications using SPar in Section 4. Then, the experimental analysis is discussed in Section 5 using representative applications. Section 6 describes and discusses related work. Finally, Section 7 remarks conclusions and presents future directions.

2. SPar fundamentals

This section presents SPar, a C++ domain-specific language used in this work to provide high-level abstractions for parallel programming. SPar was developed with a focus on productivity and portability. Fundamentally, SPar is a programming model that provides parallel programming abstractions that can help programmers in the burden of writing efficient parallel code. It does so by arranging a clear separation between the application business logic code and parallelism strategies. Other benefits are towards minimizing sequential code refactoring, supplying an intelligible language syntax aiming for simplicity, minimal code intrusion, and hiding low-level complexities particular to concurrency, parallelism, hardware architecture, and computing platforms.

The SPar programming model leverages C++ code annotations coupled with high-level language features for expressing common configurations of stream processing applications. Then, internally, it uses its compiler tools and applies source-to-source code transformations to automatically transform these code annotations into efficient parallel code. This parallel code is represented using an underlying runtime parallel system, which by default is FastFlow [12]. The source-to-source transformations are performed directly on the standard C++ AST (Abstract Syntax Tree). Therefore, SPar's compiler maintains the full semantics of the C++ language and supports powerful source-to-source transformations. SPar may be reasoned about as two distinct modules: the SPar language and the SPar compiler. From a high-level perspective, the SPar user must only be aware of SPar language features while SPar compiler handles the rest. We explain the default SPar language in Section 2.1, and the Spar compiler in Section 2.2. Then, Section 2.3 introduces the motivation of our work via composition of data and stream parallelism.

2.1. SPar language

To use the SPar language, the user must insert C++11 annotations directly in the source code. This annotation system was first available in C++11 ISO release [13]. Annotations may be inserted almost anywhere in the code with no need for restructuring. They are delimited by double brackets and receive a list of attributes as input (e.g., [[attr-list]]). Interpreting these attributes is up to the compiler, but every C++11 compliant compiler must be able to parse and place them in the AST. This attribute list is the mechanism to describe the SPar language.

SPar's language was initially conceptualized containing five attributes: (1) **ToStream** denotes where the data stream starts and ends; (2) **Stage** denotes where a stage/block of sequential code starts and ends; (3 and 4) **Input** and **Output**, describe the input and output data of a `ToStream` or `Stage`; (5) **Replicate** is a special attribute for replicating a `Stage` for parallel execution.

```
1 [[spar::ToStream]] while(1){
2  i = read();
3  [[spar::Stage,spar::Input(i),spar::Output(i),spar::Replicate(4)]]
4  { i = filter(i);
5  }[[spar::Stage,spar::Input(i)]]{
6   write(i);
7 }}
```

Listing 1: Stream parallelism with SPar annotations.

Listing 1 shows an example of a traditional *read, then filter, then write* sample stream processing application parallelized with SPar. The `ToStream` and `Stages` declared in lines 1, 3, and 5 represent identifier attributes. With these annotations, SPar's compiler will identify that the loop in line 1 represents a data stream. Therefore, each item of this data stream (each iteration) will be consumed by two sequential `Stages` (line 4 and 6). In this example, i represent the stream item. Since i is created outside the `Stage`, we use `Input` and `Output` to communicate data between the `Stages`. Finally, the `Replicate` attribute informs that this `Stage` can be computed in parallel with the specified degree of parallelism (4 in the example).

2.2. SPar compiler

After the sequential code is annotated with SPar language, SPar compiler handles the process of interpreting, optimizing, and generating the final parallel code. SPar achieves this using CINCLE (Compiler Infrastructure for new C/C++ Languages Extensions) compiler infrastructure support [12]. CINCLE is a support tool that provides basic features and a simple interface to interact with the C++ AST (Abstract Syntax Tree), semantic analysis, and source-to-source code generation. Fig. 1 depicts a simplified SPar's compilation flow with CINCLE.

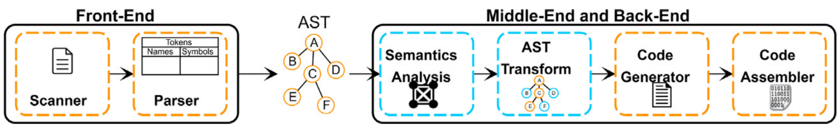


Fig. 1. SPar compilation flow.
Source: Extracted from [12].

The compilation process is divided into Front-end and a combination of Middle- and Back-end. The Front-end is responsible for parsing and analyzing the C++ code, reporting default C++ errors to the user with the support of GCC compiler. Then it uses the resulting tokens to create a fully C++ ISO compliant AST containing the SPar language annotations. With CINCLE support, this AST is fully accessible and modifiable, which permits code transformations to be performed directly on the AST during the compilation phase. This is one of the main advantages of CINCLE, since GCC and CLANG compilers do not support this kind of direct AST transformations [12]. Subsequently, the Middle- and Back-end phases of the compilation flow start with a semantic analysis that checks the AST for semantic errors in the SPar language features. If present, inconsistencies are reported back to the developer.

The next step of the compilation phase is the transformation step, performed directly on the AST. From a high-level perspective, it removes the SPar language attribute nodes and sub-trees from the AST. Conceptually, it then swaps them with the underlying parallel runtime system nodes and sub-trees. This process preserves the nodes and sub-trees that perform computations inside the stream stages and also other unrelated code. All of the re-structuring required by the underlying parallel runtime system is handled by the SPar compiler and the SPar user is not exposed to any of its intricacies. However, this step must be done with special attention to the C++ ISO [13], as errors can break the entire code. The final part of the compilation flow takes as input the transformed AST to generate the final binary file.

SPar also permits customizing some of the behavior of the runtime system using a few compilation flags:

- `spar_ondemand`: used to generate on-demand scheduling.
- `spar_blocking`: used to activate FastFlow blocking mode.
- `spar_ordered`: used to guarantee that output stream elements are delivered respecting the original input order.

2.3. SPar with stream and data parallelism

Past research revealed that SPar can improve programmability with negligible performance cost [10,14–16]. However, SPar targets stream parallelism only. In structured parallel programming, the Pipeline and Farm are parallel patterns commonly used to describe parallel stream processing applications. SPar employs both and also their semi-arbitrary composition. These parallel patterns combined enable SPar to implement a great number of modern stream processing applications. In addition to traditional stream processing, the SPar language can potentially be expressive enough to allow parallelism for some different parallelism domains (e.g., data parallelism, task parallelism, data-flow, etc.).

Although expressive for different domains, SPar is not always efficient. Sometimes the code generated by SPar may be inefficient, or at least not as efficient as other parallel strategies from literature. For instance, the Map pattern is more efficient than both the Farm and Pipeline for data parallelism applications. It contains schedulers, communicating queues, and synchronization mechanisms optimized for this application domain. Also, available optimizations like MapReduce may be used when data requires certain synchronization, in which deterministic accumulated results are obtained by combining partial

Fig. 2. Ecosystem of stream processing applications.

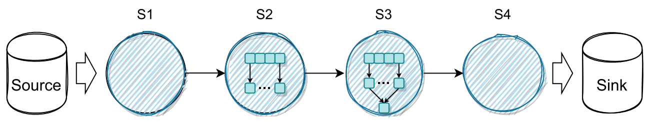


Fig. 3. Composing stream and data parallelism.

values. We discuss and analyze in greater detail this performance gap between parallel patterns in our experiment Section 5.

In this work, one of our contributions is extending SPar's programming model to support stream and data parallelism. In the ecosystem of stream processing applications (Fig. 2), today's computing workloads are implied in many applications containing internal regions with intensive data processing. Some highlights are machine learning, which implements convolution mathematical operations; natural resources exploration, which computes CFD routines (Computational fluid dynamics), wildfires reporting, which analyzes high-definition satellite images, among others. For these applications, adding an extra internal level of parallelism can increase performance by improving resource utilization.

Fig. 3 illustrates a composition of stream and data parallelism. The Figure also highlights the main goals of this work: we investigate the feasibility of combining stream and data parallelism using a single language abstraction; we investigate the benefits and efficiency such abstraction may yield. For that, we still keep stream parallelism for the coarse grain computation and exploit internal data parallelism. In this work, we exclusively target multi-core architectures. In the future, our data parallelism strategies may target architectures such as GPUs and FPGAs.

3. High-level data parallelism

In this section, we introduce new programming abstractions to simplify the development of data parallelism in C++ code. We extend SPar to support data parallelism instead of its default stream-only programming model. To do so, we increase the expressiveness of the SPar language and develop a new compiler algorithm to automatically generate data-parallel code. Our compiler algorithm generates parallel code using the FastFlow [17] runtime. In the future, the algorithmic strategies discussed in here could be applied to other parallel programming abstractions besides SPar. Furthermore, the automatic parallel code generation is not bounded to the FastFlow library and it is possible to modify the final phase of our compiler algorithm to generate code for other runtimes like OpenMP [18] and Intel TBB [19]. For example, in recent works SPar's original compiler already supports stream parallelism generating parallel code via OpenMP [20] and Intel TBB [21] runtimes.

The outline of the section is the following. Section 3.1 presents the technique we design for identifying data-parallel patterns in C++ sequential code. Then, we include it in SPar: (i) first by extending SPar's language in Section 3.2 and (ii) implementing a new compiler algorithm for source-to-source parallel code generation in Section 3.3.

3.1. Support for data parallelism

In this section, we reason about data-parallel patterns in C++ code. SPar's programming model does not support expressing information about data computational intensive regions in the code. It does not mean that SPar cannot parallelize data-parallel applications, however, the generated parallel code might not be efficient for data parallel computations. For example, SPar can be employed in matrix multiplication, which is a traditional data-parallel application. Listing 2 illustrates such implementation. The programmer annotates lines 1 and 3 with SPar's attributes. The `ToStream` attribute from line 1 indicates that the `for` loop is a data stream and each element (each iteration) is computed by a sequential computation from `Stage` (line 3). Using this sequence of annotations, SPar's compiler will automatically detect that a Farm pattern is suitable for this application, composed by an Emitter and parallel Workers (due to the `Replicate` attribute). Semantically, the generated code is functionally correct. Unfortunately, the Farm generated by SPar is not as efficient as as a true data-parallel pattern (e.g., Map and MapReduce). We later show through experiments (Section 5) the performance gap between these parallel patterns.

Our work adds new supported parallel patterns to improve SPar's parallelism-awareness and enhance its automatic parallel code generation. Consequently, more suitable parallel patterns are applied depending on the information the programmer annotates in the sequential code. In this work, we focus on the recurrently used data-parallel patterns: Map and MapReduce. Therefore, in addition to the parallel patterns already supported by SPar (Pipeline and Farm), the source-to-source code generation will also consider the Map and MapReduce patterns.

```
1  [[spar::ToStream]]
2  for(long int i=0; i<SIZE; i+=1){
3   [[spar::Stage, spar::Input(i), spar::Replicate()]]
4   for(long int j=0; j<SIZE; j++){
5    for(long int k=0; k<SIZE; k++){
6     matrix[i][j] += (matrix1[i][k] * matrix2[k][j]);
7  }}}
8
```

Listing 2: Matrix multiplication parallelized with SPar.

A Map pattern may be introduced when there is a known indexed data set locally stored in the multi-core architecture. This indexed set can be empty but must be available locally. That is because data stored in network file systems or cloud storage usually is associated with I/O bottlenecks that hinders data parallelism viability. Consequently, these systems could be better modeled with default stream parallelism using efficient schedulers. The MapReduce parallel pattern is a special case of Map where iterations exhibit certain data dependencies. In this situation, all parallel replicas work towards solving a slice of the data and results are later combined into a single output.

To support these attributes, SPar must apply a strict definitions of the characteristics that define a Map or MapReduce pattern. Data-parallel patterns are more restrictive than stream parallelism patterns. Thus, the compiler algorithm is more complex. Data patterns require the dataset size to be fixed and known a priori. On the other hand, stream parallelism may handle bounded or unbounded datasets. Data patterns are usually associated with C++ `for` loops while stream patterns can deal with any logic that provides a continuous flow of incoming data. To get the dataset size, we design an algorithm that locates the boundaries of the indexed data set (start and end) and iteration step. With this information, if not explicit, the size can be calculated implicitly. For instance, using C++ syntax: `start=0`, `end=9`, `iteration_step+=1` (10 elements); or, `start=2`, `end=64`, `iteration_step*=2` (6 elements).

