Stream Parallelism Annotations for Multi-Core Frameworks

Renato B. Hoffmann*
Dalvan Griebler*
renato.hoffmann@edu.pucrs.br
dalvan.griebler@edu.pucrs.br
School of Technology, PUCRS
Porto Alegre, Brazil

Marco Danelutto
marco.danelutto@unipi.it
Computer Science Department,
University of Pisa
Pisa, Italy

Luiz G. Fernandes
luiz.fernandes@pucrs.br
School of Technology, PUCRS
Porto Alegre, Brazil

ABSTRACT

Data generation, collection, and processing is an important workload of modern computer architectures. Stream or high-intensity data flow applications are commonly employed in extracting and interpreting the information contained in this data. Due to the computational complexity of these applications, high-performance ought to be achieved using parallel computing. Indeed, the efficient exploitation of available parallel resources from the architecture remains a challenging task for the programmers. Techniques and methodologies are required to help shift the efforts from the complexity of parallelism exploitation to specific algorithmic solutions. To tackle this problem, we propose a methodology that provides the developer with a suitable abstraction layer between a clean and effective parallel programming interface targeting different multi-core parallel programming frameworks. We used standard C++ code annotations that may be inserted in the source code by the programmer. Then, a compiler parses C++ code with the annotations and generates calls to the desired parallel runtime API. Our experiments demonstrate the feasibility of our methodology and the performance of the abstraction layer, where the difference is negligible in four applications with respect to the state-of-the-art C++ parallel programming frameworks. Additionally, our methodology allows improving the application performance since the developers can choose the runtime that best performs in their system.

KEYWORDS

programming language, parallel programming, parallel patterns, algorithmic skeletons, C++ annotations, source-to-source code generation, TBB

ACM Reference Format:
Renato B. Hoffmann, Dalvan Griebler, Marco Danelutto, and Luiz G. Fernandes. 2020. Stream Parallelism Annotations for Multi-Core Frameworks. In *24th Brazilian Symposium on Programming Languages (SBLP '20), October 19–20, 2020, Natal, Brazil.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3427081.3427088

*Both authors contributed equally to this research.

SBLP '20, October 19–20, 2020, Natal, Brazil

ACM ISBN 978-1-4503-8943-3/20/09…$15.00
https://doi.org/10.1145/3427081.3427088

1 INTRODUCTION

Stream processing applications are characterized by a flow of data continuously moving through processing filters or stages. Common examples are found in cryptography, audio and video processing, or data compression. These applications are becoming more and more popular today as huge amounts of data are generated through the operation of electronic devices or software systems. However, this is a computationally intensive endeavor for getting time efficient insights/results and requires to exploit parallelism on current multi-core architectures for increasing the performance [2].

Multi-core processors are ubiquitous in machines ranging from mobile devices to data center servers [13]. Consequently, parallel development techniques are required to leverage the architecture's inherent parallelism. These techniques aim at providing mechanisms or algorithms to deal with the complex aspects of parallelism such as load balancing, cache optimization, synchronization, and thread management. Parallel programming abstractions are important to relieve the programmer of the burden of rearranging these low-level concepts such that he can concentrate his efforts on the formulation of efficient domain-specific solutions. One way of providing these abstractions consists of adopting a structured parallel programming approach where the user is equipped with parallel patterns to express parallelism in the source code [15, 16]. To this purpose, Intel TBB (Threading Building Blocks) [18], FastFlow [1], SkePU2 [7], and GrPPI (Generic Parallel Programming Interface) [5] were designed to support parallelism on multi-cores. There are also the non-structured approaches like OpenMP and C++ Threads.

Each parallel programming interface is designed in a different way concerning abstraction and parallelism management. As an example from the scientific community, FastFlow [1] implements its runtime system over non-blocking communication queues with ready to use static and dynamic scheduling as well as offers the possibility to customize the load balancing. To the programmer, it provides building blocks to instantiate ready to use parallel patterns (*e.g.*, Map, Farm, and Pipeline) that can be easily nested to compose other parallel patterns. Intel TBB [18] is an example of the enterprise community. It is less flexible in terms of pattern composability and load balancing compared to FastFlow. TBB is used in production code and does not allow to change its very well optimized work-stealing task scheduler. The challenge here is not judging which is the better framework to always use, instead, it is to find a reasonable way that we can get the best of the two worlds and other parallel programming frameworks.

As can be seen, these frameworks vary among the design principles and may perform better under applications with a certain computational characteristic since they were implemented using different workload scheduling techniques, communication and threads

management, cache optimizations, or mutually-exclusive data access. In the context of stream processing, it has been observed that TBB's functions particularly well with applications where the workload may not be predictable or constant [10, 11, 17]. Also, FastFlow is very well optimized to deal with fine-grained stream computations and low latency requirements [4]. This demonstrates the valuable efforts in the C++ parallel programming frameworks for multi-core systems that have been made. Therefore, we are proposing a methodology that enables the programmer to choose between existing C++ parallel programming frameworks without changing the source code. We are focusing on the stream parallelism domain, using as a base for our research the domain-specific language called SPar [9], which already generates FastFlow as a target framework. Thus, our scientific contributions are summarized as follows:

- A methodology that assists programmers with high-level C++ code annotations for stream parallelism exploitation and aims to support different multi-core's parallel programming frameworks without changing the source code. It extends the research work of SPar [9].
- An implementation of this methodology with SPar to support parallel code generation for the TBB runtime system, which required the design of new transformation rules that were implemented in the compiler tool.
- An experimental performance evaluation of our prototype implementation in different real-world stream processing application types (video, compression, and image), comparing with the state-of-the-art parallelizations. We also discussed the programmability paybacks.

