Parallel Processing Letters
© World Scientific Publishing Company

SPar: a DSL for High-Level and Productive Stream Parallelism

DALVAN GRIEBLER[†],[*] MARCO DANELUTTO[‡], MASSIMO TORQUATI[‡],
LUIZ GUSTAVO FERNANDES[†]

[†] *Pontifical Catholic University of Rio Grande do Sul (PUCRS), Faculty of Informatics, Computer Science Graduate Program (PPGCC), Parallel Application Modeling Group (GMAP). Av. Ipiranga, 6681 - Building 32 - Porto Alegre - CEP: 90619-900 - Brazil*

[‡] *University of Pisa (UNIPI), Department of Computer Science, Parallel Programming Models Group, Largo Pontecorvo 3, 56127 PISA, Italy*

Received January 2017
Communicated by Guest Editors

ABSTRACT

This paper introduces SPar, an internal C++ Domain-Specific Language (DSL) that supports the development of classic stream parallel applications. The DSL uses standard C++ attributes to introduce annotations tagging the notable components of stream parallel applications: stream sources and stream processing stages. A set of tools process SPar code (C++ annotated code using the SPar attributes) to generate FastFlow C++ code that exploits the stream parallelism denoted by SPar annotations while targeting shared memory multi-core architectures. We outline the main SPar features along with the main implementation techniques and tools. Also, we show the results of experiments assessing the feasibility of the entire approach as well as SPar's performance and expressiveness.

Keywords: Stream Parallelism – High-Level Parallel Programming – Domain-Specific Languages – Parallel Design Patterns – Algorithmic Skeletons – C++11 Attributes

1. Introduction

Stream parallelism may be fruitfully used in a wide range of applications including video and audio processing, graphics, and networking applications. Stream parallel applications may target different kinds of parallel architectures (from mobile devices to desktop, servers, clusters, and even supercomputers) and may represent significant workloads. However, a number of these stream applications are still sequential. Thus, when programmers parallelize them or develop a new one, they are faced with the problems of trade-offs between coding productivity and performance. Unfortunately, the programming frameworks that are currently available for programmers to develop efficient and high performance stream applications significantly increase

[*]Corresponding author email: dalvan.griebler@acad.pucrs.br

the programming effort, because they are too low-level, architecture-dependent, and complex (see Section 2).

To target a better productivity/performance trade-off, we designed a new DSL aimed at naturally representing parallelism in stream-based applications. The idea is to offer a set of attributes — to be used as source code annotations — that may be used to clearly denote all the key components of stream parallel applications while preserving the source code of the program. In general, stream parallel applications compute a sequence of distinct activities (called stages) over a stream of input tasks. Each one of these activities consumes an item from the source input stream and produces a new item on some output stream as a result. The structure of the application can be viewed as a graph of independent activities with explicit dependencies modelled after the communications needed to transmit the data from one stage to another, that is, the internal streams. Also, it enables one to identify situations where it is possible to replicate stateless operations to concurrently process stream items [15, 3].

The design of an internal C++ DSL — in particular of a DSL targeting stream parallelism — is still a challenging task. It requires expertise in multiple areas such as programming languages, software engeneering, compiler architecture, parallel programming, etc. Recently, researchers from Stanford University designed a high-performance and high-level embedded DSLs within the Scala language. They developed the Delite compiler framework to enable DSL designers to quickly and efficiently develop DSLs targeting parallel heterogeneous hardware [14]. Integrated with Scala, Delite provides parallel patterns that can be instantiated by the application's DSL designer without worrying about the details related to parallelism exploitation and underlying heterogeneous hardware. Overall, their research builds on a stack of domain-specific solutions that in principle share similar goals with this work. Yet, our idea is to contribute to the C/C++ community, which is commonly used in a wide range of real world applications. We also propose an internal C++ domain-specific language, but we explicitly target stream parallelism. Our goal is to avoid requiring the parallel programmer to explicitly instantiate patterns, which is required in Delite. Instead, we aim to preserve the source code as much as possible, only requiring the programmer to insert the proper annotations to tag stream parallel features in the original sequential C++ source code.

In the C++ community, similar goals are presented in the Re-engineering and Enabling Performance and poweR of Applications (REPARA)[a], which is a project funded by the EU. REPARA's vision is to help develop new solutions for parallel heterogeneous computing [7] in order to achieve a balance between source code maintainability, energy efficiency, and performance [6]. REPARA differs from our work in many ways, but shares the idea of maintaining the source code by introducing abstract representations of parallelism through annotations [5]. A standard C++11 attribute mechanism is used to introduce skeleton-like code annotations (farm, pipe,

[a]http://repara-project.eu/

map, for, reduce, among others). Attributes are preprocessed to produce efficient parallel code targeting heterogeneous architectures including multi-core, GPU, and FPGA-based accelerators. From the high-level parallelism perspective of REPARA, attributes are interpreted by a re-factoring tool built on top of the eclipse IDE (Integrated Development Environment). The re-factoring tool is responsible for the source-to-source transformations driven by the attributes that generate parallel C++ code with FastFlow calls. In the REPARA methodology, code transformations occur in place and produce code that is seen by the users. As in REPARA, we aim to use standard C++ features. However, our attributes are managed at the compiler level and the source-to-source code transformations we perform are hidden from the users.

Our high-level parallel programming approach relies on language attributes to annotate code with abstract representations of potential parallelism. We argue that other annotation-based models such as OpenMP[b] are conceptually lower-level, because OpenMP users have to express the parallelism and also deal with low-level details, especially when implementing stream parallelism. These kinds of interfaces only achieve good coding productivity in specific cases, *e.g.*, in the parallelization of an independent loop. In addition, despite the fact that all of our work is based on C++, our stream parallel annotations can be moved to other languages provided they support some kind of annotations and the possibility to implement source-to-source transformations.

The main contributions of our work may be summarized as follows:

- A DSL for stream parallel applications that improves the syntax and semantics of our DSL proposed in [9], which targeted a different design tool;
- A compiler that recognizes the DSL C++11 attributes and automatically performs source-to-source transformations, generating efficient parallel FastFlow code;
- A set of transformation rules for our DSL targeting parallel patterns, suitable to re-factor annotated code into parallel FastFlow code;
- A new set of experiments comparing performance and expressiveness achieved using our DSL compared with the state-of-the-art tools.

This paper is organized in five sections. Section 2 discusses related works. Section 3 describes SPar's[c] syntax and semantics, presents the related annotation methodology, and demonstrates its expressiveness and effectiveness to parallelize a concrete streaming application. Section 4 introduces and discusses transformation rules targeting stream parallel patterns, our compiler, and the parallel code produced. Finally, Section 5 discusses the experiment results.