We designed a scheme that delineates data-parallel patterns based on sequential C++ `for` loop syntax. In fact, our design also considers `while` and `do_while` loops. Map and MapReduce patterns can be fully exploited by the SPar compiler when they match our proposed scheme for sequential C++ `for` loop syntax. In other words, every constant and function we define must be located by the compiler during compilation. Listing 3 shows a high-level representation of Map (line 1) and MapReduce (lines 2 and 3) patterns. The `lhs` and `rhs` are the left- and right-hand sides data boundaries. We did not name them `start` and `end` since `for` loops can be in either ascending or descending order. The `it` stands for iteration step. The `lhs`, `rhs`, and `it` must be static and cannot be modified after the `for` loop execution started. The `type` and `type2` can be a standard language type (i.e `int, long int, double`) or custom types (i.e. `struct, class`). The `exp` and `op` stand for expression (i.e. `<, >, <=, >=, !=`) and operation (i.e. `-=, ++, *=, &=, |=`), respectively. The loop (line 3) uses a shared variable. This is a common example of a MapReduce pattern that must implement synchronizations to safely parallelize the code.

The MapReduce scheme definition is important because the standard C++ ISO is broad and allow multiple ways of expressing the same action. For example, the data type of the iteration variable does not need to be declared within the `for` loop. Therefore, our compiler algorithm uses the `id` (identifier) and traverses the abstract syntax tree (AST) to find where the variable was declared and to extract its data type. In a second case, our scheme defines `exp()` and `op()` as functions. That is because they receive static values or variables (`rhs` and `it`) and C++ permits modifying values according to the operator. For instance, the tokens `<` or `<=` modify the right-hand side boundary to either `rhs+0` or `rhs+1` by increasing the number of iteration by zero or one, respectively.

```
1 for(type id=lhs; id exp(rhs); id=id op(it)){ }
2 type2 id2;
3 for(type id=lhs; id exp(rhs); id=id op(it)){ id2=id2 op2(v)}
```

Listing 3: High-level Map and MapReduce representation.

Until this point, we have only defined the Map and MapReduce patterns based on plain C++ syntax. However, this does not semantically guarantee that the sequential code can be safely parallelized. For example, as shown in previous Listing 3 at line 3, the data dependency may not be implemented in terms of Reduce pattern (e.g., accumulations such as sums and multiplications). Some dependencies cannot be reduced. In particular, when a different order of computations modifies the result. In that case, the automatic parallel code generation would be incorrect.

According to a recent study [22] that has conducted a deep analysis to evaluate quantitative and qualitative aspects of auto-parallelization compilers, they concluded these compilers need more sophisticated techniques for parallel code analysis. In this study, the authors described many problems: starting from execution errors, missing parallel loops, incorrect semantics, and inefficient code generation. Their findings revealed that current state-of-the-art compiler strategies cannot ensure correctness.

In our work, we designed the compiler algorithm to base its information on user-oriented annotations. Therefore, we expect the user to provide the correct information of sequential block of code that can be safely parallelized. This approach is employed by almost any parallelism framework such as OpenMP, Threading Building Blocks, C++ Parallel STL, and others. Different from them, we do not expose the user to low-level details and expect him to manually write parallel code. Instead, we provide high-level programming abstractions that equip the programmer with a small set of domain-specific attributes he can use to express parallelism information on the code. Later, the compiler reads the information and produces parallel code.

In Sections 3.2 and 3.3, we present the SPar language and compiler extensions. First, we describe two new attributes we included to increase SPar's language expressiveness. Then, we describe a new compiler algorithm and the transformation rules we implemented for SPar to support stream and data parallelism composition.

3.2. SPar language extension

The original SPar language focuses on a language to express stream parallelism in the code. However, it lacks expressiveness to represent data parallelism. Consequently, we need to extend the SPar language with new attributes to support data parallelism: `Pure` and `Impure`. The `Pure` attribute is a term already defined in functional programming for describing functions implemented in their purest form. This means that the results (outputs) depend only on the input parameters and the computation has no side effect. No side effects determine that a pure function must only: perform a computation based on input data; and return the result or store it in a locally private memory address. Examples of side effects are:

- Modify a globally shared variable;
- Read or write files;
- Synchronizations such as `return`, `break`, and `socket`;

In functional programming, the "pure" definition has many properties. In SPar, we limit pure functions to the parallelism property. Therefore, our pure definition carries the information that a block of code annotated with `Pure` can be executed in parallel with no restrictions. The programmer can use this attribute to annotate regions of sequential code that can benefit from parallelism, or in simplified terms, regions (usually loops) that take a long time to compute. Then, SPar analyses these regions trying to find opportunities to optimize the code. Currently, we apply optimizations via a Map parallel pattern targeting CPUs-only. In the future, other suitable optimizations may be applied using different parallel patterns (e.g., pack, filter, etc.), exploiting compiler vectorizations, or offloading the computation into specialized hardware like GPUs and FPGAs.

The next attribute included in the SPar language is the `Impure`. This attribute allows otherwise impure code regions to be annotated as pure. For example, if inside a function there is a single line of code with side effects, by definition all the function is considered not pure, or impure. Therefore, the `Impure` attribute may be used to annotate that specific code region. This way, the compiler can "purify" the function and allow parallelism transformations. In SPar, annotating a code region with `Impure` means that SPar will try to automatically implement the required synchronization to allow parallelism. By default, an impure region is protected using locks. Before that, the compiler checks the code trying to detect ways to optimize the synchronization without locks.

In this work, we already detect Reduce operations which we optimize using the MapReduce pattern. This option is better than synchronizing all threads with locks when a large number of cores is available. Other optimizations can be implemented in the future using speculative synchronization mechanisms or even other parallel patterns (e.g., stencil, scan, etc.) and their implications with the `Impure` attribute.

Listing 4 shows an example of parallelization using the matrix multiplication algorithm with the new attributes. Parallelizing it with the new SPar language is straightforward. Compared to the original SPar language, it only requires the programmer to annotate the code using two extra attributes: `Pure` and `Impure`. Line 3 annotates a `Pure` attribute, meaning the entire loop can be safely parallelized. However, lines 9 and 10 have side effects and the user can annotate this impure block of code using `Impure`.

```
 1 [[spar::ToStream]]
 2 for(long int i=0; i<SIZE; i+=1){
 3  [[spar::Stage, spar::Pure, spar::Input(i), spar::Replicate()]]
 4  for(long int j=0; j<SIZE; j++){
 5   for(long int k=0; k<SIZE; k++){
 6     matrix[i][j] += (matrix1[i][k] * matrix2[k][j]);
 7     [[spar::Impure]]
 8     {
 9      sum += matrix[i][j];
10      sum_line[j] += matrix[i][j];
11     }
12 }}}
```

Listing 4: Matrix multiplication with new attributes.

Listing 5 provides a sequence of commands to show how the compiler should transform the annotations into the appropriate patterns. `Pure` means the computation annotated by this attribute is in its purest form, has no side effects, and depends only on its inputs. Therefore, we encapsulate and abstract this block of code into a single function call `pure_function()`, as show in line 2. In this example, the parallelization is not safe yet, because it is conditioned to a compiler automatically purifying the impure block of code. Once the compiler purifies the impure region, it can parallelize the code using Map (line 1), and each parallel worker computes a partial result (line 2). In the end, the partial results are accumulated into a single output using the Reduce pattern (line 3).

```
1 MAP i = 0,1,...,SIZE
2  sum_local = pure_function(i);
3 REDUCE sum += sum_local;
```

Listing 5: High-level annotated attributes

3.3. SPar compiler implementation

In this section, we start from previous Listing 5 and disclose how our new compiler algorithm interprets the SPar language via code annotation and automatically generates suitable parallel patterns using source-to-source code transformations. Fig. 4 illustrates a flowchart of our implementation methodology, which is discussed in the subsequent Sections Section 3.3.1 up to Section 3.3.4. Each column represents a compiler phase of our algorithm.

3.3.1. Semantic analysis

We start by extending the SPar compiler to support two new attributes included in SPar's language: `Pure` and `Impure`. The first compiler step is responsible for traversing the C++ AST and performing a semantic analysis to verify the annotated attributes correctness. In the analysis, we only check for SPar semantics and expect the programmer

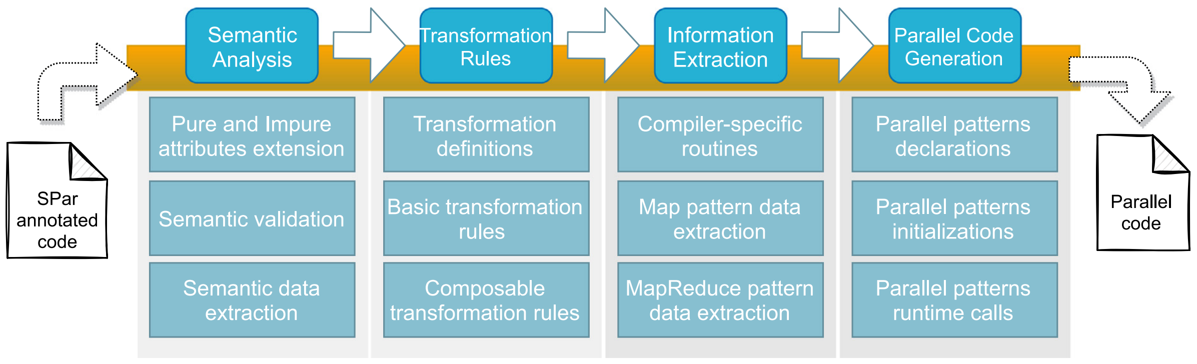

SPar annotated code
Semantic Analysis
Transformation Rules
Information Extraction
Parallel Code Generation
Pure and Impure attributes extension
Semantic validation
Semantic data extraction
Transformation definitions
Basic transformation rules
Composable transformation rules
Compiler-specific routines
Map pattern data extraction
MapReduce pattern data extraction
Parallel patterns declarations
Parallel patterns initializations
Parallel patterns runtime calls
Parallel code

Fig. 4. Implementation methodology used to implement the new compiler algorithm in SPar.

to provide correct information about the data flow. Available technology does not allow static analysis to ensure that a `Pure` or `Impure` are in fact functionally correct. For example, our analysis is limited to checks such as: a `Impure` must be declared within a `Pure`, and `ToStream` must be the outermost attribute annotation.

During the AST traversal, the compiler also gathers crucial semantic information about the composition of the annotations. This is information about where `ToStream` was declared, how many internal `Stages` or `Pures`, `Input` and `Output` variables, and others. The annotations' semantic analysis is necessary so that later it is guaranteed that the compiler logic will converge and select a suitable parallel patterns. The `Stage` attributes may become Pipeline stages or Farms, and `Pure` and `Impure` may become Map or MapReduce patterns.

By default, the compiler only analyzed the `ToStream` scope because it was the only attribute that could be composed with multiple internal `Stage` attributes. The composition of `Stage` inside `Stage` is not semantically permitted. In our extension, we included two new attributes that can be semi-arbitrarily composed. The `Pure` attribute may be declared as an identifier attribute inside a `Stage` scope or as an auxiliary attribute of a `Stage`. The `Impure` attribute must be declared as an identifier attribute inside a `Pure` scope. To check their composition we implemented two new routines that account for the `Pures` inside a `Stage` and the `Impure` inside a `Pure` scope. Once the SPar language information is collected, the next compiler step uses the information combined with pre-defined transformations rules set to decide for a suitable parallel patterns.

3.3.2. Transformation rules

For the SPar compiler, we have proposed new definitions and transformation rules to support source-to-source code transformations combining data and stream parallelism. SPar's definitions and transformation rules were originally proposed in [10]. They focus on stream parallelism and we extended them in this work to also support and combine data parallelism. Table 1 shows all of the definitions, including new ones we created and applied modifications.