The rest of the paper is organized as follows. Section 2 discusses related works and Section 3 presents the specifics of the annotation methodology and discusses the code generation process. Subsequently, Section 4 demonstrates the experimental results. Finally, Section 5 presents the final remarks.

2 RELATED WORK

During a period of three years (2015 - 2018), researchers involved in RePhrase project worked on software engineering tools that aim at decreasing the complexity of parallel programming in C++ language [19]. The project covered concepts such as parallel pattern composition, parallel programming abstractions, pattern recognition techniques, and semi-automated parallelization tools. Specifically, one of the main goals of this project was supporting parallelism in data-intensive applications. It considers parallelism details such as thread creation, communication, scheduling, data management, etc. The pattern set defined in [3] is an important part of the RePhrase project. In it, the researches present parallel patterns suitable for offline or on-the-fly processing. Typically, offline patterns can be represented by data parallelism whereas on-the-fly processing is a continuous flow of data or data stream. The authors implemented these parallel patterns in the FastFlow library. Using FastFlow, they demonstrated the usability of the patterns by successfully parallelizing pseudo and real-world applications. On the other hand, our approach focuses on abstracting stream parallel computation by means of source code annotations.

In [5], a parallel pattern based interface named GrPPI is presented. From a high-level perspective, their work provides a unified interface for existing frameworks such as FastFlow or TBB. The authors utilized C++ templates in order to generically encapsulate the different patterns provided by other frameworks. Therefore, GrPPI attempts to provide a C++ standard for parallel programming by means of algorithmic skeletons. Comparatively, instead of providing a unified interface, we aim at providing an abstraction layer between the developer and TBB framework. Moreover, the techniques applied to achieve this goal are different from GrPPI. We used standard C++ annotations instead of C++ templates, and we provided ways to compose/annotate programs while enabling the programmer to abstract from the pattern implementation details.

A methodology to automatically identify potential parallel regions in the source code could assist mitigating the cost of developing parallel programs. In [6], they investigated such technique to help detect and annotate parallel patterns in C++ programs. They employ a compile-time approach to reduce the costs of runtime profiling tools. Furthermore, their method supports Farm, Pipeline, and Map patterns, which are suitable for data intensive applications. They use C++ annotations to mark the parallel patterns identified. Overall, the tool was able to recognize a significant number of parallel regions in their test cases. Moreover, the authors state that his approach may miss important parallel regions and add slight performance overheads. Also, they do not provide a compiler tool to parse the annotations inserted in the code. In contrast, we provide a compiler tool that is able to parse, check the semantics, and generate parallel code for different runtimes (FastFlow and TBB)

The work presented in [12] evaluate source-to-source code parallelization compilers. They focus on shared-memory multi-core architectures. Mainly, they consider AutoPar, Par4All, and Cetus among other compilers for automatic parallelization. These are all tools suitable for OpenMP parallelization. Although they function on the outdated version of OpenMP 2.5, the author suggested ways to utilize it with OpenMP 4.5, which is suitable for heterogeneous architectures equipped with CPU and GPGPU. They test and compare these compilers in order to assess their strengths and weaknesses, performance, usability and suggest improvements. There is not a clear evidence of a better compiler, while some perform better than others in different situations. Another aspect is that these compilers generate an AST (Abstract Syntax Tree) from the code, perform the appropriate changes and transform it back to code. Our work in contrast generates code for stream processing applications on multi-core shared memory architectures. For that it uses FastFlow (implemented in [9]) and TBB (added in this work) instead of OpenMP. We also present the transformation rules required to perform the source-to-source code transformations.

3 STREAM PARALLELISM ANNOTATIONS

This Section presents the methodology, syntax and semantics of the code annotations, compiler design and transformation rules.

3.1 Methodology

In a nutshell, the proposed methodology aims to simplify the development of stream parallel applications with different parallel programming runtime using code annotations. Figure 1 illustrates a high-level perspective of the proposed methodology. Firstly, the developer must insert source code annotations to describe a stream

parallel processing region. These annotations were developed to provide nomenclature that feels familiar to the stream processing domain [9]. The next step is to interpret the code annotations and generate the proper parallel implementation. This task is accomplished using CINCLE (A Compiler Infrastructure for New C/C++ Language Extensions) tools [8]. CINCLE is a compiler infrastructure for generating new C/C++ embedded DSLs (Domain Specific Language). It is not a compiler, but a support tool that provides basic features and a simple interface to enable AST (Abstract Syntax Tree) transformations, semantic analysis and source-to-source code generation. The main goal is to simplify the processes of creating high-level parallelism abstractions by using the standard C++11 attribute mechanism. Differently from C++ templates modeling common parallel patterns, the annotations make possible to preserve the original code structure. Furthermore, there is no need for the developer to know about the specific underlying runtime mechanisms and API details. Instead, the parallelism can be expressed by high-level annotations identifying stream parallel computations.