[b]The OpenMP web page is http://openmp.org/
[c]The SPar web page is https://gmap.pucrs.br/spar

2. Related Work

In addition to Delite and REPARA, which have already been discussed, there are other different programming frameworks that provide stream parallel abstractions for the application programmers.

FastFlow[d] is a framework that was created in 2009 by researchers at the University of Pisa and University of Turin in Italy [3]. It provides stream parallel abstractions adopting an algorithmic skeleton perspective. The implementation is built on top of efficient fine grain lock-free communication mechanisms [2, 1]. The FastFlow runtime support has been tested in different applications and has been able to achieve a good trade-off among time-to-market, efficiency, and performance portability on various platforms. We used FastFlow as the target of the source-to-source transformations and to process SPar attributes when generating code for shared memory multi-core systems, because it provides the programmer with high-level abstractions (C++ template classes) modelling different parallel patterns for stream parallelism.

StreamIt is a programming framework for streaming applications developed at the Massachusetts Institute of Technology (MIT)[e] targeting cluster and multi-core systems [15]. StreamIt natively supports stream parallelism by providing abstractions similar to those provided in FastFlow. They mainly diverge in the names. For instance, StreamIt offers three filter interconnection modes: pipeline, splitjoin, and feedback loop [16]. Similar concepts in FastFlow are called pipeline, farm, and feedback parallel patterns. StreamIt provides a lower level of abstraction than SPar and may be considered as a possible target for our source-to-source compilation process.

TBB (Threading Building Blocks) is an Intel tool library for parallel programming [f], providing support for the implementation of high-performance applications in standard C++ and it does not require a special compiler for shared memory systems. It emphasizes scalable and data parallel programming while completely abstracting the concept of threads by providing a concept of task. TBB builds on C++ templates to offer common parallel patterns (map, scan, parallel_for, among others) implemented on top of a work stealing scheduler [13]. TBB also results in a lower level of abstraction exposed to the final application programmer than the one presented by SPar and may be considered a possible target runtime in our scenario.

3. SPar: a DSL for Stream Parallelism

SPar is a C++ internal DSL provided as an annotation-based language that models the main properties of stream parallel applications. Our DSL is implemented using the standard `C++11` attributes mechanism and compiled (source-to-source) to an intermediate code with calls to a high-performance library (FastFlow) to target

[d]http://mc-fastflow.sourceforge.net/
[e]http://groups.csail.mit.edu/cag/streamit
[f]http://threadingbuildingblocks.org

multi-core systems. Indeed, any other framework that provides farm and pipeline parallel patterns could have been used as target runtime for SPar.

The SPar annotation mechanism gives the application developer more power than 'pragma' mechanisms, which are compiler pre-processing directives and not part of the C++ grammar. C++ annotations may be put almost anywhere in a program according to the C++ standard grammar [10]. The standard annotation grammar is general enough to support the customization of new attributes and determine where they are allowed in the source code (*e.g.*, to annotate types, classes, code blocks, etc.). SPar maintains the original syntax definition, but it imposes some restrictions to ensure correct parallel code generation.

The attribute names were inspired by the stream parallelism domain, where usually a sequence of independent activities are used as stages when processing a stream. Each activity may consume and produce something. Moreover, the operations that can compute over different stream elements may be replicated to increase the degree of parallelism. SPar introduces few annotations divided into identifier (ID) and auxiliary (AUX) attributes (more details in Section 3.2):

- `ToStream` and `Stage` ID attributes are used to identify the stream computing regions as well as activities producing a stream or processing stream items as a "filter" in the source code.
- `Input`, `Output`, and `Replicate` AUX attributes specify the I/O behavior and degree of parallelism in SPar stages.

These attributes were especially designed to support a programming style for high-level stream parallelism, enforce coding productivity, and provide the exibility to express parallelism and performance in different ways, in addition to being C++ standard interface compliant. Before starting to describe each one of the attributes in detail, Listing 1 gives an example of a common stream parallel application coded with SPar attributes. In this application, the data type of stream items is "string". The stream processing code region is the loop block. The stream comes from an external source, which is a file. For each iteration, a new stream element is read and a sequence of operations is performed. A similar code may be used if the stream comes from the network or any other external source and the programmer may not know in advance the length of the stream. In this case, the programmer has to explicitly manage the program (stream) termination. This stream operation can be seen in line 4, which checks the end-of-stream condition (the end of the file, in this case).

We added a `ToStream` annotation in front of the `while` loop to denote the fact that it represents the stream processing code region. No `Input` attribute was needed since the stream comes from an external source and each stream item is produced inside the stream region. Also, no `Output` attribute was required because nothing is produced inside the stream region that will be used outside of it. Thus, the stream operations identified through SPar annotations are: 1) read stream element (line 3), 2) check end of the stream (line 4), 3) compute with the stream element (line 6) and 4) write the result to an output source (line 8). However, we just annotated two stages in the code, because the region identified by `ToStream` already constitutes an implicit stage. This code behaves as an assembly line, where read, compute and write stages execute simultaneously.

```
1 [[spar::ToStream]] while(1){
2   std::string stream_element;
3   read_in(stream_element);
4   if(stream_in.eof()) break;
5   [[spar::Stage,spar::Input(stream_element),spar::Output(
      stream_element)]]
6   { compute(stream_element); }
7   [[spar::Stage,spar::Input(stream_element)]]
8   { write_out(stream_element); }
9 }
```

Listing 1: Simple stream computation using SPar to annotate parallelism.

Listing 1 clearly shows the expressiveness and high-level abstractions provided by SPar. SPar does not require rewriting/restructuring sequential code and keeps the vocabulary of the new terms introduced for the annotations close to the application developer (the domain user idiom). The aspects regarding mechanisms and policies such as low-level programming models, hardware-level performance optimizations, scheduling policy implementation, load balancing, data and task level problem decomposition, and parallelism strategies to be used (parallel patterns and algorithmic skeletons) are completely abstracted by SPar annotations. Also, our attributes are flexible enough to annotate the previous code in different ways. For instance, we could add the `Replicate` attribute (see how it works in Section 3.2) on the `Stage` as well as put all the code in a single `Stage` with or without `Replicate`. These two different cases may lead to two alternative parallel implementations. However, not all possible annotations will produce correct parallel code. In this code example, the last `Stage` is state-full and therefore it cannot be replicated, unless the application developer knows how to concurrently manage the stage state and properly modifies the replicated code. In order to help the user to correctly and efficiently annotate stream parallelism with SPar, we designed a step-by-step annotation methodology.

3.1. *A Methodology to Successfully Apply the SPar Annotations*

This section introduces a methodology to annotate stream parallelism using SPar attributes based on five questions that the programmer should answer in order to annotate the sequential source code (Figure 1). The methodology orients the programmer in the annotation process of the application code by requiring him to answer these questions.