In Table 1, a $\square$ is a generic block of code and the scope of the sentences is represented by $\{...\}$. Annotations are marked as $[[...]]$ and may contain a list of attributes as an argument. The ID attributes are T (`ToStream`), S (`Stage`), P_n (`Pure`), and I_n (`Impure`). The AUX attributes are I (`Input`), O (`Output`), R_n (`Replicate`), and optionally P_n (`Pure`). Regarding parallel Patterns, the Pipeline is defined as $Pipeline(S_1, S_2, \ldots)$ and is composed of a list of sequential computational stages S_n. The Farm is defined as $Farm(E, W, C)$, where E is a sequential stage that also distributes data items among the workers, W is a group of parallel worker threads and C is an optional sequential data collection stage. Any composition of these patterns may be used (e.g. $Pipeline(S_1, Farm(E, W), S_2)$). New parallel patterns are Map defined as $Map(\square)$, and MapReduce defined as $MapReduce(\square, \square)$.

Inclusions and modifications in Table 1 are highlighted with the teal color. This table can be seen as the SPar compiler heuristic to select a suitable parallel pattern. Each time the compiler is called it performs the semantic analysis. Then, it inspects this definition table from D_0 up to D_{n-1} until a definition is satisfied. The first satisfied definition is applied. Then, the compiler starts again from D_0. It repeats this operation until no definition is satisfied.

The new transformation rules we included can be classified in two groups. The first group contains two basic transformation rules, where a code annotated with SPar can be transformed in Map or MapReduce patterns. The second group contains two composable transformation rules that target parallel code generation using pattern composition. These are a set of transformation rules supporting the composition of stream-parallel patterns (Pipeline and Farm) with data-parallel patterns (Map and MapReduce).

To describe the new definitions and transformation rules, we extended the original notation. The attributes `Pure` and `Impure` are represented by the P_n and I_n notations, respectively. In our work, the Map and MapReduce parallel patterns are analogous to C++ `for` programming loops, we represent those with $\forall_n$. The first two rules use D_5 and D_6 to perform aggressive transformations that enable data parallelism in SPar. These rules completely ignore the streaming counterpart and transform the entire `ToStream` into pure data-parallel patterns. We briefly explain them in the following.

Rule (1) is created via definition D_5, since a T (`ToStream`) contains as its first declaration a single $\forall$ (`for`) that is annotated with a S (`Stage`) containing in its attribute list a P (`Pure`) and R (`Replicate`), and inside the $\square$ (block of code) does not contain an I_n. In this rule, the $\forall$ becomes a Map parallel pattern and the $\square$ computation is passed as its input parameter.

$$[[T_0]]\{\forall_0\{[[S_0, P_0, R_n]]\{\square_0\}\}\} \\ \Downarrow \\ Map(\square_0) \tag{1}$$

Rule (2) is similar to the previous one since it converges into a Map combined with a Reduce. This Rule is created via definition D_6. Accordingly, the rule defines that a T may become a MapReduce when it contains as first declaration a single $\forall$ with a single $\square$ annotated with a S containing in its attribute list a P, R, and inside the $\square$ contains an I_n.

$$[[T_0]]\{\forall_0\{[[S_0, P_0, R_n]]\{\square_0, [[I_0]]\{\square_1\}\}\}\} \\ \Downarrow \\ MapReduce(\square_0, \square_1) \tag{2}$$

The other two transformation rules we added enable composition of stream with data parallelism. These transformation rules are executed in two steps. Rather than transforming the entire `ToStream` into a Map or MapReduce, our compiler algorithm only transforms the internal region from $\square$ into data-parallel patterns and combines them with coarse-grained stream parallelism. Therefore, a single $\square$ becomes a black box that may contain none, one, or more data-parallel patterns. Subsequently, in the second step, a $\square$ may become a `Stage` S_n in a

Table 1
New definitions designed to support the data parallelism transformation rules.

D_0	A generic stage ψ is automatically generated for gathering results when the last $\square$ is annotated with S containing in its attribute list a R_n and O.
D_1	A $\forall_n$ may become a Map when its first declaration is a $\square$ annotated with S containing in its attribute list a P_n, and inside the $\square$ does not contain an I_n.
D_2	A $\forall_n$ may become a MapReduce when its first declaration is a $\square$ annotated with S containing in its attribute list a P_n, and inside the $\square$ contains an I_n..
D_3	A $\square$ annotated with S and may containing a Map or MapReduce can become a stage S_n in a Pipeline, or a stage E or C in a Farm when its attribute list does not contain a R_n.
D_4	A $\square$ annotated with S containing in its attribute list a R_n and may containing a Map or MapReduce can only become a W stage in a Farm.
D_5	A T may become a Map when its first declaration is a $\forall_n$ with a single $\square$ annotated with S containing in its attribute list a P_n and R_n, and inside the $\square$ does not contain an I_n.
D_6	A T may become a MapReduce when its first declaration is a $\forall_n$ with a single $\square$ annotated with S containing in its attribute list a P_n and R_n, and inside the $\square$ contains an I_n.
D_7	A T becomes a Farm when the first S annotation contains in its attribute list a R_n, when there are two S at most, and D_5 or D_6 are not satisfied.
D_8	A T becomes a Pipeline when the first S annotation does not contain in its attribute list a R_n, when there are more than two S annotations, and D_5 and D_6 are not satisfied.
D_9	A Farm becomes a S_n stage in a Pipeline when D_7 is not satisfied and a $\square$ is annotated with S containing in its attribute list a R_n.

Pipeline, or a E, W, or C in a Farm. This way, our compiler algorithm allows the composition of data and stream parallelism.

Rule (3) exemplifies a data and stream parallelism composition. According to definition D_1, a $\forall$ that contains a $\square$ annotated with P as its first declaration will become a Map pattern. So, in the compiler first iteration the $\forall_0\{[[P_0]]\{\square_2\}\}$ is transformed into a $Map(\square_2)$ and wrapped into a $\square_m$ black box that is inserted in the C++ AST. In the second step, the T is transformed into a Farm according to definition D_7, in which $\square_0$ becomes the E and the compiler concatenates $\square_1$ with $\square_m$ while passing them as a parameter to the W in a Farm.

$$\begin{aligned}
&[[T_0]]\{\square_0,[[S_0,R_n]]\{\square_1,\forall_0\{[[P_0]]\{\square_2\}\}\}\}\\
&\qquad\Downarrow\\
&\textbf{Step1}:\square_m = Map(\square_2)\\
&[[T_0]]\{\square_0,[[S_0,R_n]]\{\square_1,\square_m\}\}\\
&\qquad\Downarrow\\
&\textbf{Step2}:Farm(E(\square_0),W(\square_1\square_m))
\end{aligned}\tag{3}$$

3.3.3. Information extraction

In the previous compiler phase, the compiler selects suitable parallel patterns. Once SPar's compiler determines the parallel pattern or their composition, it extracts the necessary data. This step is responsible for executing compiler routines that traverse the C++ abstract syntax tree (AST) to gather information regarding the new Map and MapReduce patterns. Data parallelism is more restricted and patterns can only be generated when all necessary information is extracted. Therefore, in this compiler step, both the information extraction and C++ syntax analysis routines can abort the data-parallel patterns generation. If this happens, our compiler algorithm resumes the original SPar execution flow only generating stream parallelism patterns.

In Section 3.1, we presented our strategy for detecting data-parallel patterns in the sequential code. We define the basic data required to generate the Map and MapReduce patterns based on C++ syntax. This step must extract essential data such as identifiers, variable types, indexed set size, and others. To extract these information, we have implemented parsing algorithms based on standard C++17 ISO [23], that specifies a `for` loop as:

`for (init-statement condition`$_{\texttt{opt}}$`; expression`$_{\texttt{opt}}$`) statement`

We designed an FSM (finite-state machine) algorithm for gathering the information about each one of the three fields of C++ `for` loops. Before entering the FSM, the block of code annotated with SPar is transformed into a list containing an internal C++ AST representation of all nodes and their literal data representation (e.g., `}`, `<`, `int`, `double`). Then, our FSM receives as input this list and uses the literal string while processing the characters individually. Each new character that enters the FSM is accumulated in each state and transitions when special tokens are detected. For example, in a traditional `for` loop design we expect the programmer to write the following code:

```
for (int i = 0; i < SIZE; i++) {...}
```

In the `init-statement` we expect to extract three information: the identifier, identifier data type, and left-hand side boundary. Therefore, our FSM has three states, one for each information. In this situation, the special tokens are an `identifier`, `eq_token`, and `semicolon_token` (based on C++17 ISO). In our example, everything before `identifier` is the data type, all in between the data type and `eq_token` is the identifier, and finally, all in between the `eq_token` and `semicolon_token` is the left-hand side boundary. At the end, the identifier is `i`, data type is `int`, and left-hand side boundary is `0`. We design this algorithm for extending the C++ expressiveness to our programming model. This way, there is no difference if the programmer uses as data type a `int`, a `unsigned int`, or a custom `struct` or `class` object.

In the first information extraction step, if at the end of execution only the variable data type is missing, we implemented an optimization to find the variable data type using its identifier. The importance of this optimization relies upon the fact that many applications may declare variables in different regions of the code. Consequently, we implemented a new compiler routine that traverses the C++ AST and locates where the variable was declared to obtain its data type. This is also important for the `Impure` region since global shared variables were certainly declared outside the scope of the code annotated with SPar.

In the second step, we identify the right-hand side boundary and loop expression in the `condition` field. This time, we implemented an FSM with only two states. First, we extract the expression matching the input with the standard C++17 token. In this work we consider the tokens >, <, >=, and <=. Everything in between these tokens and `semicolon_token` is the right-hand side boundary. When the FSM detects the tokens >= or <= it additionally increments the right-hand side value by 1 for maintaining logical correctness.

Finally, the last step uses `expression` to extract the operation, iteration step, and loop direction. In this step, we do not employ an FSM. Our work considers the following C++ tokens: ++, --, +=, and -=. The loop direction is straightforward and can be inferred depending on if the iteration step is incremented or decremented. The iteration step is always 1 for the first two tokens (++ and --) since they are C++ abstractions for unitary incremental. However, for the two last tokens (`+=` and `-=`), the value is extracted in between these tokens and `semicolon_token`.

Map and MapReduce patterns exploit the aforementioned algorithm to extract vital information for their Map counterpart. Nonetheless, MapReduce has a second extraction step that gathers information for the `Impure` region of the code. They are the reduce identifier, identifier data type, operation, and, optionally, the identifier size. In our reduce algorithm, we consider recurrent code representation of reduce operations, such as: `+=`, `-=`, and `=` composed with `+`, `-`, `*`, `^`, `&`, and `|`. The composition of `=` with a C++ `operation` (e.g., `var = var + 1;`) is more complex and we use an FSM with four states, while the `+=` logic uses an FSM with two states. The identifier and operation are extracted from the `Impure` region. However, in Reduce operations, since the variable is shared it was almost certainly declared in another code region. This way, we use the unique C++ identifier to locate where the variable was declared. If it is found, we extract the data type along with the array size when required.

As an optimization, we included in our algorithm a parser that enable reductions using static arrays, since our algorithm sees little variation between a variable declared as `sum += partial_sum;` or declared as `sum[i] += partial_sum;`. Many state-of-the-art programming abstractions do not support pointers, only via manual implementation. For now, we only support arrays that are statically allocated.

3.3.4. Parallel code generation

Finally, if data is correctly extracted, our compiler automatically generates the appropriate parallel patterns using FastFlow runtime calls. The code generation is composed of three different routines for patterns declaration, initialization, and runtime calls. Listing 6 shows a slice of the parallel code automatically generated by our compiler algorithm. The code represents the matrix multiplication algorithm annotated with SPar in previous Listing 4. This example gives an idea of how many parallelism details a programmer must deal with to parallelize an application, even if the application is as simple as matrix multiplication.