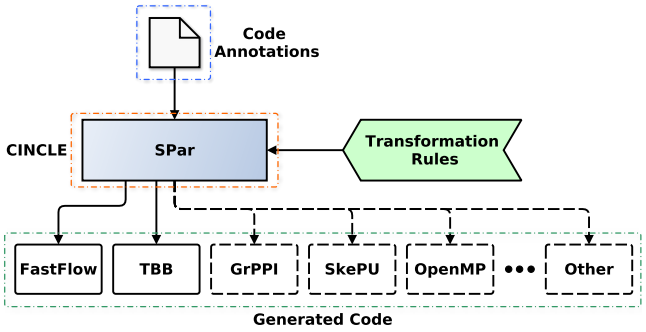

Figure 1: Proposed Methodology.

To put it simply, we use CINCLE's support to transform the code annotations into the proper parallel code of the supported runtime (FastFlow, TBB, GrPPI, or OpenMP) and to generate the final parallel code. Combined with the code annotations, the developer can switch from one target framework to another and explore the different performance results by simply changing a compilation flag. For TBB this flag is `-SPAR_TBB` and, if not used, the default is FastFlow. When switching from one system to another, the same parallel code can be re-compiled to better meet the requirements of that new system. This ability to alternate between runtime environments can also be a option to improve application performance since it can perform better under different circumstances or be more suitable to a different kind of processing. As an example, OpenMP is better suited for data parallelism on multi-core while FastFlow can be better exploited when developing stream applications. As a bonus, if there is no FastFlow available in the new system, the developer can simply change to other supported runtime.

Finally, all of this process is handled with the transformation rules applied during the compilation process. There are transformation rules to describe in which way to transform the code annotations into the proper parallel code. Each underlying runtime has its own independent rule set, which aims to cover the implementation details for the stream domain of applications applied to the runtime API. For this work, FastFlow and TBB have been supported into the transformation rules for the compiler. FastFlow is the default generation of code that was presented in [9] and TBB is first developed in this work. Hence, GrPPI and OpenMP are aimed for targeting in the future. Although OpenMP works with similar idea of annotations, it is only straightforward to annotate and parallelize `for` loops without data dependencies. For stream parallelism, it is quite complex, nonproductive, and requires the implementation of low-level synchronization mechanisms as reported previously [8].

3.2 Syntax and Semantics

In order to simplify the stream parallelism, we used the annotations proposed in [9]. These annotations are standard C++11 attributes mechanisms. They are delimited by double brackets `[[attr-list]]`, where a list of attributes are used. Furthermore, the C++ standard grammar [14] states that annotations can be placed almost anywhere in the code, allowing for the original code structure to be maintained. However, the attribute implementation will determine if it can be used for types, objects, code blocks, etc. With our annotations, the developer is able to express various different stream parallel processing configurations. Furthermore, limitations were imposed to ensure the correctness of the parallel code transformations. We also considered two classes of attributes, ID (identification) and AUX (auxiliary). Therefore, when using this annotations it is mandatory to contain one ID as the first attribute and optionally, a list of AUX (e.g. `[[ID-attr,AUX-attr-list]]`). The whole attribute syntax utilizes 2 ID and 3 AUX, which are briefly described below:

- `ToStream`: is an ID attribute used to identify the stream parallel computation region of the code;
- `Stage`: is an ID attribute used to identify a filtering/processing stage of the stream processing region. It must be utilized inside the scope of a `ToStream`;
- `Input`: is an AUX attribute that specifies which are the data elements consumed by the `ToStream` parallel region or a `Stage`;
- `Output`: is an AUX attribute to specify which data is produced for a subsequent `Stage`. Can only by utilized with `Stage` ID attribute;
- `Replicate`: is an AUX attribute to marks a `Stage` to be executed in parallel to improve performance. It takes as input an integer number or, if left blank, defaults to an environmental variable, which are used to represent the parallelism degree.

```
1  [[spar::ToStream]]
2  for(int num=1;num<1000;num++){
3   int rev = 0;
4   [[spar::Stage,spar::Input(num,rev),spar::Output(num,rev),spar::
        Replicate(4)]]{
5    int t_num = num;
6    while (t_num != 0){
7     rev = rev* 10 + t_num % 10;
8     t_num = t_num / 10;
9   }}
10  [[spar::Stage,spar::Input(num,rev)]]{
11   if (num == rev)
12   std::cout << " The number " + std::to_string(num) + " is a
        palindrome." << std::endl;
13  }
14 }
```

Listing 1: Example for using the code annotations.

Relative to the semantics, a number of rules must be respected in order to ensure the correctness when generating the parallel code. Firstly, every annotation must be placed in front of an iteration or compound statement. This ensures the scope of the parallel computation regions is properly defined. Additionally, the semantic rules do not allow for nested `ToStream` annotations, which means it is not possible to generate a stream flowing inside another stream. However, the number of `Stages` inside a `ToStream` region is not restricted. Both `Input` and `Output` must contain at least one argument. Consumer `Input` and producer `Output` attributes must have matching arguments. Furthermore, `Output` can only be used with a `Stage` attribute, whereas Input can be utilized with both `ToStream` and `Stage`. Finally, the `Replicate` can be solely used with a stateless operator, which means the associated computation stage can not have unchecked critical/atomic code blocks.