The first question to answer is "*Where is the stream region?*" in the sequential code. Usually, a stream region is associated with an "assembly line". In a program, we can often identify and visualize the stream region as the most time consuming

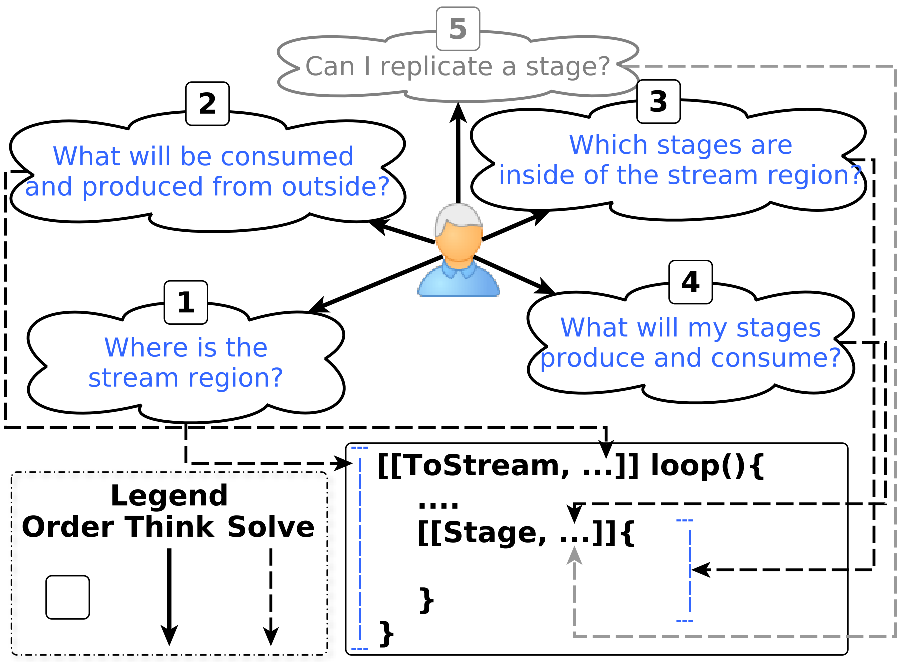

5
Can I replicate a stage?
2
What will be consumed and produced from outside?
3
Which stages are inside of the stream region?
1
Where is the stream region?
4
What will my stages produce and consume?
Legend
Order Think Solve
[[ToStream, ...]] loop(){
....
[[Stage, ...]]{
}
}

Fig. 1: Annotation methodology.

portion of the code. In most cases, the stream computation will be inside a loop, which generates a new stream item per iteration. In all other cases, the stream will come from an external source and the developer should carefully analyze to identify the relevant code section gathering the stream items and computing results out of them. Once the stream region has been identified and properly annotated, we have to look inside the region answering the question "*What does the region consume and produce?*". The idea here is to identify the parameter for the `Input` and `Output` auxiliary attributes that are associated with the stream region.

The third question ("*Which stages are inside of the stream region?*") helps to identify the assembly line's workstations. In the program, they are inside the stream region already annotated based on the answers of the previous questions. To answer the third question, we suggest looking for the sequence of operations and annotating as many stage regions as necessary, respecting SPar's syntax and semantics. The next question to be answered according to our methodology is "*What will my stages produce and consume?*", to specify the `Input` and `Output` attributes.

In the assembly line, we can only assign more workers to a given workstation when the computations of different tasks taking place in the workstation are independent. The fifth question of SPar's methodology, *Can I replicate a stage?*, is aimed at finding out the degree of parallelism associated with the different stages. In the program we may replicate a stage if it can get a new stream item and compute it independently from any other stream item. The developer may use a `Replicate` attribute to improve the performance of any stream region identified as a stage, which is stateless and processes independent items.

We cannot recommend any specific methodology for SPar aimed at help the programmer in the performance optimization process. In fact, each application has a particular structure and the number of stages, stream size, workloads, number of replicas, and stream format should be specified accordingly. Our recommendation is to analyze each case separately and to perform some experiments exploiting the fact that SPar is simple and it ensures the possibility to test different configurations of a stream parallel application with minimal code modifications. Section 3.3 will demonstrate how a real streaming application may be developed using SPar's methodology. In Section 5, we will discuss the performance and productivity experiments of a video OpenCV application (Section 3.3) and other applications that assess our SPar based parallelization methodology.

3.2. *SPar Attributes*

In this section, we describe in detail the syntax, semantics, and behavior of the SPar attributes.

ToStream The `ToStream` attribute is intended to be used to denote that a given C++ program region is going to provide stream parallelism. As dictated by the International Standard [10], the SPar grammar uses and extends the same syntax. A `ToStream` attribute may only be used to annotate a compound statement or iteration statement. SPar requires that an annotated region must contain at least one `Stage` and follow a syntax such as `[[spar::ToStream]]` where additional AUX attributes may possibly be included after the `ToStream`. It is worth pointing out that, as in the standard C++ grammar, the auxiliary attributes possibly associated with the `ToStream` ID attribute are not necessarily ordered. Restrictions are only made for ID attributes to identify a region in the stream parallelism, appearing first in the list. The SPar runtime transforms the `ToStream` region in the first stage of the assembly line. We do not allow a `Replicate` attribute to be added, because this region should perform sequentially to generate, stop, and schedule the stream elements.

Stage The `Stage` attribute is used to annotate a computing phase where operations are computed over the stream items. If we imagine that we are in an assembly line, `Stage` represents a workstation in the production line. Inside a `ToStream` region, SPar can support any number of `Stages`. A `Stage` attribute is written as `[[spar::Stage]]`, and it may be used to annotate a compound or iteration statement. Also, it may be equipped with AUX attributes to further specify the input/output data needed to compute the stage as well as the degree of parallelism to be used to implement the stage. By default, `Stage` and `ToStream` attributes may have arguments (such as those in the AUX attributes), but we are not using them in the current version of SPar.

Input The `Input` attribute represents an another important property of stream parallelism. In SPar, the programmer should use this keyword to express the input data in both ID attribute annotations. Its arguments will be parsed to build the stream of tasks (data items) that will flow inside the `ToStream` or `Stage` region. Using the assembly line example, input denotes the items "consumed" by a given workstation. A typical `Input` attribute will therefore look like: `[[spar::Stage, spar::Input(<var-list>)]]`. It denotes the input data for the `Stage` that has to be taken from the list of variables. When using the `Input` attribute, at least one argument should be given.

Output Programmers should use the `Output` attribute to express the output data stream for both ID attribute annotations. Using the assembly line as example, output is what the ID attribute will produce for the next workstation. A typical `Output` attribute will look like: `[[spar::ToStream, spar::Output(<var-list>)]]`. It denotes the stream produced by the annotated statement will host data from the list of variables. As for the `Input` attribute, when using the `Output` attribute at least one argument should be given.