Lines 1 and 2 store a copy of the original values that require synchronization. In this case, memory copy is necessary because the reduction is performed using an array. Line 3 initializes the reduction variables since FastFlow requires the initial value and an empty reference. Lines 4 to 16 implement FastFlow's MapReduce parallel pattern schema, which is based on C++ lambda functions. We replace the lambda function by the `pure_function()` block annotated with `Pure`. Finally, in lines 17 and 18 the accumulated reduction values are assigned to the original variables.

```
1  spar_global.sum = sum;
2  memcpy(spar_global.sum_line, sum_line, sizeof(int)*SIZE);
3  spar_reduce reduce_clean, reduce(spar_global.sum, spar_global.
       sum_line);
4  spar_pf->parallel_reduce(reduce, reduce_clean, 0, SIZE+0, 1,
5  [&] (long int i, spar_reduce & reduce) {
6   for(long int j = 0; j < SIZE; j++){
7    for(long int k = 0; k < SIZE; k++){
8     matrix1[i][j] += (matrix1[i][k]*matrix2[k][j]);
9     reduce.sum = reduce.sum+matrix[i][j];
10    reduce.sum_line[j] += matrix[i][j];
11   }
12  }
13 },
14 [&] (spar_reduce & reduce, const spar_reduce spar_aux_reduce) {
15  reduce += spar_aux_reduce;
16 });
17 sum = reduce.sum;
18 memcpy(sum_line, reduce.sum_line, sizeof(int)*SIZE);
```

Listing 6: MapReduce parallel code generation.

The MapReduce requires an additional reduction `struct`. As an example, in Listing 7 we provide the reduce `struct` generated by our compiler algorithm for the same matrix multiplication application. As can be seen, the code is quite complex when reductions are performed over arrays. The compiler completely abstracts these complexities from the programmer. We couple the generated struct directly with the mechanisms required by FastFlow. Other runtime libraries may slightly modify this reduce `struct` for their purposes.

Lines 2 to 6 in Listing 7 show the logic that will be used to initialize all partial variables. Between lines 7 to 9 is located the logic that initializes all shared variables with the current values captured during execution time. Note that arrays cannot simply assign data between pointers, instead, we use `memcpy` to initialize them. Finally, Lines 10 to 16 contain the logic that will be used to combine partial results. Our compiler algorithm leverages C++ operator overloading to optimize this step.

```
1  struct spar_reduce{
2   spar_reduce() : sum(0.0) {
3    for(int i=0; i<SIZE; ++i){
4     sum_line[i] = 0.0;
5    }
6   }
7   spar_reduce(int sum, int * sum_line) : sum(sum) {
8    memcpy(this -> sum_line, sum_line, sizeof(int)*SIZE);
9   }
10  spar_reduce & operator +=(const spar_reduce & v) {
11   sum += v.sum;
12   for(int i=0; i<SIZE; ++i){
13    this->sum_line[i] += v.sum_line[i];
14   }
15   return *this;
16  }
17  int sum;
18  int sum_line[SIZE];
19 };
```

Listing 7: Reduce struct code generation.

To generate source-to-source parallel code we employ FastFlow's Map and MapReduce programming abstractions and initialize them using the information extracted in the previous compiler phase. The FastFlow templates for the Map and MapReduce patterns are presented in Listing 8. Note that FastFlow is a recent parallel library that uses well-known data-parallel patterns. These patterns' API is either equivalent or at least similar between different solutions. Therefore, other C++ libraries that also leverage data-parallel patterns may be used for providing further extensions to our programming model, such as OpenMP, Intel TBB, GrPPI, Kokkos, etc.

```
MAP ( left-hand side boundary, right-hand side boundary, iteration
      step, pure_function() )

MAP REDUCE ( impure variables, initial values, left-hand side
      boundary, right-hand side boundary, iteration step,
      pure_function(), impure_function() )
```

Listing 8: FastFlow templates for the Map and MapReduce parallel patterns.

4. Parallelism with SPar

In this section, we explain the strategies employed to parallelize applications using the SPar programming model. We selected real-world computations from both stream and data parallelism domains. Section 4.1 discusses the NAS Parallel Benchmarks (NPB) considering different kernels and pseudo-applications from the data-parallelism domain. Then, Section 4.2 discusses our strategies for parallelizing three stream processing applications composing stream with data parallelism. We extracted or adapted from the stream processing domain the Mandelbrot set, Lane Detection, and Computer Vision applications.

4.1. Data parallelism

The NAS Parallel Benchmarks is a popular suite used to evaluate massively parallel systems. It contains eight benchmarks extracted from the computational fluid dynamic (CFD) domain representing real computations with realistic workloads. The workload represents a heavy mathematical computation that can be easily found in popular scientific HPC applications. The NPB is maintained by the NASA advanced supercomputing (NAS) division and receives updates to be compliant with modern computing systems. We selected six benchmark applications for our experiments, four kernels, and two pseudo-applications. The remaining two applications are not considered because they require complex global synchronization strategies that are out of the scope of this work. The four kernels are briefly described as follows:

- **Embarrassingly Parallel (EP)**. It generates a large number of Gaussian random deviates and computes the Gaussian deviation. This method is useful to stress floating-point operations [24].
- **Multi Grid (MG)**. It computes a scalar Poisson equation where the kernel continuously alternates between coarse and fine grids to perform restriction and prolongation operations. It generates irregular data communications that result in numerous cache misses [24,25].
- **Conjugate Gradient (CG)**. It computes an approximation of the smallest eigenvalue of an unstructured matrix. This kernel stresses data communication mechanisms [25].
- **Discrete 3D Fast Fourier Transform (FT)**. It computes a Fast Fourier Transform (FFT) of a 3D partial differential equation. This kernel requires intensive data communication [24,25].

The two pseudo-applications implement different methods to solve a Navier–Stokes system of differential equations. We summarize them as follows:

- **Block Tri-diagonal solver (BT)**. Computes the Alternating Direction Implicit (ADI) factorization on a 3D matrix. Then, it solves the unknown vectors using the back substitution method for each direction [25].
- **Scalar Penta-diagonal solver (SP)**. It uses the Beam-Warming approximate factorization to decompose the 3D matrix. Then, the unknown vectors are solved using the tridiagonal matrix algorithm and the back substitution method [25].

The NPB sequential code is mainly composed by multiple routines that implement computational intensive loops. We identified such loops and annotated them using the available SPar features. During the parallelization, we observed that the SPar language and its five attributes are suitable for implementing some data-parallel applications. However, its flexibility still lacks key internal mechanisms to readily exploit data parallelism. In the following, we highlight some drawbacks that we tackle in this work.

1. SPar generates code based on the scope that it was declared. Therefore, for each new function/class/struct/loop iteration, it generates and re-initializes all of its mechanisms, introducing huge performance penalties. In our work, we minimize performance penalties by using global pointers, initializing them once, and reusing parallel mechanisms.
2. SPar cannot correctly capture some variables via `Input` and `Output` attributes. For example, a `double q[NQ]` may be interpreted as `double q`. Instead, we designed a new algorithmic scheme based on an FSM logic, which was described in Section 3.3.3.
3. SPar requires that every variable that enters or leaves its scope are declared via `Input` and `Output` attributes. This can be tricky in complex applications that require manually entering a big number of variables. In our new version, we employ standard C++ lambdas to intermediate variable capture.

Due to the previous drawbacks, we parallelized only EP, FT, and CG with the original SPar. These applications provide an overview of SPar's performance behavior. We based our parallel procedure on a handwritten FastFlow implementation [26,27]. However, knowing SPar's overhead penalties, we understand that some potential parallel regions should not be implemented because the parallel gains do not compensate for the overhead.

Using SPar with data parallelism, we were able to implement all six applications. Our parallel procedure was equivalent to a lower-level programming abstraction such as FastFlow. In CG the FastFlow parallel procedure uses dynamic scheduling in the most computational intensive loop. On the other hand, SPar supports default static scheduling. In the MG kernel, we implement one less MapReduce concerning the FastFlow versions. The reason is that our current compiler algorithm considers summation while the application requires a reduction of type `max`.

Concerning the FT kernel, it uses a custom variable data type to represent complex numbers. At some point in the code, FT requires a reduction operation over this complex type. In SPar, however, the MapReduce pattern only recognizes standard C++ data types. So, we adapted the code by separating the real and imaginary part of the complex number into two variables of `double` type. The remaining three NPB applications have an equivalent parallelism procedure with respect to the FastFlow version.

In Listing 9 we illustrate a working example of NPB's data-parallel applications parallelized with SPar. This procedure extends to all NPB applications. It shows an internal routine of CG kernel we implemented using the new SPar language. For the original SPar, the code is similar except the recent `Pure` and `Impure` attributes. Consequently, programmers must manually deal with impure regions or not parallelize this code regions at all. Besides, in original SPar the programmer is expected to inform the input and output variables for each `ToStream` and `Stage`.

```
[[spar::ToStream]]
for (int j = 0; j < lastcol - firstcol + 1; j++) {
 [[spar::Stage, spar::Pure, spar::Replicate()]]
 {
  [[spar::Impure]]
  {
   d = d + p[j]*q[j];
  }
 }
}
```

Listing 9: CG's internal routine parallelization with SPar.

4.2. Stream and data parallelism composition

In this Section, we discuss the strategies designed for parallelizing three stream processing applications. We leverage composition of stream and data parallelism to exploit the maximum parallel resources of the underlying multi-core architecture. This combines two levels of parallelism. Fig. 5 shows the flow-graph for each application. Mandelbrot set and Lane detection was implemented using three computational stages where the first and last one perform sequential I/O operations. The middle one is a parallel stage that performs the bulk of the computation. This composition is equivalent to a Farm pattern that has replicated workers with internal data parallelism inside. On the other hand, Computer Vision was implemented using 4 stages. The first stage reads data. The second stage is a Neural Network that can only be parallelized using data parallelism strategies. The third stage is a replicated stage that acts as the Workers of the Farm pattern. The fourth sequential stage combines the output, applies another computational filter to the image, and then writes the output to disk. Next, we describe each application and its parallelization strategies in greater detail.

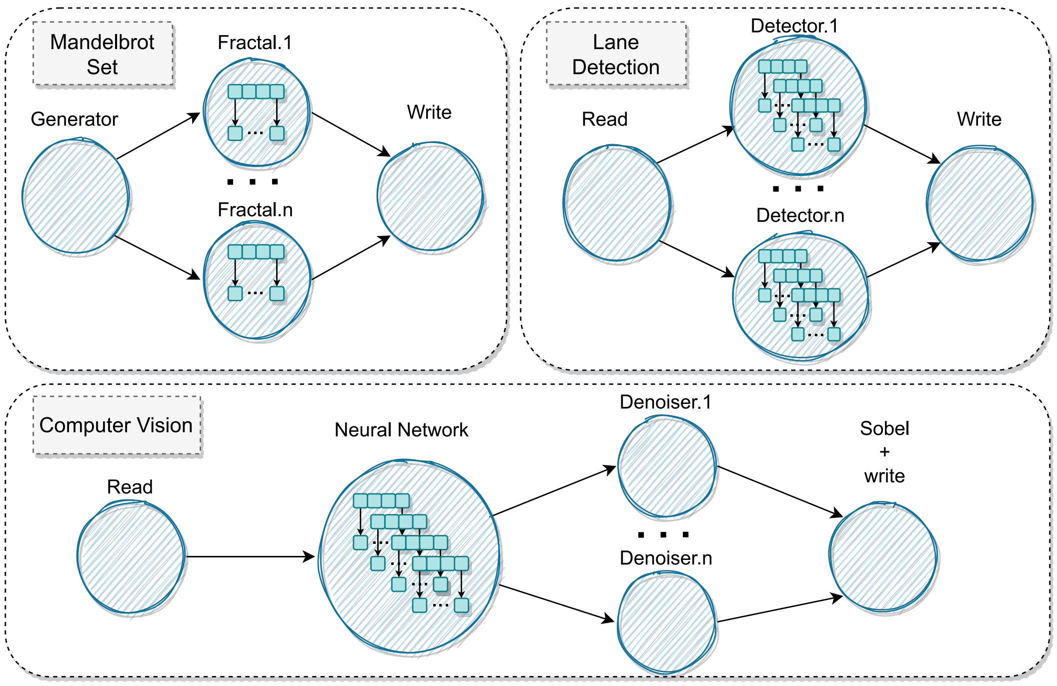


Fig. 5. The flow-graph of the stream processing applications.

4.2.1. Mandelbrot Set

Mandelbrot Set is a mathematical visualization set. It computes a fractal in the complex plane by assigning a color intensity to each pixel according to its position in the plane. This application displays an unbalanced workload. The reason is that points located inside the complex plane within the Mandelbrot set have much more computation than the ones from outside. For the stream processing procedure, our application's input is a 2D matrix dividing stream items as matrix lines. To improve load-balancing, we try to exploit data parallelism by splitting this line between data-parallel workers.