With the example in Listing 1, we demonstrate the usability of the annotations in the parallelization of a C++ pseudo-code that discovers and prints palindrome numbers between 1 and 1000. Firstly, `ToStream` is used to mark the parallel computing region as the scope of the `for` iteration statement in line 2. In this case, `ToStream` does not consume data from other parts of the code so no `Input` is needed. Effectively, this means that the stream items will be created and generated directly from `ToStream` region independently from the rest of the code. The code situated between the `ToStream` annotation and the first `Stage` is the generation stage of the stream, which produces the numbers for the following stages. This generation stage is the only code inside a `ToStream` that can be left outside the scope of a `Stage`. In sequence, the computation stage annotated in line 4 is responsible for reversing the number. Here, the stage will consume `num` and `rev` in `Input` from the previous stage, which is the generation one. Then, it will perform its computations and send the data to the following stage, which is specified by the `Output` annotation. Additionally, the `Replicate` attribute guarantees that the stage code will be executed in 4 different threads. In this case, it is safe to replicate the computation, since this is a stateless stage. It does not require to access previous states of the variables. Numbers are processed independently from one another. Lastly, the `Stage` in line 10 consumes `num` and `rev` from `Input`, checks if the number is palindrome and prints it. If the input order should be preserved, a compilation flag has to be used as we explain in Section 3.3.2.

3.3 Source-to-Source Code Generation

In this Section, we present the internal compiler design and transformation rules.

3.3.1 Compiler Design. In this Section, we provide an overview of the low-level details of the methodology. After the source code is properly annotated, the following step is the code generation. Figure 2 illustrates the compilation process, which is supported by CINCLE. The compilation phase starts with a call to GCC compiler to perform a semantic and syntax analysis of the C++ code. Then, the scanner module produces the tokens that will be forwarded to the parser to create the AST (Abstract Syntax Tree). This AST is fully accessible, allowing for complete control of every node. This way, the code transformations can be performed directly on the tree during compilation time. Nodes can be removed or shifted freely, allowing for any possible code transformation required to generate the calls to the abstracted runtime. This is one of the main advantages of CINCLE, since this operations would not be possible just using the GCC compiler.

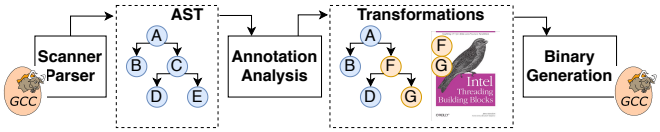

Figure 2: Compilation Flow.

Subsequently, a semantic analysis module will check the correctness of the annotations. Here, the semantic rules previously described are checked and inconsistencies are reported to the developer. It is important to point out that if a stage is annotated as parallel, this step does not check for data races. The stateless nature of the stage is a complete responsibility of the programmer. Later, the code transformation step performs the appropriate transformations directly on the AST. These consist of substituting the annotations nodes in the tree with appropriate calls to abstracted runtime API. This step must be performed with extensive care to the C++ ISO, features as mistakes can easily brake the entire code. The final steps can be expressed as the parallel code generation from the transformed AST. After the end of this process, GCC is called again to produce the final binary.

Inside the compiler, we create separately an AST for the annotations such as depicted in Figure 3 to simplify the implementation of the designed transformation rules after the semantic analysis. This is a simplified view of the actual representation, which is based on the tree generated from the code previously explained in Listing 1. Each attribute is represented as a node in the tree that stores its information about the arguments. The root node is a function definition node (FD) that is followed by a `ToStream` (T) node with an identifier node (ID) of stages. Here, there are two `Stage` (S) each with its own AUX (auxiliary attributes). One of them contains an `Input` (I), `Output` (O) and `Replicate` (R) annotation and the other just an `Input`. Starting from this AST, the compiler performs the transformation according to the rules in the following Section.

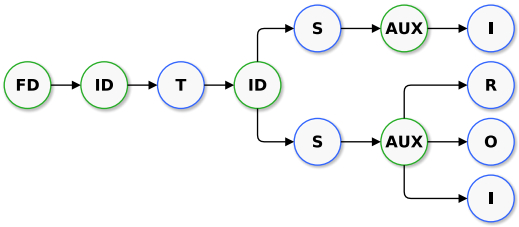

Figure 3: Representation of the SPar annotations in the AST.

3.3.2 Transformation Rules. The goal of the compiler transformation phase is to generate the proper calls to the parallel runtime (FastFlow or TBB) from the annotated code. A set of transformation rules is defined in order to respect the semantics previously described. In order to clarify the rest of this Section content, we

define some terms. A □ 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`), and S (`Stage`). The AUX attributes I (`Input`), and O (`Output`) take a list of arguments a_i. Finally, R (`Replicate`) takes n an integer variable or literal number greater than zero as an argument.

There are two different transformation rule sets, one for FastFlow and another for TBB. Each one of these rule sets aims to cover the requirements for the stream parallelism domain applied to each runtime. We start with FastFlow rule set definition. To express and generate parallel code for FastFlow, we used Pipeline and Farm parallel constructs. These are suitable patterns for stream processing applications. The Pipeline is defined as $pipe(S_1, S_2, ...)$ 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. $pipe(S_1, farm(E, W), S_2)$).