Replicate The `Replicate` attribute is used to denote the degree of parallelism of a given `Stage` region. We can therefore express this sentence as follows: `[[spar::Stage, spar::Replicate(<integer value>)]]`. No more than a single argument is accepted to represent the number of replicas (workers) in a given `Stage`. This argument can be a literal or variable integer. If no argument is provided, SPar gets the number of workers from the `SPAR_NUM_WORKERS` environment variable. When this attribute is specified in a `Stage` annotation, the SPar runtime generates a given number of workers, each running a replica of the `Stage` region. Consequently, each worker consumes different stream items from the previous computation (either the `ToStream` or `Stage` regions), and possibly produces the computed stream item to the next `Stage` identified via a proper `Output` attribute. By default, a `Stage` region consumes stream items in order but produces them out of order. To maintain the order of the produced stream items, the users only need to specify the `-spar_ordered` flag when compiling a program.

3.3. *Simple SPar Attribute Usage: a Video OpenCV Application*

Video applications represent a classic example of stream parallelism. Video streams can come from different sources (network, local, and camera) and usually programmers do not know how many items will appear on the stream or when the stream will end. Examples of real world video streaming operations include body or face tracking and video filtering. One of the most commonly used C++ libraries in this area is OpenCV[g] and we decided to use it to implement a simple SPar benchmark. Listing 2 presents only the stream region of the application, which was taken from OpenCV usage examples. Instead of reading video frames from the camera, we modified the application to read frames from a video file, applying common video computations on each video frame, to ensure the reproducibility of the experiment.

Our benchmark application extracts a specific RGB channel, applies a Gaussian filter, performs a Weighted screen operation (commonly used in film production), and applies the Sobel filter on each video frame. Before entering into the stream region, the application opens the input and output video files. Inside the infinite loop, the application reads frames from the video file (line 3 in the listing above), tests if it is empty (line 4), performs a sequence of video operations (between line 6 and 16), and writes the results in the output file (line 19). Following SPar's methodology, we can clearly identify the stream parallel region, that is the relevant information needed to insert the `ToStream` annotation. Inside this region, we added two `Stage` regions with the corresponding dependencies that are captured by the `Input` and `Ouput` attributes. As the first `Stage` region may be computed independently over different stream items, we added the `Replicate` attribute to increase the degree of parallelism in this application.

```
 1 [[spar::ToStream, spar::Input(res,channel,src,S)]] for(;;){
 2   total_frames++;
 3   inputVideo >> src;
 4   if (src.empty()) break;
 5   [[spar::Stage, spar::Input(res,channel,src,S), spar::Output(res
      ), spar::Replicate()]]{
 6     vector<Mat> spl;
 7     split(src, spl);
 8     for (int i =0; i < 3; ++i){
 9       if (i != channel){
10         spl[i] = Mat::zeros(S, spl[0].type());
11       }
12     }
13     merge(spl, res);
14     cv::GaussianBlur(res, res, cv::Size(0, 0), 3);
15     cv::addWeighted(res, 1.5, res, -0.5, 0, res);
16     Sobel(res,res,-1,1,0,3);
17   }
18   [[spar::Stage, spar::Input(res)]]
19   { outputVideo << res; }
20 }
```

Listing 2: Video OpenCV using SPar.

[g]http://opencv.org/

4. Implementation

In this section we introduce the most important aspects concerning the implementation of the SPar language. We describe the transformation rules, main features included in our source-to-source compiler, and we discuss the compiler as well as the way the SPar annotations are managed to generate the final parallel code.

4.1. *Transformation Rules*

Our transformation rules process SPar sentences transforming them into stream parallel patterns [12, 11, 4]. These rules are coded in the compiler and target the available low-level (w.r.t. SPar) parallel frameworks. The generated code will be subsequently compiled to produce the actual parallel object code. In our case, we transform the C++ SPar annotated code into C++ with specific calls to the FastFlow framework, targeting the multi-core systems. We first introduce some notations that are useful to express SPar semantics (a complete description of the formalism may be found in [8]):

- T_{id}: is a `ToStream` annotated region associated with an integer variable identifier (id).
- S_{id}: is a `Stage` annotated region associated with an integer variable identifier (id).
- $\Box_{id}$: is a block containing one or more statements associated with an integer identifier (id).
- I_i: denotes an `Input` auxiliary attribute; I_i contains an argument list a_i that represents one or more typed variables.
- O_i: denotes an `Output` auxiliary attribute; O_i contains an argument list a_i that represents one or more typed variables.
- R_n: denotes a `Replicate` auxiliary attribute, where n represents the number of replicas that correspond to an integer variable.
- [[...]]: denotes an annotation that may have a list of attributes.
- {}: denotes the scope of the sentence.

The transformation rules make use of farm and pipeline parallel patterns to introduce parallelism. We can informally define these patterns as follows:

farm(E,W,C) The farm accepts one to three arguments (farm(W), farm(E, W), farm(E, W, C)). The arguments represent the farm task scheduler (also called "emitter", E), scheduling input tasks to the workers, the farm worker (W), computing the tasks, and the farm collector (C) gathering results from the workers. The emitter and the collector may be omitted. In this case, default scheduling and gathering policies are implemented in the farm. Each one of the three elements only accepts a single $\Box_{id}$ as an argument.

pipe(S_1, S_2, ...) The pipe accepts two or more arguments. Each argument may be a $\Box_{id}$ or a $farm()$ and it represents a single stage of the pipeline.

D_0	A *generic stage* ψ is a $\Box$ annotated with S that contains in its attribute list R_n and O_i and therefore requiring a further $\Box$ gathering its results.
D_1	A $\Box$ may appear as a *pipe* stage or as an E or C stage in a $farm$ if its annotation list S does not contain the attribute R_n.
D_2	A $\Box$ with an annotation list S containing an R_n attribute may only appear as a W stage in a farm.
D_3	A T is a *farm* when the first S annotation contains R_n in the attribute list of a maximum two S.
D_4	A T is a *pipe* when the first S does not have R_n in the attribute list or when there are more than two Ss.
D_5	A $farm$ is a stage of *pipe* when D_3 cannot be applied and $\Box$ is annotated with S that contains R_n in the attribute list.

Table 1: Auxiliary definitions for the transformation rules.

The transformation rules from SPar to parallel patterns are based on the auxiliary definitions and rules in Table 1.

First, we introduce three transformation rules where a T region will be transformed directly into a $farm$ (In Table 2, the rules 1, 2, 3). Rule 1 is used to transform $[[T_0]]\{\Box_0, [[S_0, R_n]]\{\Box_1\}\}$ sentence. We can transform it into a $farm$ that has an emitter (which receives $\Box_0$) with worker replicas of $\Box_1$, according to D_1, D_2 and D_3 of Table 1. Rule 2 manages an annotation sentence with $[[T_0]]\{\Box_0, [[S_0, O_i, R_n]]\{\Box_1\}\}$. The relative transformation first produces C that receives ψ (based on D_0). Then, E receives $\Box_0$ (based on D_1) and lastly, $\Box_1$ is assigned to W (based on D_2).