The upper left-hand side part of Fig. 5 shows the resulting Mandelbrot flow-graph. This flow-graph represents the parallelism algorithm automatically generated by SPar taking into account our annotations. In order to achieve that, we inserted a single `Pure` attribute in a relevant data-parallel region. The SPar compiler uses this information to generate the optimized parallel code. In this work, we ensured this is the case.

We parallelized the Mandelbrot Set inserting three SPar language annotations. They are represented in Listing 10. We implemented three stages of computations were the first one is the stream generator that initializes the matrix and distributes items that will be computed by workers using Mandelbrot's fractal routine. Inside the second parallel stage, we used data parallelism to statically divide the stream items and compute them in parallel. In the third and final stage, stream items are concatenated in an ordered fashion and written into a file to obtain the resulting Mandelbrot set image. Concerning original SPar annotations, the only difference in the code is we annotated a single `[[spar::Pure]]` in the data parallelism region.

```
[[spar::ToStream, spar::Input(dim,niter), spar::Output(dim,i)]] {
 // schedule a new line of the 2D matrix
 [[spar::Stage, spar::Input(dim,niter,i,M), spar::Output(M), spar::
     Replicate()]] {
 [[spar::Pure]]
  // apply fractal stage computation
 }
 [[spar::Stage, spar::Input(dim,i,M)]] {
  // write lines in order
 }
}
```

Listing 10: Mandelbrot Set's parallelization with SPar.

4.2.2. Lane Detection

Lane Detection is a stream processing application that receives images captured by autonomous vehicles and performs a computation to detect the lane boundaries. For that, it uses the Canny edge detector and the Hough transform algorithms. A use case of this application is for improving autonomous vehicle computing systems' performance and precision. In this application, each frame captured by the camera is considered a stream item.

```
[[spar::ToStream, spar::Input(grayImages, fullimgsize)]]
 // schedule the frames captured by a camera
 [[spar::Stage,spar::Input(iFile, numberOfpoints, fullimgsize),spar
     ::Replicate()]] {
  // apply detector stage computation
  [[spar::Pure]] {
   // apply a step of gaussian filter
  }
  [[spar::Pure]] {
   // apply a step of sober filter
  }
  [[spar::Pure]] {
   // apply a step of non—maximum suppression
  }
 }
 [[spar::Stage, spar::Input(iFile)]] {
  // write frames in order
 }
}
```

Listing 11: Lane Detection's parallelization with SPar.

Similar to the Mandelbrot Set, the graph-flow of this application results in a Farm parallel pattern that has three computational stages where the middle one is replicated. This can be seen in the upper right-hand side part of Fig. 5. The input frames are read in the first stage and forwarded to the subsequent detection stage. Then, the computation in this detection stage is performed in parallel. Each worker receives a frame and applies the detection filters. Finally, the application collects these resulting frames containing the lane edges and writes them in order into a file.

Latency is critical in this application. Since the workload may become unbalanced when some frames require more computation than others, we applied the second level of data parallelism in the detection stage. As demonstrated in the flow-graph in Fig. 5 we included three Map parallel patterns in the data-parallel regions of the code. These regions have fine-grained computation and represent less than half of total computation.

4.2.3. Computer Vision

Computer Vision is a synthetic application we created to represent a more complex stream processing application flow. In summary, it is an NN (Neural Network) that trains itself with input images captured from a camera and then applies one denoising and then another sobel filter to the resulting images. The output is then written to disk. The neural network we used[1] is written in C++. The topology of our NN is configured with two hidden layers, each with 100 neurons. In the application, each frame captured by the camera is a stream item of our pipeline. Therefore, this application has a different characteristic in the sense that the NN cannot be replicated using stream parallelism. Each image in the pipeline updates the NN weights. If replicated, two parallel images would cause a race condition by trying to update the same NN weight. However, the denoising and sobel filters can be parallelized using stream features. Therefore, for this application, stream parallelism alone is not sufficient. Combining stream and data parallelism is the best option.

The resulting flow-graph of our application is shown at the bottom of Fig. 5. Our plan for parallelizing this application uses a composition of Pipeline and Farm like strategies. In this case, the first stream stage is sequential where we use 4 `Pure` attributes to parallelize NN internal matrix multiplication computations. Then, we have another stage for denoising filtering where we replicate to increase parallelism. Finally, we apply the sobel filter in the last stage and write the output sequentially since it is a smaller portion of the computation.

```
[[spar::ToStream, spar::Input(video)]] {
 // schedule the frames captured by a camera
 [[spar::Stage, spar::Input(frame), spar::Output(frame)]]
 {
  // apply feed forward
  // apply back propagation
  [[spar::Pure]] {
   // matrix multiplication
  }
  [[spar::Pure]] {
   // matrix multiplication
  }
  [[spar::Pure]] {
   // matrix multiplication
  }
  [[spar::Pure]] {
   // matrix multiplication
  }
 }
 [[spar::Stage, spar::Input(frame), spar::Output(frame), spar::
     Replicate()]] {
  // apply denoiser stage computation
 }
 [[spar::Stage, spar::Input(frame)]] {
  // apply sobel stage computation
  // write frames in order
 }
}
```

Listing 12: Computer Vision's parallelization with SPar.

5. Experiments

Experiments were conducted to assess the efficiency of the new compiler algorithm which we implemented as prototype in SPar. From now on, we refer to this new version as SPar+. We used stream processing applications from different domains with the goal of providing information on which class of applications may benefit, or not, from the composition of stream and data parallelism.

Section 5.1 briefly presents our methodology and the environment employed in our tests. The tests are divided in two parts. In Section 5.2, we evaluate the new compiler algorithm performance when automatically generating parallel code for data parallelism applications. In Section 5.3, we evaluate the composition of patterns from different parallelism paradigms. Section 5.4 compares our results with previous work and Section 5.5 summarizes our findings.

5.1. Methodology and environment

The experiments were executed on a machine with two Intel(R) Xeon(R) Silver 4210 CPU @ 2.20 GHz, featuring 20 cores and 40 threads. Each hyper-threaded core has 64KB private L1, 1MB private L2 and 13.75MB of L3 shared. The machine has 64 GB of RAM @ 2400 MHz and is equipped with four HDD 3.5 @ 7200rpm using SATA 3.1 with 6.0 Gb/s. The kernel was Linux 5.4.0-59-generic and the OS Ubuntu 20.04.1 LTS. We used GCC 9.3.0 with `-O3` flag enabled. The FastFlow version was v3.0.0.

The tests were executed from the degree of parallelism 1 up to 40 (maximum degree). The degree of parallelism may not represent the actual active thread count in the system. Rather, SPar spawns dedicated threads for each parallel stage according to the specified degree of parallelism. The execution was repeated 5 times and the graphs represent the average value. The standard deviation was plotted using error bars and may not be visible when the value is negligible. We kept the repetition number low because results were close and the standard deviation is also low. To guarantee the correctness of the parallel versions, we compared the hash value of the output with the sequential version.

5.2. Data parallelism

This section introduces our experiments regarding the performance of the new SPar compiler algorithm when automatically generating data-parallel patterns. These experiments are important to evaluate the efficiency of our code generation strategy. In data parallelism, the NPB is a meaningful benchmark containing important computational characteristics such as intensive memory communications, complex data dependencies, different memory access patterns, and hardware components/sub-systems overload. We executed the applications using NPB's class B (parameters available on website[2]).

The graphs in Fig. 6 summarizes our results. In these graphs, the x axis shows the degree of parallelism while the y axis shows the total execution time in seconds using a logarithmic scale 2. Moreover, the $y2$ axis shows the normalized difference between SPar+ and FastFlow. We used FastFlow's execution time as baseline, meaning that positive values stand for SPar+ being slower than FastFlow while negative values vice-versa. To better assess the difference between the versions, we colored the bars using red and green. They indicate the obtained p-value from our statistical analysis [28]. Before executing the tests, we applied a homogeneity test to evaluate if the samples (5 repetitions) are in a normal distribution. This was done using the Shapiro–Wilk test, in which the results indicate the best hypothesis test to execute (parametric or nonparametric). If samples were in a normal distribution, we executed the paired T-test. Otherwise, we used Wilcoxon test. In our statistical analysis, the null hypothesis (H0) stands for SPar+ equal to FastFlow. To reject H0 and assume H1 (SPar+ is significant different from FastFlow), the p-value must be less than 0.05. When this happens and H0 is rejected, we colored them in red. Otherwise, when H0 is not rejected we colored them in green.

We compare the original SPar and the new SPar+ proposed in this work against handwritten and manually optimized FastFlow versions obtained from past work [26,27]. One of our goals is minimizing overhead compared to the manually optimized FastFlow code. Since SPar

[1] https://github.com/ralampay/ann

[2] https://www.nas.nasa.gov/publications/npb_problem_sizes.html

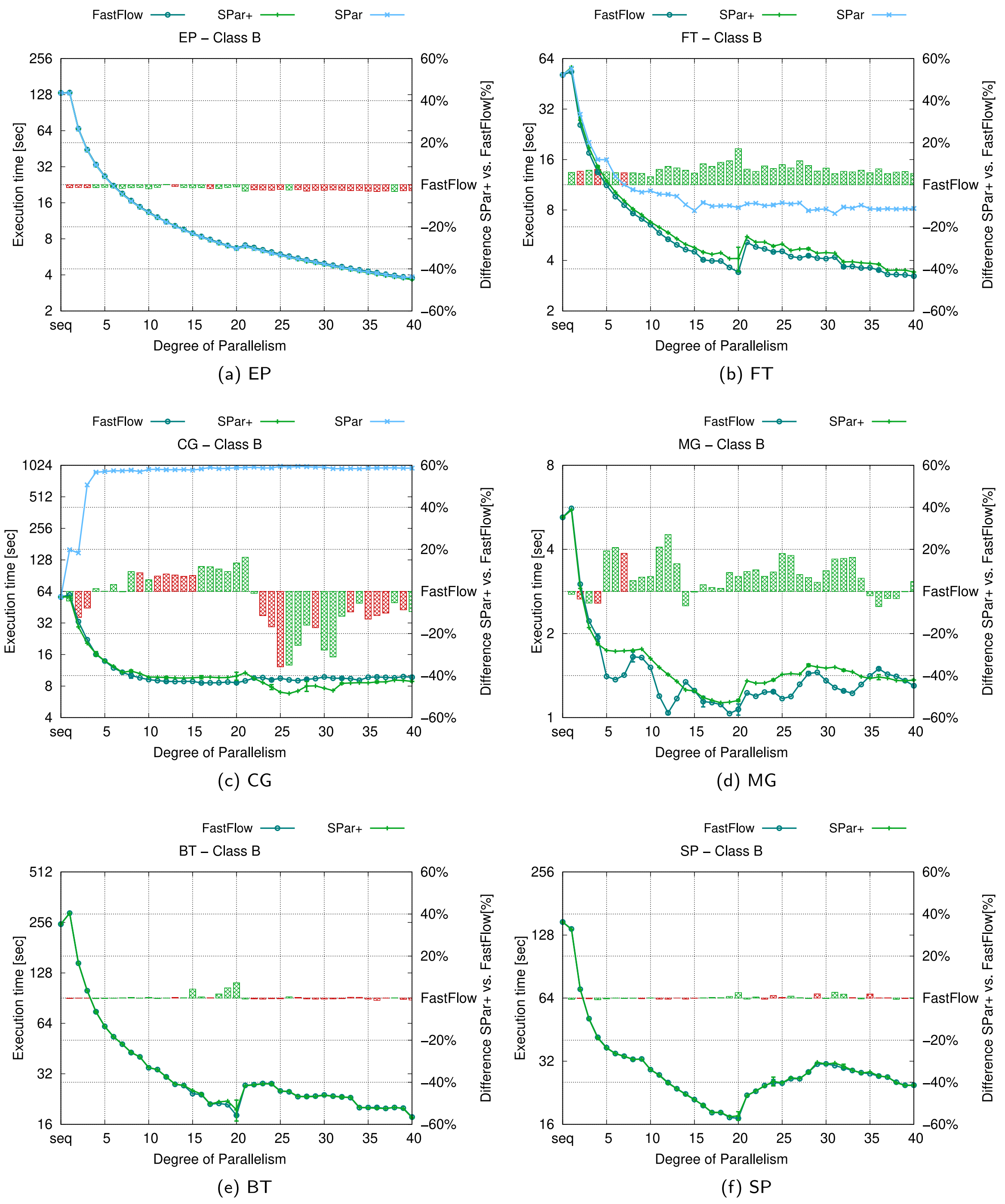


Fig. 6. NPB with handwritten parallelizations [27] vs. automatic parallelizations using SPar and SPar+. Lines represent the execution time while bars represent the normalized difference between SPar+ and FastFlow. Bar's colors show if this difference is significant from the statistical standpoint under 95% of reliability.

generates FastFlow parallel code, an optimal outcome is achieving the same level of performance while providing higher-level programming abstractions. Also, in the experiments we assess the performance of SPar in complex data-parallel applications. Although SPar only generates stream parallelism, in some applications it can achieve similar performance to data-parallel approaches [12]. This is exactly what happens in Fig. 6(a) while others showed a poor performance outcome.