Table 1: Definitions of the transformation rules for FastFlow extracted from [9].

D_0	An additional gatherer stage ψ is generated when the last block of code is annotated with S that contains in its attribute list R_n and O_i.
D_1	A generic block of code may appear as a *pipe* stage, as an E or C stage in a $farm$ if its annotation list S does not contain the attribute R_n. A block of code can be assigned to E when the following block of code is assigned to W. A block of code can be assigned to C when the previous block of code is a W. Otherwise, the block of code is assigned to a *pipe* stage.
D_2	A generic block of code with an annotation list S containing an R_n attribute may appear as a W stage in a $farm$.
D_3	A T is a $farm$ when there is a S in the attribute list that can generate a W due to D_2 where the block of code annotated by T is the E.
D_4	A T is a *pipe* when the first S is not a W due to D_2 or when there are more than two Ss.
D_5	A $farm$ is a stage of *pipe* when D_4 is applied first and mandatory applies D_2 while D_1 is optional for E.

We demonstrated the definitions used to create the transformation rules in Table 1. The priority of the transformations rules follows the growing order. Therefore, when the rules apply, a definition D_i will be chosen where i is the lowest possible. Additionally, in some cases more than one definition D_i may apply to the same annotation. If more than one apply and the lowest i does not generate a direct call to the template, the following i may be used instead. We also exemplify the use of these definitions in Rule 1, which is the code previously described in Listing 1. This refers to the annotations $[[T_0]]\{\square_0, [[S_0, I_0, O_0, R_n]]\{\square_1\}, [[S_1, I_1]]\{\square_2\}\}$. In this case, D_0, D_4, and D_5 do not apply. D_0 is used in rarer cases, where the output of a replicated stage might be used outside of the stream region. Then, the last annotation with $[[S_1, I_1]]$ will be the Farm's collector to conform with definition D_1. The second annotation $[[S_0, I_0, O_0, R_n]]$ will be in conformation with definition D_2, which results in $\square_1$ as the Farm's worker. Lastly, $[[T_0]]$ applies for definition D_3, which generates a $farm$ with $\square_0$ as the Farm's emitter.

$$[[T_0]]\{\square_0, [[S_0, I_0, O_0, R_n]]\{\square_1\}, [[S_1, I_1]]\{\square_2\}\} \Downarrow farm(E(\square_0),\ W(\square_1),\ C(\square_2)) \quad (1)$$

Table 2: Definitions of the transformation rules for TBB.

D_0	An additional gatherer filter ψ is generated when the last block of code is annotated with S that contains in its attribute list R_n and O_i.
D_1	A generic block of code is a sequential filter in a *pipe* if its annotation list S does not contain the attribute R_n.
D_2	A generic block of code with an annotation list S containing an R_n attribute is a parallel filter in a *pipe*.
D_3	A T annotation is transformed into a *pipe* when the attribute list contains at least one S.

To express and generate parallel code for TBB we propose a new rule set. In TBB, the $pipeline(filter(S_0), filter(S_1), ...)$ template was considered, which contains a list of filters with S_i processing stages. Each one of these filters may be *seq* (sequential) or *par* (parallel). The definitions are described in Table 2 and the priority of the transformations rules follows the growing order. Again, more than one definition may apply to the same annotation and a definition D_i will be chosen where i is the lowest possible. Starting with Rule 2, it is for transforming $[[T_0]]\{\square_0, [[S_0, I_0, O_0, R_n]]\{\square_1\}$ sentence. D_0 is not applied in any case of the annotation schema. D_1 matches solely for annotation $[[S_1, I_1]]$, which generates a sequential filter. Then the annotation with $[[S_0, I_0, O_0, R_n]]$ generates a parallel filter according to D_2. The annotation $[[T_0]]$ matches for definition D_3, which generates the *pipeline*, receiving as argument the generated stages from the previous definitions. Regarding to the sentence $[[T_0]]\{\square_0, [[S_0, I_0, O_0]]\{\square_1\}, [[S_1, I_1, R_1]]\{\square_2\}\}$ (Rule 3), D_0 is also not applied. The $[[S_0, I_0, O_0]]$ annotation matches with the definition D_1, generating a sequential filter. Then the $[[S_1, I_1, R_1]]$ annotation matches for D_2 to generate a parallel filter. Lastly, the $[[T_0]]$ annotation matches with D_3, which generates a *pipeline* receiving as argument the stages generated from the previous definitions.

$$[[T_0]]\{\square_0, [[S_0, I_0, O_0, R_n]]\{\square_1\}, [[S_1, I_1]]\{\square_2\}\} \Downarrow pipeline(seq(\square_0), par(\square_1),\ seq(\square_2)) \quad (2)$$

$$[[T_0]]\{\square_0, [[S_0, I_0, O_0]]\{\square_1\}, [[S_1, I_1, R_1]]\{\square_2\}\} \Downarrow pipeline(seq(\square_0), seq(\square_1),\ par(\square_2)) \quad (3)$$