Another possible SPar sentence to transform is the $[[T_0]]\{\Box_0, [[S_0, R_n]]\{\Box_1\}, [[S_1]]\{\Box_2\}\}$. Rule 3 states that we can transform it into a $farm$ (based on D_3) where the E receives $\Box_0$ (based on D_1), W is $\Box_1$ (based on D_2) and C is the $\Box_2$ (based on D_1). The last transformation rule represented in Table 2 operates on the sentence $[[T_0]]\{\Box_0, [[S_0]]\{\Box_1\}\}$, which will be directly transformed into a *pipe* based on Def4 if the first S from the T region does not include any R_n (Rule 4).

The most common set of transformations in SPar are those in Table 3. They are more complex rules that combine farm and pipeline parallel pat-

$[[T_0]]\{\Box_0, [[S_0, R_n]]\{\Box_1\}\} \quad \Rightarrow \quad farm(E(\Box_0), W(\Box_1));$	(1)
$[[T_0]]\{\Box_0, [[S_0, O_i, R_n]]\{\Box_1\}\} \quad \Rightarrow \quad farm(E(\Box_0), W(\Box_1), C(\psi));$	(2)
$[[T_0]]\{\Box_0, [[S_0, R_n]]\{\Box_1\}, [[S_1]]\{\Box_2\}\} \Rightarrow farm(E(\Box_0), W(\Box_1), C(\Box_2));$	(3)
$[[T_0]]\{\Box_0, [[S_0]]\{\Box_1\}\} \quad \Rightarrow \quad pipe(\Box_0, \Box_1);$	(4)

Table 2: Basic transformation rules (only farm and pipeline).

$[[T_0]]\{\Box_0, [[S_0]]\{\Box_1\}, [[S_1, R_n]]\{\Box_2\}\} \Rightarrow pipe(\Box_0, farm(E(\Box_1), W(\Box_2)));$	(5)
$[[T_0]]\{\Box_0, [[S_0]]\{\Box_1\}, [[S_1, O_i, R_n]]\{\Box_2\}\} \Rightarrow pipe(\Box_0, farm(E(\Box_1), W(\Box_2), C(\psi)));$	(6)
$[[T_0]]\{\Box_0, [[S_0]]\{\Box_1\}, [[S_1, R_n]]\{\Box_2\}, [[S_2]]\{\Box_3\}\} \Rightarrow pipe(\Box_0, farm(E(\Box_1), W(\Box_2), C(\Box_3)));$	(7)

Table 3: Complex transformation rules (combinations of farm and pipeline).

terns. For example, in Rule 5 we know that it will be a pipeline because S_0 and $farm$ will become a stage by D_5, because S_1 is followed by R_n. For the $[[T_0]]\{\Box_0, [[S_0]]\{\Box_1\}, [[S_1, O_i, R_n]]\{\Box_2\}\}$ sentence we will use the same definition as Rule 5 due to the S_0 and S_1. Yet, we have to generate ψ to make it an argument of C because there is O_i along with R_n in the last S (based on D_0). The resulting transformation is therefore Rule 6. Rule 7 presents another common sentence in SPar for stream parallelism. The transformation is based on D_4 to become a *pipe*. Then, D_5 makes a $farm$ as an argument of the *pipe*. Thus, $farm$ is based on D_1 and D_2.

As the SPar semantics impose few restrictions, its sentences may be combined in many ways. Even though not all of the possibilities are illustrated, our definitions allow one to implement new and different transformation rules. In this section, we have shown how transformation rules are built, enabeling them to turn SPar sentences into parallel patterns. Therefore, any algorithm considering new transformation rules must meet our definitions and be able to decide among equivalent transformations which is the correct/best one to be applied in the system. In our case, the SPar compiler implements an algorithm to meet all possible transformation rules. Readers may refer to [8] for more details about transformation rule management and usage.

To prototype these rules we made the following assumptions:

(1) Details of parallel patterns like communication and synchronization are already dealt with in the target runtime.
(2) Stream elements are determined according to the O_i and I_i arguments.
(3) Optimizations such as load balancing and scheduling are delegated at the runtime level.

4.2. *The SPar Compiler*

The compiler we designed to handle the SPar DSL uses the CINCLE (Compiler Infrastructure for New C/C++ Language Extensions) tool described in [8]. CINCLE basically provides a parser of the standard C++14 grammar along with an interface to support transformations of the Abstract Syntax Tree (AST) resulting from parsing as well as the generation of source code from the transformed AST. Figure 2 illustrates the overall structure of our compiler: blue boxes represent the SPar specific modules while orange boxes represent the original CINCLE modules

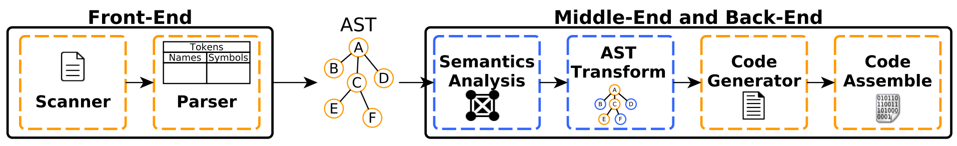


Fig. 2: SPar Compiler.

reused in our SPar compiler. The picture clearly outlines that CINCLE provides a convenient support to implement SPar. The only missing parts are the semantic analysis of the custom attributes and the AST transformations needed to introduce parallelism.

When compiling a program using the SPar compiler[h], the system calls the GNU C++ compiler, right before invoking the scanner, to perform the semantic and syntax analysis of the C++ code. Next, the scanner gets the tokens produced while scanning the original code and delivers them to the parser to create the AST that will be used as input for both the middle-end and back-end. The semantic analysis of SPar annotations can rely on annotation correctness so that AST transformations that enable stream parallelism can be implemented. The final step of the compiler is relative to the generation of the parallel code, directly represented in the AST. Subsequently, the GCC compiler is called to produce the binary code.