In graphic 6(a), all the results are equivalent since EP is an embarrassingly parallel application containing a single parallel loop. However, there is already a slight advantage when using data-parallel patterns over stream patterns. At the highest degree of parallelism (40 threads), SPar+ is 5.1% faster than SPar using the same annotations in the code. The main explanation is that the stream patterns use an extra thread for scheduling. Therefore, at the highest degree of parallelism, there are 41 threads (40 parallel workers + 1 scheduler) disputing resources.

In graphics 6(b) and 6(c), it becomes clear that the stream parallelism generated by SPar is inefficient for these applications. The main problem is that the scheduler is a bottleneck since it has to orchestrate a fine-grained workload. The tasks perform low intensity computations which means working threads finish very quickly and become idle until a new task arrives from the scheduler. The execution time increases with higher degrees of parallelism since more threads are idly waiting to receive new tasks. The difference between SPar versions is up to 2.4x in FT and up to 108.4x in CG. It is worth noting that in all graphs the execution time increases in the transition from the degree of parallelism 20 to 21 due to the hyper-threading, which introduces workload balancing issues.

We were not able to implement the other three applications with SPar for the reason we explained in Section 4.1. On the other hand, the new language and compiler algorithm increased SPar+ expressiveness and flexibility and we were able to resolve those limitations while enabling parallelism for all applications. Added results are illustrated in graphics 6(d), 6(e) and 6(f), and revealed that SPar+ can achieve similar performance to handwritten FastFlow parallelizations. The major differences between SPar+ and FastFlow are in FT, CG, and MG. In CG, FastFlow uses dynamic scheduling, whose optimal configuration was obtained through experimental tests. SPar+ by default applies static scheduling. However, as explained in [27], the extra overhead required for dynamic scheduling only pays off for bigger workloads.

In MG, the difference is that SPar+ implements one less MapReduce pattern than FastFlow. The reason is that, currently, SPar+ only supports summation reduce operations while MG implements a reduction of type `max`. Therefore, in some degrees of parallelism SPar+ achieved up to 37.1% less performance compared to FastFlow. Similarly, in the FT kernel there is one MapReduce routine that is performed over complex numbers. Since C++ does not represent complex numbers, they implemented it as a `struct` with two `double` data types for representing the real and imaginary values from the imaginary number. SPar's compiler only performs the Reduce operation considering C++ native data types.

The BT and SP pseudo-applications are NPB's most complex data-parallel applications and we parallelized them using multiple Map patterns. Graphics 6(e) and 6(f) presents the results. As can be seen, the performance is equivalent between SPar+ and FastFlow. In these parallelizations, the code generated by SPar+ is identical to FastFlow's parallelism procedure. Slight divergences in execution time or standard deviation may be seen because these pseudo-applications are sensitive to where data is placed in memory. For example, in previous experiments, we were observing situations where the SPar+ code was either worse or better than FastFlow in some degrees of parallelism. This is shown in Fig. 7. We fixed it by removing one memory allocation SPar+ performed at the beginning of the parallel code. As consequence, application data was realigned in memory and the performance issue was solved. New results are the ones illustrated in graphic 6(f). It is worth noting that the decrease in performance for higher degrees of parallelism happens because BT and SP are both bounded to their sequential partial differential equation solvers.

In this section, our goal was to evaluate SPar using data-parallel applications. We expected SPar+ to offer higher levels of abstraction while executing with similar performance to handwritten FastFlow parallel programs. The experiments have revealed that SPar+ achieves similar results concerning handwritten parallelizations, except FT, CG, and MG. In FT and MG SPar+ achieves the lowest performance on most of the parallelism degrees while in CG it is the contrary. However, the statistical test shows that the differences in these applications are not significant from a statistical viewpoint. On the other hand, the graphs of EP, BT, and SP show more occurrences of significant difference although in average they are small (at most 3.18%). We designed SPar+ to be a transparent parallel programming model, therefore it cannot be used to express every low-level parallelism intricacy, especially on a representative benchmark suite like the NPB. We can conclude that the new SPar compiler algorithm is reliable and efficient. We extend our experiments to assess the composition of stream and data parallelism in the subsequent Section.

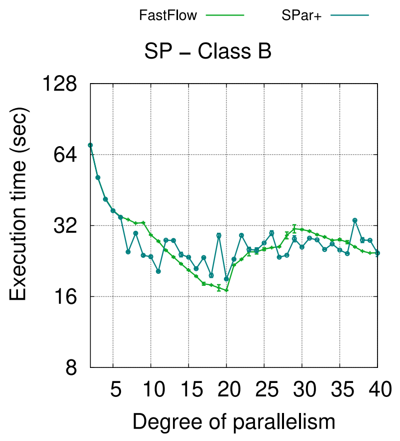


Fig. 7. Performance issue with SPar+ in SP.

5.3. Stream and data parallelism composition

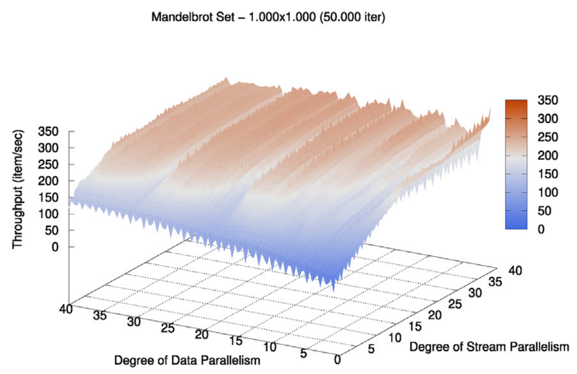
In this section, we assess the performance of different parallelism paradigms. The goal is to investigate the behavior of stream and data parallelism composition and how to compose different parallel patterns. Such effort may bring new insights to this research domain regarding the viability of these compositions. We selected three stream processing applications from different domains. They are briefly described in Section 4.2. These applications contain distinct characteristics we intend to stress to observe the performance of our new compiler algorithm. Mandelbrot Set was selected due to its unbalanced workload. Lane Detection contains routines for real-time image processing. The Computer Vision application implements a heavy computational intensive neural network and other image filters.

The results shown in this section are represented via 3D colored graphs in Fig. 8. A 2D grid (x and y axis) represents every composition of data and stream degrees of parallelism. The results for each metric are simultaneously represented in the z axis and colorized box. From a visual perspective, identifying the best degrees of parallelism is not intuitive. Instead, we want to give the reader an idea of the behavior to identify performance trends. For important data points in the graph, we discuss absolute values in the text.

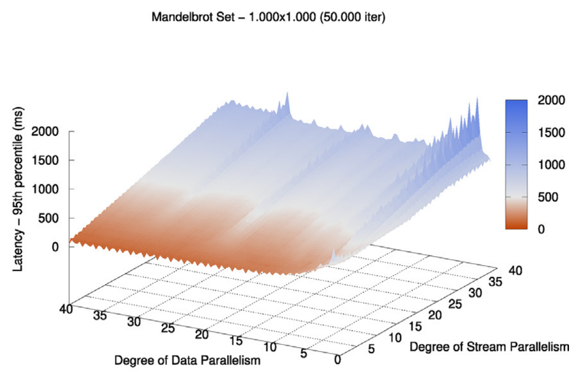
There are two types of graphs: one illustrating the throughput in items per second; another illustrating latency using the 95th percentile latency in milliseconds. The throughput was measured by dividing the total number of items by total execution time. As explained in the methodology Section 5.1, execution time was obtained via the average of multiple executions. Latency was acquired using the kth percentile score from statistics [29]. Therefore, we measure the latency of each streaming item. Then, we order this data set and remove the highest 5% values, which by definition are considered outliers. The remaining latency value sitting on top of the list is the 95th value we used to plot the graphs. We used 5% because it is close to the percentage of cache misses we obtained in our experiments. When data is fetched in main memory, it can severally increase the latency, representing outliers.

5.3.1. Mandelbrot set

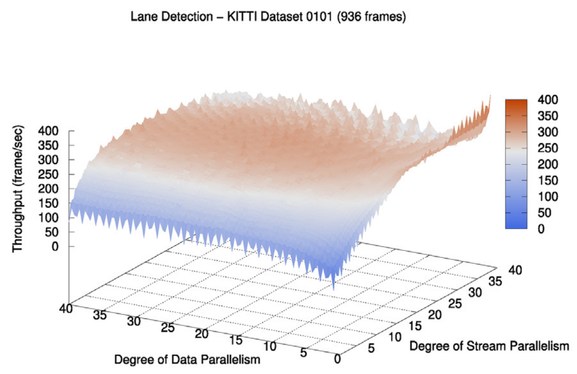
Figs. 8(a) and 8(b) illustrate the results obtained with the Mandelbrot Set application. The first behavior we observe is that stream parallelism scales better than data parallelism. The best throughput is 363 items per second and it is obtained with SPar using the degree 40 of stream parallelism. However, latency is high when the best throughput is achieved, where we captured 696.4 ms of delay in many items. According to trends in the graphics, better latency is enabled with SPar+ and higher degrees of data parallelism. For example, the best



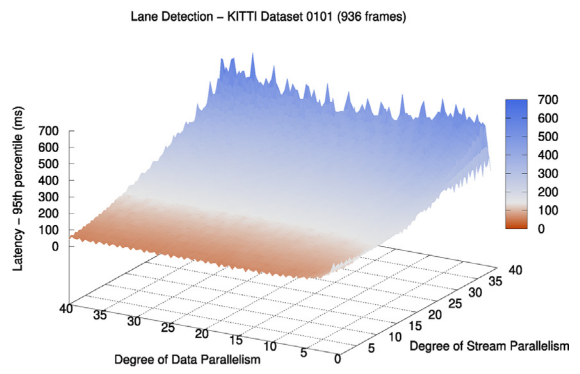
(a) Mandelbrot Set Throughput



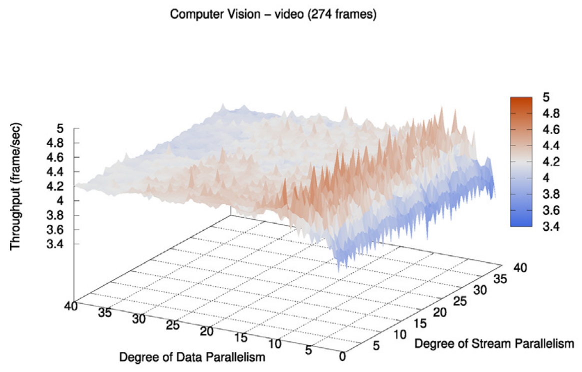
(b) Mandelbrot Set Latency



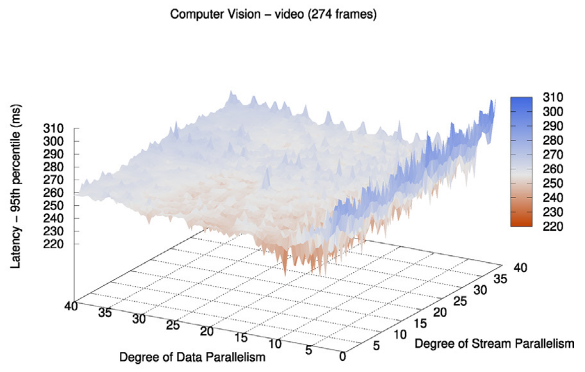
(c) Lane Detection Throughput



(d) Lane Detection Latency



(e) Computer Vision Throughput



(f) Computer Vision Latency

Fig. 8. Performance measures of three stream processing applications parallelized with SPar+. In the graphics, Orange represents the best results and Blue vice-versa. For throughput higher values are better, while for latency lower values are better. SPar uses only stream parallelism and its data are plotted in the first line from right to left, when degree of data parallelism is zero. SPar+ are the remaining lines varying the degree of data parallelism. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

latency of 43.6 ms is captured with SPar+ configuring degree of data parallelism 37 and stream 1.