In addition to the previous transformation rules demonstrated, TBB implementation also imposes a few other restrictions. Before

```
class filter_1 : public tbb::filter{
public:
    filter_1(): tbb::filter(tbb::filter::serial_in_order)
    {}
    void* operator1()(void* decoy){
        for ( init-statement; condition ; expression ) {
            statement
            return decoy;
        }
    }
    void* operator2()(void* decoy){
        init-statement
        while ( condition ) {
            statement
            expression
            return decoy;
        }
    }
}
```

```
class filter_1 : public tbb::filter{
public:
    filter_1(): tbb::filter(tbb::filter::serial_in_order)
    { init-statement }
    void* operator1()(void* decoy){
        if ( condition ) {
            statement
            expression
            return decoy;
        }
    }
    void* operator2()(void* decoy){
        while ( condition ) {
            statement
            expression
            return decoy;
        }
    }
}
```

Figure 4: Transformation case in the TBB filter.

that, it is necessary to understand TBB's nature when handling the pipeline flow. In TBB's pipeline, a token is a processing element that can be active during the execution. During the pipeline execution, the tasks will associate themselves with a token and carry it for as many consecutive filters as possible. If a sequential filter is already occupied by another task, the other task will park its token there and look to carry other tokens through other filters. Because of that, every filter needs to operate in a non-continuous manner and each time return a pointer to the task carrying this token.

Effectively, this imposes two restrictions to the first processing filters, generally responsible for the data generation. They are illustrated by 4. The first one defines that no initialization of preprocessing variables is possible in the operator code since it would overwrite the variable every time the operator is executed. The second restriction is that the `for` loop can not be utilized in the first stage as the generation loop. The reason is that the initialization field of the `for` would have the same initialization problem previously described. In addition, the expression field would never execute as the operator would return its value prior to that. These transformations are performed by the compiler directly in the AST and the programmer does not need to consider them when annotating the source code. The remaining transformation are direct calls to TBB's API. We illustrate these situations and the transformations required to solve them in Figure 4. On the left side of the figure, we exemplify two operators, `operator1` with the `for` loop limitation and `operator2` with the initialization situation. On the right, we demonstrate the required transformations to properly accommodate the code in TBB's API.

The maximum number of active tasks during the program execution is defined accordingly to the Replicate argument. This number is also used to define the maximum number of concurrent TBB tokens. Each TBB token is equivalent to one data processing element active throughout the pipeline execution. That is an important aspect for the performance of the application, where a very small number can lead to idle processors waiting for tokens and too high can utilize extra memory. As a solution, we define this number of tokens with the number of active parallel workers times ten. In [11], this was considered to be the overall best performing number of tokens for the tested set of stream applications.

One final remark to be considered in the code generation is the need for re-ordering data elements when a stage is replicated. This is necessary to guarantee the integrity of the output data in some applications as data reaches the final stage in a non-deterministic manner. For SPar, it does not require any code changes to be performed and can be achieved using the compilation flag `-spar_ordered`. In FastFlow's case, it requires to generate code that instantiate a different template class, which is `ofarm` (ordered farm). On the other hand, TBB requires us to generated a different filter type using `serial_in_order` instead of the default `serial_out_of_order` for the stages that are not replicated.

4 EXPERIMENTS

The experiments aimed to assess the performance of the annotated methodology proposed in the previous Sections. We have performed the tests using the Ferret (image similarity search), Bzip2 (compression), Lane Detection (video), and Person Recognition (video) applications. These applications were first presented in [10] and [11], where more low-level details can be found and input files information.

The tests were performed in a machine equipped with 32GB of RAM memory and two processors Intel(R) Xeon(R) CPU E5-2620 v3 2.40GHz (total of 12 physical cores and 24 threads with Hyper-Threading). The operating system was Ubuntu Server 64 bits kernel 4.15.0-88-generic, and GCC 7.4.0. The compilation was also performed with the -O3 optimization flag. Other libraries are Opencv version 2.4.13.6, TBB 2017 (INTERFACE_VERSION 9107), and FastFlow (revision 2.2.0-45). Each plotted value on the graphs is obtained from the arithmetic mean of 10 executions performed for each parallelism degree value ranging from 1 up to 24. Additionally, the standard deviation is also plotted in the form of error bars, which may not be visible in most cases as its mostly negligible. For each one of the applications, we tested SPar generating FastFlow (spar) and SPar generating TBB (spar-tbb). Both these SPar versions used the same annotations and no changes were required to switch between the two. Additionally, we have also compared our annotated implementations with versions utilizing Pthreads (pthread), FastFlow (ff), and TBB (tbb). This way, we can showcase a performance comparison with the generated code and state-of-the-art

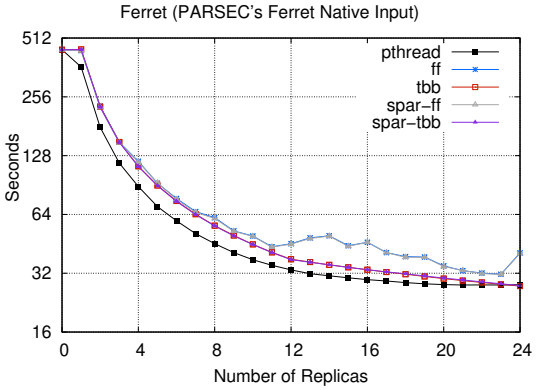

Figure 5: Ferret application.