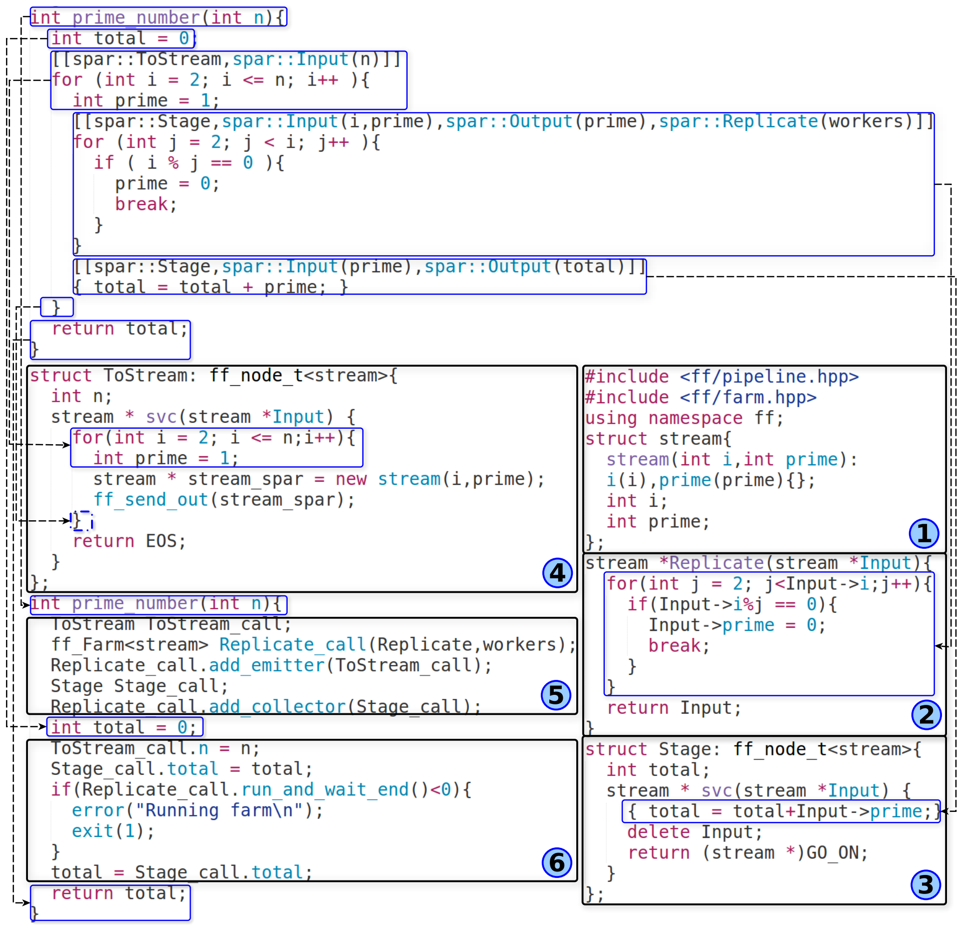
4.3. *Source-to-Source Transformations*

To illustrate how the SPar source-to-source transformations work, we consider a simple application computing prime numbers. Figure 3 uses this application to illustrate the transformations in six steps and maps the original code into the code generated by the SPar compiler. At the top of the figure we put the annotated code using SPar attributes with blocks properly labeled to demonstrate the SPar transformation steps.

First of all, the compiler algorithm starts analyzing the SPar AST, looking for the input and output dependencies so that the data structure inside of the ① step block can be built. The input and output specifications are processed to identify each stream element. Then, pieces of the source code and annotation blocks are transformed according to the transformation rule 3. We first produce the first stage and subsequently the second stage. It is worth pointing out that inside blocks ② and ③ we must properly manage data, taking care of the associated input and output dependencies. Then, we build block ④ that will be used as the emitter for the stream region, taking care of managing when to send the stream elements as well as the end-of-stream delivery.

Lastly, right after the function definition, the whole farm structure is initialized (block ⑤). Also, as a result of the transformation in place of the original

[h]The compiler may be download in https://sourceforge.net/projects/spar-dsl-compiler/



```
int prime_number(int n){
 int total = 0;
 [[spar::ToStream,spar::Input(n)]]
 for (int i = 2; i <= n; i++ ){
   int prime = 1;
   [[spar::Stage,spar::Input(i,prime),spar::Output(prime),spar::Replicate(workers)]]
   for (int j = 2; j < i; j++ ){
     if ( i % j == 0 ){
       prime = 0;
       break;
     }
   }
   [[spar::Stage,spar::Input(prime),spar::Output(total)]]
   { total = total + prime; }
 }
 return total;
}
```

```
struct ToStream: ff_node_t<stream>{
  int n;
  stream * svc(stream *Input) {
    for(int i = 2; i <= n;i++){
      int prime = 1;
      stream * stream_spar = new stream(i,prime);
      ff_send_out(stream_spar);
    }
    return EOS;
  }
};
```

```
#include <ff/pipeline.hpp>
#include <ff/farm.hpp>
using namespace ff;
struct stream{
  stream(int i,int prime):
  i(i),prime(prime){};
  int i;
  int prime;
};
```

```
stream *Replicate(stream *Input){
  for(int j = 2; j<Input->i;j++){
    if(Input->i%j == 0){
      Input->prime = 0;
      break;
    }
  }
  return Input;
}
```

```
int prime_number(int n){
  ToStream ToStream_call;
  ff_Farm<stream> Replicate_call(Replicate,workers);
  Replicate_call.add_emitter(ToStream_call);
  Stage Stage_call;
  Replicate_call.add_collector(Stage_call);
  int total = 0;
  ToStream_call.n = n;
  Stage_call.total = total;
  if(Replicate_call.run_and_wait_end()<0){
    error("Running farm\n");
    exit(1);
  }
  total = Stage_call.total;
  return total;
}
```

```
struct Stage: ff_node_t<stream>{
  int total;
  stream * svc(stream *Input) {
    { total = total+Input->prime;}
    delete Input;
    return (stream *)GO_ON;
  }
};
```

Fig. 3: Mapping the transformations to FastFlow generated code.

`ToStream` annotation block, we produce block ⑥ and update input and output values. We also call the FastFlow runtime executing the farm skeleton. Although different transformation rules may be used when processing different applications, the transformation described above outlines all the steps performed in typical situations. FastFlow supports us with an interface providing all the stream parallelism patterns needed to compile SPar, but it does not prevent us from needing to properly deal with data management and other C++ low-level aspects such as pointers. On the other hand, it relieves us from creating algorithms to support task scheduling and stream ordering, which are primitively provided by FastFlow.

5. Experiments

To assess the usability and performance of SPar, we ran different experiments using different kinds of applications, including the already discussed video processing application as well as small kernels such as Mandelbrot set computation, K-means, and a prime number generation[i]. The goal is to show that we are able to generate efficient parallel code in different scenarios without significant performance degradations compared to manual coding. Also, we intend to demonstrate that SPar is a valid high-level and productive DSL alternative, which preserves the sequential source code while other state-of-the-art tools require code rewriting.