When using SPar, the best latency is up to an order of magnitude higher with respect to SPar+. The best latency achieved with SPar is 568.8 ms of delay with a throughput of 226. This happens at the degree of stream parallelism 20. In SPar, we also notice that latency is decreasing until reaching degree 20, when it reverses the trend and starts increasing. This result reveal that SPar generating stream-only parallelism cannot efficiently exploit resource usage when dealing with logical threads from the hyper-threading region.

For achieving the best outcome and selecting the desired throughput and latency, application users may choose different degrees of stream and data parallelism. Each parameter offers a trade-off that can be exploited to achieve a balance between these two performance metrics. As revealed by results, along the grid, different degrees of parallelism achieve various performance options that can be selected as parallel configurations.

Furthermore, it is worth highlighting that some degrees of data parallelism introduce severe load balancing constraints, limiting throughput and increasing latency. The highest impact is seen in the degree of data parallelism 7. However, this is a peculiar result. In this case, the overall execution time was impacted by the internal fine-grained data parallelism. The performance deteriorated because data parallelism also suffers from unbalancing workload. However, it suggests that a second nested level of fine-grained parallelism has the ability to balance workloads derived from originally unbalanced applications that exploit coarse-grained stream parallelism.

5.3.2. Lane detection

Figs. 8(c) and 8(d) illustrate the results we obtained with the Lane Detection application. Again, we observe that stream parallelism scales better than data parallelism. This was expected because now our parallel implementation introduces three data-parallel Map patterns that together represent less than half of the total computation. However, this time, the best throughput (367 items per second) was achieved with SPar+ with stream degree 40 and data degree 1. Similarly, SPar achieved a close throughput that is 2.6% worse than SPar+. At the best throughput level, both version achieved equivalent latencies of 228.2 and 230.4 respectively for SPar and SPar+.

The lowest latency (36.4 ms) was captured using SPar+ when configuring stream parallelism with degree 2 and data parallelism with degree 18. On the other hand, the best latency of SPar is 155.2 ms, which is 4.26x higher than the latency achieved by SPar+. Comparable to results observed in Mandelbrot Set, the best latency is achieved increasing data parallelism, while better throughput is achieved increasing stream parallelism. Different configuration options along the grid can be exploited for balancing throughput and latency.

5.3.3. Computer vision

Figs. 8(e) and 8(f) sketch the results obtained with the Computer Vision application. In this application, increasing the degree of stream parallelism does not scale at all. The reason is that replicated course-grained stream processing stages are not significantly intensive routines in this scenario. The most computationally intensive stage is the neural network. However, replicating it could lead to race conditions. This explains why the results using SPar and its stream-only parallelism accomplish low throughput and high latency. The best throughput using SPar is 3.78 items per second with a latency of 282 ms. On the other hand, the best throughput in this application is 5.02 (1.33x higher than SPar) and is achieved when combining stream with degree 8 and data parallelism with degree 9. In this configuration, latency is low as well, where we measured 218 ms of delay. The best overall latency is achieved with SPar+ and is very similar to the one captured in the best throughput. It is 217 ms and was captured with degree 9 of stream parallelism and degree 8 of data.

In both SPar and SPar+ version, the best throughput and latency measured during our experiments are either the same or at least very similar. Therefore, a single parallelism parameters configuration may give the best outcome in this applications. The results may extend to other applications from this domain, but further investigation is necessary.

5.4. Comparison with previous work

This section provides a comparison with respect to our previous work [11]. In previous work, we used an older machine equipped with 24 GB of RAM and two processors Intel(R) Xeon(R) CPU E5-2620 3@2.40 GHz with 6 cores each and support to hyper-threading, adding up to 24 threads. Besides, we conducted our experiments only to obtain the best execution time for each stream processing application. However, stream processing applications may run during long periods of time, possibly to infinity. Therefore, better metrics for this class of application are throughput and latency, which are considered in this work.

The NPB results are equivalent, but show smaller differences between SPar and SPar+ since the machine is slower and runtime overheads are mitigated by total execution time. Regarding our study towards composition of stream and data parallelism, previous results revealed similar trends to this paper's experiments. Our 3D graphics plot the same data patterns. Table 2 shows a comparison between the best execution time of current and previous work. In our previous work, we showed that composition between stream and data parallelism may not always lead to performance gains. In fact, it may degrade performance when the overheads of data-parallel mechanisms outgrow the parallelism gains. This is the case in previous work where SPar+ stops scaling with lower degrees of parallelism. However, in our current work we revealed new insights: although throughput is not always improved, having a nested fine-grained data-parallelism can enhance latency.

Table 2
Comparison with previous work [11].

Apps.	Metrics	Current work		Previous work [11]	
		Spar	Spar+	Spar	Spar+
Mandel-brot Set	Time (s)	2.75	3.28	1.97	2.33
	Std. Dev	0	0.01	0.75	0.75
	Stream par.	40	40	24	20
	Data par.	–	1	–	1
Lane Detection	Time (s)	2.61	2.55	3.94	5.12
	Std. Dev	0.03	0.01	0.03	0.02
	Stream par.	39	40	24	14
	Data par.	–	1	–	8

5.5. Findings summary

In this section, our goal was to analyze the behavior and reveal new insights towards the composition of stream and data parallelism. The experiments showed that there are opportunities to improve parallelism efficiency by exploiting this approach. The most important characteristic we are interested in is improving resource utilization while also improving throughput and latency metrics.

Table 3 summarizes our results by exhibiting the best observed throughput and latency. We highlighted in color the best SPar version for each application. SPar+ is better in 5 out of 6 cases. It achieved the best latency in all situations. Furthermore, SPar can provide higher throughput at the cost of latency. This happens in Mandelbrot where SPar achieves a throughput 19% higher than SPar+. In this case, the unbalanced workload is accentuated because SPar+ uses 2 levels of parallelism. However, in the best latency configuration, SPar+ provides a 15.97x lower latency while the measured throughput is only 3.06x lower than the best configuration from SPar. In other words, it gives up on two thirds of throughput to trade-off a fifteen times lower latency. The same happens to Lane Detection. In Lane detection, one possible configuration gives up on one third of throughput in favor of six times lower latency. Computer Vision using SPar+ achieves the best results: both latency and throughput are improved with respect to SPar ones. SPar+ users can exploit different compositions and select configurations that maximize their needs of throughput and latency.

There are open research questions towards combining different levels of parallelism. However, the literature still lacks solutions that exploit compositions between different parallelism paradigms. We showed that using a single multi-core architecture, stream and data parallelism can be composed to maximize resource usage via workload balancing between available resources. Optionally, different configurations may be used to maximize either latency or throughput. Other approaches for future research contemplate compositions with different architectures. For example, using stream parallelism combined with data parallelism to exploit both highly scalable and distributed cloud environments (stream parallelism) and internally implement fine-grained parallelizations for multi-cores (data parallelism).

Table 3
Summary of best results regarding throughput and latency. We colorized the column that contains the best result between SPar and SPar+ for each stream processing application.

Guideline	Metrics	Mandelbrot Set		Lane Detection		Computer Vision	
		SPar	SPar+	SPar	SPar+	SPar	SPar+
Best Throughput	Throughput (items/sec)	363.35	304.30	358.37	367.55	3.78	5.02
	Latency - 95th per. (ms)	696.4	828.8	228.2	230.4	282	218
	Execution Time (sec)	2.75	3.28	2.61	2.55	72.48	54.61
	Degree of Stream par.	40	40	39	40	1	8
	Degree of Data par.	-	1	-	1	-	9
Best Latency	Latency - 95th per. (ms)	568.8	43.6	155.2	36.4	282	217
	Throughput (items/sec)	226.43	118.80	297.08	164.64	3.78	5.00
	Execution Time (sec)	4.42	8.42	3.15	5.68	72.48	54.75
	Degree of Stream par.	20	1	20	2	1	9
	Degree of Data par.	-	37	-	18	-	8

6. Related work

In the literature, many efforts are being conducted towards increasing the level of parallelism by offering high-level abstractions with negligible performance costs. Table 4 summarizes such related work. We focus on research targeting stream processing and multi-core architectures.

The StreamIt [30] DSL introduces a new language and compiler. Similar to SPar, StreamIt offers a high-level interface for expressing stream parallelism. It also generates parallel code using source-to-source transformations. However, StreamIt requires learning a new syntax and language based on Java while SPar uses C++11 attributes, which are fully recognized and represented in the standard language AST (abstract syntax tree).

GrPPI [31] (Generic Reusable Parallel Pattern Interface) offers a parallel programming abstraction with generic parallel patterns. For that, the programmer only instantiates parallel patterns once and chooses, at compile-time, which runtime GrPPI should generate parallel code. In GrPPI, the programmer is responsible for identifying the best pattern and refactoring the code to implement it manually. In contrast, SPar automatically decides and generates the suitable parallel patterns.

OpenMP is the *de facto* standard for data parallelism in C++ and multi-core architecture. Some researchers notice the difficulties of developing stream processing applications using OpenMP and proposed extensions. However, they provide mechanisms to express data dependencies in the code and enable task parallelism rather than procedures targeting some stream parallelism features.

OmpSs [32] extended OpenMP by proposing a new programming model to annotate parallel code based on pragma directives. Besides, OmpSs supports asynchronous parallelism and heterogeneous programming (GPUs and FPGAs). They have their own compiler and runtime system. In recent versions, OpenMP also supports a *task-based model* taking inspiration from OmpSs' programming model. Developers are equipped with pragmas for creating tasks and linking them according to their data dependencies.

OpenStream [33] also extended OpenMP by offering additional support to task parallelism and Pipelines. This tool is based on pragma compilation directives used by the programmer to annotate dependencies between tasks and providing information about the data flow. Instead of using pragmas, SPar leverages C++ attributes that are available for any compiler that recognizes C++11 and newer. Attributes are fully represented in the C++ grammar. Therefore, they are part of the C++ language and semantics while pragmas are not.

WindFlow [34] introduces a header-only library for leveraging data stream parallelism in multi-core and heterogeneous architectures. The library provides common stream processing operators such as Map, FlatMap, Filter, and others that can be interconnected in a Pipeline or a Directed acyclic graph (DAG). However, the programmer is in charge of identifying convenient operators and refactoring the code to implement them manually. SPar's language does not introduce such parallelism details to the programmer since its compiler generates the parallel code.

PiCo [35] is a DSL that tries to simplify parallelism concerning other Big Data solutions. However, although PiCo proposes a high-level abstraction over FastFlow (similar to SPar's goal), the syntax used by PiCo is still very similar to FastFlow. Differently, SPar clearly distinguishes between low-level parallelism optimizations and high-level abstractions, in which parallelism complexities are hidden from the programmer. SPar's main goal is to help application programmers achieve higher levels of productivity and performance via a high-level programming model.

7. Conclusion

In this paper, we introduced new optimizations to improve stream processing applications. In that, we leverage the idea of internal fine-grained data parallelism exploitation located inside concurrent stages from coarse-grained stream parallelism stages. The optimizations made were new techniques for identifying data-parallel patterns in C++, a new language, semantic analysis, and a set of transformation rules to perform source-to-source parallel code generation. To investigate the feasibility of employing the proposed optimizations in higher-level programming abstractions, we elected the SPar programming model as a use case. We extended SPar by adding two new attributes to its language and implementing a new algorithm for its compiler. With it, SPar is able to generate parallel code more efficiently because it can select parallel patterns between stream patterns (Pipeline and Farm), data patterns (Map and MapReduce), and a composition of them. In addition to that, SPar is now able to express complex data-parallel computations such as presented in NPB.

In the experiments, results evaluating parallel code generation via NPB's data-parallel applications have revealed that SPar's new compiler algorithm can improve performance by up to 108.4x compared to old SPar compiler. Furthermore, the performance of automatic parallel code generation is equivalent to handwritten parallelizations in most cases.