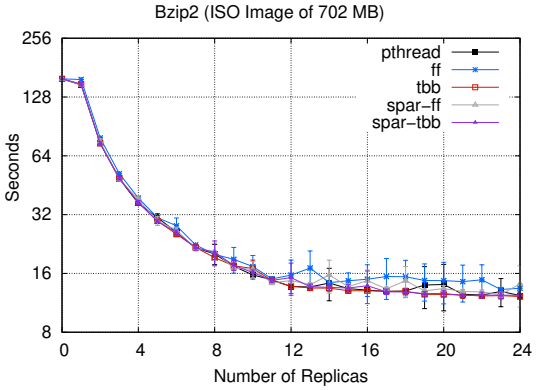

Figure 6: Bzip2 application.

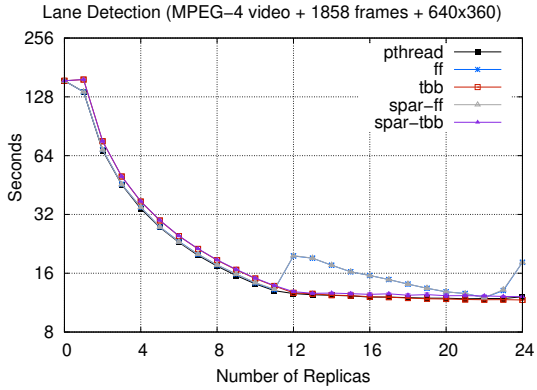

Figure 7: Lane Detection application.

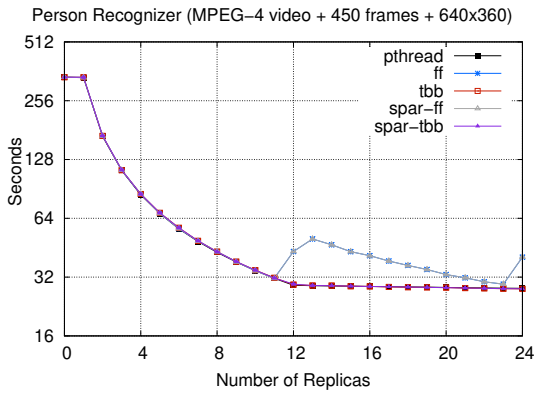

Figure 8: Person Recognition application.

manual implementations. Moreover, we guaranteed the correctness of the parallel versions by comparing the hash value of the output with the sequential version.

The graphs in Figures 5, 6, 7, and 8 showcase the performance results achieved. They express the execution times of each application in a logarithmic scale with base 2. Moreover, in all of the resulting performance graphs, the 0 value of parallelism degree is the sequential version of the program. As another important aspect, the number of replicas in the x axis does not necessarily represent the number of active threads in the system. Rather, each pipeline stage may have one dedicated thread or, in the case of parallel stages, a thread pool determined in size by the value of the number of replicas.

With regards to the parallel Bzip2 results in Figure 6, we observe an overall higher standard deviation. We think this situation is due to the variations in disk accesses, which is inflated by the big input file. Beyond that, the results were similar among all versions throughout the execution. The exception is `ff`, which upon reaching the 11 parallelism degree is affected by a load balancing issue accentuated by the use of hyper-threading. A single thread situated in one processor core process its workload faster than a physical and a virtual thread sharing another processor core. That is because the threads constantly switch CPU access between instructions to better fill the hardware pipeline, which in theory may render slower single thread performance but better overall program performance. Since the scheduling is static, all the execution is halted until all threads stop execution. This does not happen in the TBB versions, which generate work-stealing scheduling. For the most, `spar-tbb` achieves a very similar result to the manual `tbb` implementation.

The results of Ferret in Figure 5 demonstrate a similar situation to Bzip2 in the `ff` and `spar` versions. On the other hand, the TBB versions performed slightly better due to the work-stealing scheduling achieving a more balanced workload. Again, `spar-tbb` and `tbb` were constantly similar. However, it is noticeable that `pthread` is still the overall better performing implementation. Eventually, other versions catch up in performance at the 24 parallelism degree. Both Lane Detection (Figure 7) and Person Recognition (Figure 8) showcase the same performance situation previously explained for the `ff` version. However, `spar` and `ff` performed slightly better and similar to `pthread` compared to the other versions on Lane Detection before number of replicas 11. This is because Lane Detection computation and workload is naturally more well balanced than in the other 3 applications. The cost of performing dynamic

work-stealing scheduling is higher than the static scheduling alternative. Therefore, the Lane Detection application is a case where lower cost for workload scheduling performs slightly better before the number of replicas 12.

When evaluating the results, `spar-tbb` achieved a very similar result to the manual `tbb` in all applications. Regarding `spar-ff` and `ff`, they also achieved a very similar result in most cases. Finally, we observed that by generating TBB code with SPar, we achieved a superior performance result in Ferret, Lane Detection, and Person Recognition applications, specially utilizing the hyper-threading resource. In Table 3, we presented the total SLOC (Source Lines of Code) for each version, including the sequential implementation. The `spar` only contains one version since both `spar-ff` and `spar-tbb` use the same annotations. Alone, SLOC is not expressive enough to conclude which version is more productive. However, it provides an overview of the code intrusion that each API generates. Among all applications, `spar` achieved the lowest SLOC value. It is closely followed by `tbb` and `ff` versions, which are an abstraction layer below. As expected, `pthread` obtained the highest SLOC evaluation, since it is the most complex implementation. When putting these results together with the performance results previously presented, it is possible to observe that the highest SLOC implementations achieve the overall best performance results. This is because, they provide more customization options that can be leveraged by an experienced developer. Beyond reducing the SLOC number, SPar reduces the complexity switching between runtime versions (FastFlow and TBB). This allows for lower time-to-deploy when changing from one system to another.