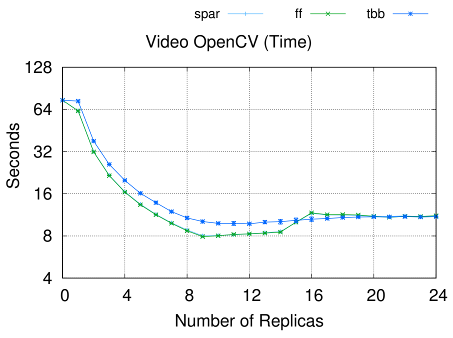
We compare the SPar performance results with those obtained by handwritten code in FastFlow, TBB, and OpenMP (when possible) and evaluate the amount of code needed to implement stream parallelism as a rough measure of the effort required by parallel application programmers. Lastly, we discuss the SPar optimization flags (`spar_ondemand` and `spar_blocking`, see Section 5.3) and their role in performance. All the experiments have been run on a 16 core, 2-way hyperthreading, Intel Sandy Bridge multi-core running Centos Linux (kernel 2.6.32). The compiler used is GNU gcc 5.3.0 (`-O3` optimization turned on). The measures of interest were taken 40 times and averaged in the plots along with their standard deviations plotted using error bars.

5.1. *Performance*

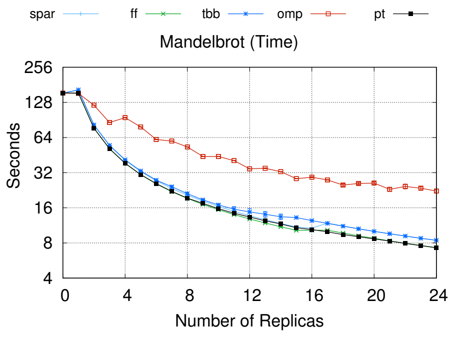
The code for FastFlow, TBB, and OpenMP applications were either available from framework repositories or developed for this specific purpose. Figure 4 shows our results. SPar code scales pretty well up to the number of physical cores available. For the video processing application, the performance is bound by the disk bandwidth (*i.e.* frames read and written boundwith). In all cases, scalability is comparable to that achieved by handwritten code. In most cases, the completion times are close or better than the handwritten applications, mainly in those cases where the degree of parallelism is actually not higher than the amount of available cores.

The high contrast in the OpenMP version for the Mandelbrot set application is due to stream-like computing that requires the use of a critical section, therefore resulting in a significant drop in the performance. In the prime number application, SPar performed identically to other frameworks that used the optimized parallel 'for' loop implementation (*ff-loop* and *tbb-loop*). The dynamic scheduling (*omp-dyn*) in OpenMP and the `spar_ondemand` (*spar-ond*) optimization flag in SPar, greatly influenced these good results. For K-means, we noted that SPar has overhead in this kind of data parallel computation (unlike the results in the prime numbers application).

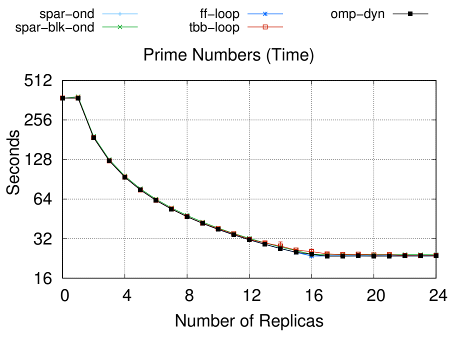
[i]In our case, the data parallelism has been managed by streaming data partitions to farm workers.



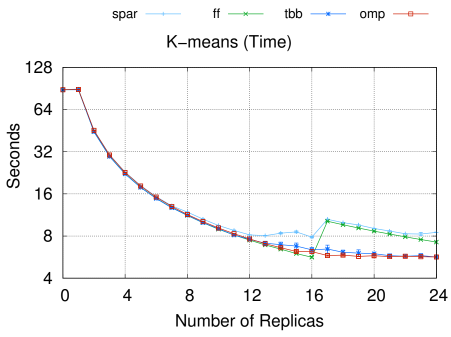
(a) Video OpenCV execution times.



(b) Mandelbrot execution times.



(c) Prime Numbers execution times.



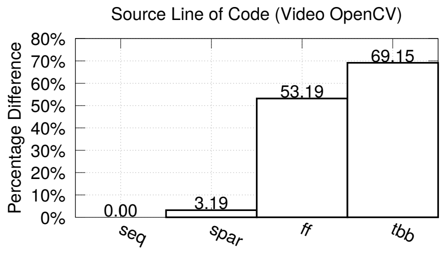
(d) K-Means execution times.

Fig. 4: Performance comparison with hand-written FastFlow(ff), TBB (tbb), OpenMP(omp).

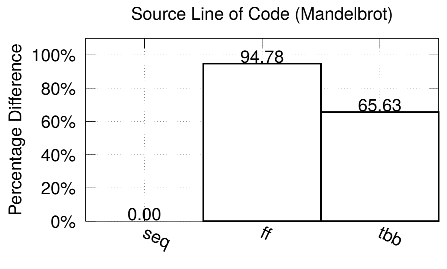
5.2. *Productivity*

We measured the number of lines (SLOC) of code added to the sequential version of the application to implement stream parallelism as a rough measure of the framework productivity figures. Figure 5 shows that SPar requires roughly the same amount of SLOC as OpenMP. SPar requires far fewer additional SLOCs than those needed to implement the same parallel applications using FastFlow or TBB.

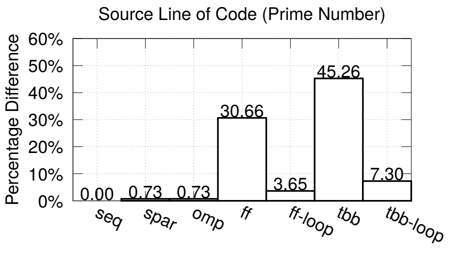
Despite the fact that the same amount of knowledge is needed to insert correct SPar annotations or to write equivalent efficient applications in OpenMP, TBB, or FastFlow, the lower SLOC values typical of SPar guarantee better programmer productivity and thus a potentially smaller chance of introducing errors in the code. Moreover, FastFlow and TBB only demonstrate productivity in the cases where a simple parallel `for` loop pattern can be implemented exploiting the new C++ lambda feature (see Figure 5(c) and 5(d)). In contrast, OpenMP is not as productive as SPar to annotate stream parallelism, as it needs additional code to implement task-based parallel patterns.



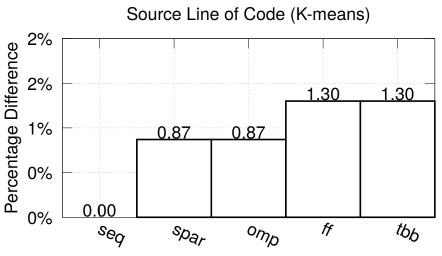
(a) Video OpenCV



(b) Mandelbrot.



(c) Prime Numbers.