Experiments assessing stream processing applications revealed that lower latency and higher throughput can be achieved when employing a nested level of data parallelism within a coarse-grained stream parallelism compared to a stream-only approach. So, programmers can exploit different configurations of stream and data parallelism to balance between throughput and latency, even when there are more threads than physical cores in multi-cores. In the future, automatic strategies from machine learning or self-adaptive domains can be used to provide the best degrees of parallelism for programmers.

As future work, we plan to conduct experiments on other stream processing applications from different domains to find which class of applications may benefit from this approach. Now that we introduced a new compiler algorithm that generates stream and data parallel code, it becomes simpler to test parallelism composition in other applications. There is still room for improve static analysis to improve parallelism exploitation on `Impure` annotations. Also, we intend to investigate the support for automatic parallel code generation for combining stream and data in different architectures such as clusters of multi-cores, or multi-cores with GPUs.

Table 4
Comparison between related works.

Work	API	Programming language	Runtime systems	Supported architectures	Parallelism exploitation	Language domain
GrPPI [31]	Template library	C++	FastFlow, TBB, OpenMP and, C++ Parallel STL	Multi-cores	Explicit	Stream and Data parallelism
OpenStream [33]	Pragma compilation directives	C/C++	POSIX Threads	Multi-cores	Explicit	Task parallelism
OmpSs [32]	Pragma compilation directives	C/C++	Nanos++	Multi-cores, clusters, and accelerators	Explicit	Task parallelism
WindFlow [34]	Parallel library	C++	FastFlow	Multi-cores and accelerators	Explicit	Stream processing
PiCo [35]	C++ domain specific language	C++	FastFlow	Multi-cores	Explicit	Big Data
StreamIt [30]	External domain specific language	Java	Custom	Multi-cores and clusters	Abstract	Stream processing
SPar [12]	C++ domain specific language	C++	FastFlow, TBB, and OpenMP	Multi-cores, clusters, and accelerators	Abstract	Stream processing

CRediT authorship contribution statement

Júnior Löff: Software, Investigation, Validation, Visualization, Writing – original draft. **Renato B. Hoffmann:** Software, Validation, Writing – original draft. **Dalvan Griebler:** Conceptualization, Project administration, Validation, Writing – review & editing. **Luiz G. Fernandes:** Supervision, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We would like to acknowledge the support of 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 (Nº 19/2551-0001895-9), FAPERGS 10/2020-ARD project SPar4.0 (Nº 21/2551-0000725-7), Universal MCTIC/CNPq call 28/2018 project SParCloud (Nº 437693/2018-0), and MCTIC/CNPq call 25/2020 (Nº 130484/2021-0).

References

[1] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Torquati, Fastflow: high-level and efficient streaming on multi-core, in: Programming Multi-Core and Many-Core Computing Systems, Parallel and Distributed Computing, 2017.
[2] M. Cole, Algorithmic skeletons: Structured management of parallel computation, 1989.
[3] D. Griebler, L.G. Fernandes, Towards distributed parallel programming support for the spar DSL, in: Parallel Computing Is Everywhere, Proceedings of the International Conference on Parallel Computing, ParCo '17, IOS Press, Bologna, Italy, 2017, pp. 563–572, http://dx.doi.org/10.3233/978-1-61499-843-3-563.
[4] T. Mattson, B. Sanders, B. Massingill, Patterns for Parallel Programming, first ed., Addison-Wesley Professional, 2004.
[5] L. Dagum, R. Menon, OpenMP: An industry standard API for shared-memory programming, IEEE Comput. Sci. Eng. 5 (1) (1998) 46–55.
[6] W. Gropp, W.D. Gropp, E. Lusk, A. Skjellum, A.D.F.E.E. Lusk, Using MPI: Portable Parallel Programming with the Message-Passing Interface, Vol. 1, MIT Press, 1999.
[7] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, K. Tzoumas, Apache flink: Stream and batch processing in a single engine, Bull. IEEE Comput. Soc. Tech. Committee Data Eng. 36 (4) (2015).
[8] J. Reinders, Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism, " O'Reilly Media, Inc.", 2007.
[9] Apache Software Foundation, Apache Beam, URL: https://beam.apache.org.
[10] D. Griebler, M. Danelutto, M. Torquati, L.G. Fernandes, SPar: A DSL for high-level and productive stream parallelism, Parallel Process. Lett. 27 (01) (2017) 1740005, http://dx.doi.org/10.1142/S0129626417400059.
[11] J. Löff, R.B. Hoffmann, D. Griebler, L.G. Fernandes, High-level stream and data parallelism in C++ for multi-cores, in: XXV Brazilian Symposium on Programming Languages, SBLP, SBLP '21, ACM, Joinville, Brazil, 2021.
[12] D. Griebler, Domain-Specific Language & Support Tool for High-Level Stream Parallelism (Ph.D. thesis), Faculdade de Informática - PPGCC - PUCRS, Porto Alegre, Brazil, 2016, URL: http://tede2.pucrs.br/tede2/handle/tede/6776.
[13] ISO/IEC-14882:2011, Information Technology - Programming Languages - C++, Technical Report, International Standard Organization, Geneva, Switzerland, 2011, URL: https://www.iso.org/standard/50372.html.
[14] D. Griebler, R.B. Hoffmann, M. Danelutto, L.G. Fernandes, Higher-level parallelism abstractions for video applications with spar, in: Parallel Computing Is Everywhere, Proceedings of the International Conference on Parallel Computing, ParCo '17, IOS Press, Bologna, Italy, 2017, pp. 698–707, http://dx.doi.org/10.3233/978-1-61499-843-3-698.
[15] D. Griebler, R.B. Hoffmann, J. Loff, M. Danelutto, L.G. Fernandes, High-level and efficient stream parallelism on multi-core systems with spar for data compression applications, in: XVIII Simpósio Em Sistemas Computacionais De Alto Desempenho, SBC, Campinas, SP, Brasil, 2017, pp. 16–27, URL: https://gmap.pucrs.br/dalvan/papers/2017/CR_WSCAD_2017.pdf.
[16] D. Griebler, R.B. Hoffmann, M. Danelutto, L.G. Fernandes, High-level and productive stream parallelism for dedup, ferret, and Bzip2, Int. J. Parallel Program. 47 (1) (2018) 253–271, http://dx.doi.org/10.1007/s10766-018-0558-x.
[17] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Torquati, Fastflow: High-level and efficient streaming on multicore, in: Programming Multi-Core and Many-Core Computing Systems, John Wiley & Sons, Ltd, 2017, pp. 261–280, http://dx.doi.org/10.1002/9781119332015.ch13.
[18] OpenMP ARB, Openmp application program interface version 5.0, 2018, URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf.
[19] M. Voss, R. Asenjo, J. Reinders, Pro TBB: C++ Parallel Programming with Threading Building Blocks, first ed., A Press, USA, 2019.
[20] R.B. Hoffmann, J. Löff, D. Griebler, L.G. Fernandes, Openmp as runtime for providing high-level stream parallelism on multi-cores, J. Supercomput. (2022) 1–22.
[21] R.B. Hoffmann, D. Griebler, M. Danelutto, L.G. Fernandes, Stream parallelism annotations for multi-core frameworks, in: XXIV Brazilian Symposium on Programming Languages, SBLP, SBLP '20, ACM, Natal, Brazil, 2020, pp. 48–55, http://dx.doi.org/10.1145/3427081.3427088.
[22] S. Prema, R. Nasre, R. Jehadeesan, B.K. Panigrahi, A study on popular auto-parallelization frameworks, CCPE 31 (17) (2019).
[23] ISO/IEC 14882:2017, ISO/IEC 14882:2017 - Programming Languages – C++, 2000, International Organization for Standardization, Geneva, Switzerland, 2017.
[24] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, S. Weeratunga, The NAS Parallel Benchmarks, Technical Report, NASA Ames Research Center, Moffett Field, CA - USA, 1994.
[25] H.-Q. Jin, M. Frumkin, J. Yan, The OpenMP Implementation of NAS Parallel Benchmarks and its Performance, Technical Report, NASA Ames Research Center, Moffett Field, CA - USA, 1999.
[26] D. Griebler, J. Loff, G. Mencagli, M. Danelutto, L.G. Fernandes, Efficient NAS benchmark kernels with C++ parallel programming, in: 26th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP, PDP '18, IEEE, Cambridge, UK, 2018, pp. 733–740, http://dx.doi.org/10.1109/PDP2018.2018.00120.

[27] J. Löff, D. Griebler, G. Mencagli, G. Araujo, M. Torquati, M. Danelutto, L.G. Fernandes, The NAS parallel benchmarks for evaluating C++ parallel programming frameworks on shared-memory architectures, Future Gener. Comput. Syst. (2021).
[28] D. Taeger, S. Kuhnt, Statistical Hypothesis Testing with SAS and R, first ed., Wiley Publishing, 2014.
[29] X. Dimitropoulos, P. Hurley, A. Kind, M.P. Stoecklin, On the 95-percentile billing method, in: International Conference on Passive and Active Network Measurement, Springer, 2009, pp. 207–216.
[30] W. Thies, M. Karczmarek, S. Amarasinghe, Streamit: A language for streaming applications, in: R.N. Horspool (Ed.), Compiler Construction, Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 179–196.
[31] D. del Rio Astorga, M.F. Dolz, J. Fernández, J.D. García, A generic parallel pattern interface for stream and data processing, Concurr. Comput.: Pract. Exper. 29 (24) (2017) e4175, e4175 cpe.4175.
[32] A. Duran, E. Ayguadé, R.M. Badia, J. Labarta, L. Martinell, X. Martorell, J. Planas, OmpSs: A proposal for programming heterogeneous multi-core architectures, PPL 21 (2) (2011) 173–193.
[33] A. Pop, A. Cohen, OpenStream: Expressiveness and data-flow compilation of OpenMP streaming programs, ACM Trans. Archit. Code Optim. 9 (4) (2013) http://dx.doi.org/10.1145/2400682.2400712.
[34] G. Mencagli, M. Torquati, D. Griebler, M. Danelutto, L.G.L. Fernandes, Raising the parallel abstraction level for streaming analytics applications, IEEE Access 7 (2019) 131944–131961.
[35] C. Misale, M. Drocco, G. Tremblay, A.R. Martinelli, M. Aldinucci, PiCo: High-performance data analytics pipelines in modern C++, Future Gener. Comput. Syst. 87 (2018) 392–403.

Júnior Löff is an M.Sc student in Computer Science at the Pontifical Catholic University of Rio Grande do Sul (PUCRS), and research member of the Parallel Applications Modeling Group (GMAP) at PUCRS. He received his B.Sc Degree in Computer Engineering from PUCRS in 2020. His research interests include: Parallel and distributed systems, high-performance applications modeling, and hardware/software co-design.

Renato B. Hoffmann is a M.Sc student in Computer Science at the Pontifical Catholic University of Rio Grande do Sul (PUCRS), and research member of the Parallel Applications Modeling Group (GMAP) at PUCRS. He received his B.Sc Degree in Computer Engineering from PUCRS in 2020. His research interests include: High performance computing, parallel programming, parallel architectures, and high-performance algorithms.

Dalvan Griebler is an Associate Professor at the Pontifical Catholic University of Rio Grande do Sul (PUCRS) and research coordinator of the Parallel Application Modeling Group (GMAP). Also, he is an Associate Professor at Três de Maio Faculty (Setrem) and head of the Laboratory of Advanced Research on Cloud Computing (LARCC) at Setrem. He received the Ph.D. in computer science from both PUCRS and University of Pisa in 2016. His main research interests are: parallel and distributed computing, methodologies, languages and libraries for high-level parallel programming; benchmarking; cloud computing; applied data science; and data stream processing.

Luiz G. Fernandes is an Associate Professor of the graduate program in computer science (PPGCC) at the Pontifical Catholic University of Rio Grande do Sul (PUCRS). His primary research interests are Parallel and Distributed Computing, High Performance Applications Modeling, Green Computing and Parallel Programming Interfaces. Dr. Fernandes received his Ph.D. in Computer Science from the Institut National Polytechnique de Grenoble (France) in 2002. He currently leads the Parallel Applications Modeling Group (GMAP) at PUCRS.