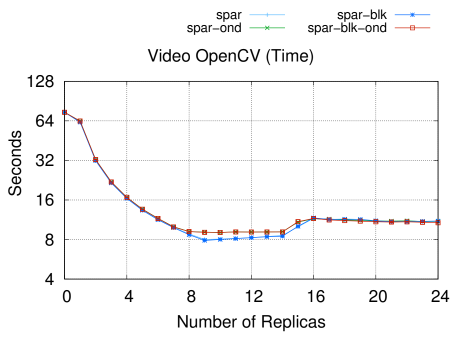
(d) K-Means.

Fig. 5: Coding productivity of FastFlow(ff), TBB (tbb), OpenMP(omp) .

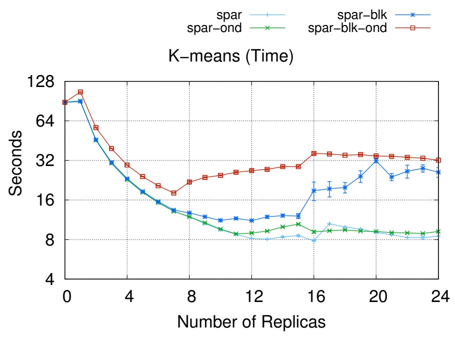
5.3. *Optimizations*

SPar supports some optimizations, such as the possibility to customize the farm scheduling policies or exploit different communication mechanisms supported by its FastFlow based multi-core back-end. Figure 6 shows the different results achieved on our Sandy Bridge architecture, when running the applications using two different option flags: one choosing between blocking/non-blocking communication mechanisms to implement FastFlow communications ("*-blk-*" stands for blocking support, otherwise we are referring to non-blocking support) and the other choosing between on-demand/round-robin scheduling policies for the farm emitter ("*-ond-*" stands for on-demand scheduling, otherwise we used the classic round-robin scheduling policy of tasks to workers in the farm pattern).

When the "grain" of the application is large enough (*e.g.*, in the video processing application) these mechanisms do not impact the overall performance figures. However, when the grain is considerably smaller, SPar achieves the expected results: more efficient mechanisms lead to more efficient applications (*e.g.*, non-blocking communication primitives perform better than blocking ones) when used within SPar. Again, more experiment results are discussed in [8].



(a) Video OpenCV execution times.



(b) K-Means execution times.

Fig. 6: Performance of SPar optimization flags.

6. Conclusions

In this paper we introduced a DSL providing high-level abstractions to introduce stream parallel patterns in C++ applications. We discussed the design of the DSL, its main features, and the main features of the toolchain used to generate efficient and scalable applications on top of FastFlow, and therefore targeting shared memory multi-core architectures.

Currently, we are working to refine the model as well as to fine-tune its toolchain. We are also working to implement new real world applications using SPar, and to complete the implementation of a back-end for the SPar toolchain, targeting a cluster of (multi-core) workstations via MPI. Preliminary results have already been shown in the PhD dissertation [8] that demonstrate how SPar can be used to ensure functional and performance portability across different target architectures, namely shared memory multi-cores and distributed memory clusters of workstations.

Acknowledgements

This work has been partially supported by CAPES, FAPERGS, and EU projects REPARA (No. 609666) and RePhrase (No. 644235).

References

[1] Aldinucci, M., Danelutto, M., Kilpatrick, P., Meneghin, M., and Torquati, M. Accelerating Code on Multi-Cores with FastFlow. In *Euro-Par 2011 Parallel Processing* (Bordeaux, France, September 2011), vol. 6853, Springer, pp. 170–181.

[2] Aldinucci, M., Danelutto, M., Kilpatrick, P., Meneghin, M., and Torquati, M. An Efficient Unbounded Lock-Free Queue for Multi-core Systems. In *Euro-Par 2012 Parallel Processing* (Rhodes Island, Greece, August 2012), vol. 7484 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 662–673.

[3] Aldinucci, M., Danelutto, M., Kilpatrick, P., and Torquati, M. FastFlow: High-Level and Efficient Streaming on Multi-core. In *Programming Multi-core and*

Many-core Computing Systems (March 2014), vol. 1 of *Parallel and Distributed Computing*, Wiley, p. 14.

[4] Cole, M. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, Cambridge, USA, 1989.

[5] Danelutto, M., Garcia, J. D., Sanchez, L. M., Sotomayor, R., and Torquati, M. Introducing Parallelism by using REPARA C++11 Attributes. In *24th Euromicro Inter. Conf. on Parallel, Distributed and Network-Based Processing (PDP)* (February 2016), IEEE, p. 5.

[6] Danelutto, M., Torquati, M., and Kilpatrick, P. A Green Perspective on Structured Parallel Programming. In *23rd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)* (Turku, Finland, March 2015), IEEE, pp. 430–437.

[7] Garca, J. D., Sotomayor, R., Fernndez, J., and Snchez, L. M. Static Partitioning and Mapping of Kernel-Based Applications Over Modern Heterogeneous Architectures. *Simulation Modelling Practice and Theory 58*, 1 (November 2015), 79–94.

[8] Griebler, D. *Domain-Specific Language & Support Tool for High-Level Stream Parallelism.* PhD thesis, Faculdade de Informática - PUCRS, Porto Alegre, Brazil, June 2016.

[9] Griebler, D., Danelutto, M., Torquati, M., and Fernandes, L. G. An Embedded C++ Domain-Specific Language for Stream Parallelism. In *Parallel Computing: On the Road to Exascale, Proceedings of the International Conference on Parallel Computing* (Edinburgh, Scotland, UK, September 2015), ParCo'15, IOS Press, pp. 317–326.

[10] ISO/IEC, . Information Technology - Programming Languages - C++. Tech. rep., International Standard, Geneva, Switzerland, December 2014.

[11] Mattson, T. G., Sanders, B. A., and Massingill, B. L. *Patterns for Parallel Programming.* Addison-Wesley, Boston, USA, 2005.

[12] McCool, M., Robison, A. D., and Reinders, J. *Structured Parallel Programming: Patterns for Efficient Computation.* Morgan Kaufmann, MA, USA, 2012.

[13] Reinders, J. *Intel Threading Building Blocks.* O'Reilly, Sebastopol, CA, USA, 2007.

[14] Sujeeth, A. K., Brown, K. J., Lee, H., Rompf, T., Chafi, H., Odersky, M., and Olukotun, K. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *ACM Transactions on Embedded Computing Systems 13*, 4 (July 2014), 25.

[15] Thies, W., and Amarasinghe, S. An Empirical Characterization of Stream Programs and Its Implications for Language and Compiler Design. In *Intern. Conf. on Parallel Architectures and Compilation Techniques* (Austria, September 2010), PACT '10, ACM, pp. 365–376.

[16] Thies, W., Karczmarek, M., and Amarasinghe, S. P. StreamIt: A Language for Streaming Applications. In *Proceedings of the 11th International Conference on Compiler Construction* (Grenoble, France, April 2002), Springer, pp. 179–196.