Table 3: Total SLOC for each version

Version	Ferret	Bzip2	Lane Detection	Person Recognition
seq	223	1278	126	126
spar	258	1404	130	132
tbb	376	1483	170	175
ff	357	1607	164	174
pthread	623	1917	415	417

5 CONCLUSION

This paper discussed an approach to simplify the development of stream parallel applications by means of source code annotations. We also presented the proposed annotations schema and its usability in a simple example. Furthermore, we presented the code generation process, which operates through source-to-source transformations performed directly in the SPar's compiler AST. These transformations aim at converting the annotations into the proper calls to the supported parallel runtime. We have described these transformation rules for two different runtimes: FastFlow and TBB. Beyond that, we performed tests on a series of stream processing applications, which demonstrated the performance viability of the solution. The performance of the applications developed using our solution was very similar to the performance achieved using the manually developed versions. Moreover, by generating TBB, we improved performance on three of four tested applications. Beyond that, we also identified that our solution achieves the overall lowest code intrusion through a SLOC analysis. For future work, we intend to the expand the transformation rule set to support code generation for GrPPI, SkePU, and OpenMP.

ACKNOWLEDGMENTS

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nivel Superior - Brasil (CAPES) - Finance Code 001, Univ. of Pisa PRA_2018_66 "DECLware: Declarative methodologies for designing and deploying applications", the FAPERGS 05/2019-PQG project ParAS (Nº 19/2551-0001895-9), and the Universal MCTIC/CNPq 28/2018 project called SParCloud (Nº. 437693/2018-0).

REFERENCES

[1] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. 2014. FastFlow: High-Level and Efficient Streaming on Multi-core. In *Programming Multi-core and Many-core Computing Systems (PDC)*, Vol. 1. Wiley, 14.

[2] Henrique Andrade, Buğra Gedik, and Deepak S Turaga. 2014. *Fundamentals of Stream Processing: Application Design, Systems, and Analytics.* Cambridge University Press.

[3] Marco Danelutto, Tiziano De Matteis, Daniele De Sensi, Gabriele Mencagli, Massimo Torquati, Marco Aldinucci, and Peter Kilpatrick. 2019. The RePhrase Extended Pattern Set for Data Intensive Parallel Computing. *International Journal of Parallel Programming* 47, 1 (01 Feb 2019), 74–93.

[4] M. Danelutto and M. Torquati. 2014. Loop Parallelism: A New Skeleton Perspective on Data Parallel Patterns. In *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. 52–59.

[5] David del Rio Astorga, Manuel F. Dolz, Luis Miguel Sanchez, Javier García Blas, and J. Daniel García. 2016. A C++ Generic Parallel Pattern Interface for Stream Processing. In *Algorithms and Architectures for Parallel Processing*, Jesus Carretero, Javier Garcia-Blas, Ryan K.L. Ko, Peter Mueller, and Koji Nakano (Eds.). Springer International Publishing, Cham, 74–87.

[6] David del Rio Astorga, Manuel F Dolz, Luis Miguel SÃąnchez, J Daniel GarcÃŋa, Marco Danelutto, and Massimo Torquati. 2018. Finding parallel patterns through static analysis in C++ applications. *The International Journal of High Performance Computing Applications* 32, 6 (2018), 779–788.

[7] August Ernstsson, Lu Li, and Christoph Kessler. 2018. SkePU 2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems. *International Journal of Parallel Programming* 46 (2018), 62âĂŞ80.

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

[9] Dalvan Griebler, Marco Danelutto, Massimo Torquati, and Luiz Gustavo Fernandes. 2017. SPar: A DSL for High-Level and Productive Stream Parallelism. *Parallel Processing Letters* 27, 01 (March 2017), 1740005.

[10] Dalvan Griebler, Renato B. Hoffmann, Marco Danelutto, and Luiz Gustavo Fernandes. 2018. High-Level and Productive Stream Parallelism for Dedup, Ferret, and Bzip2. *International Journal of Parallel Programming* 47, 1 (February 2018), 253–271.

[11] Dalvan Griebler, Renato B. Hoffmann, Marco Danelutto, and Luiz Gustavo Fernandes. 2018. Stream Parallelism with Ordered Data Constraints on Multi-Core Systems. *Journal of Supercomputing* 75, 8 (July 2018), 4042–4061.

[12] Re'Em Harel, Idan Mosseri, Harel Levin, Lee-Or Alon, Matan Rusanovsky, and Gal Oren. 2019. Source-to-Source Parallelization Compilers for Scientific Shared-Memory Multi-core and Accelerated Multiprocessing: Analysis, Pitfalls, Enhancement and Potential. *International Journal of Parallel Programming* (08 2019).

[13] John L. Hennessy and David A. Patterson. 2019. *Computer Architecture: A Quantitative Approach.* Morgan Kaufman.

[14] 14882:2014 ISO/IEC. 2014. *Information Technology - Programming Languages - C++.* Technical Report. International Standard, Geneva, Switzerland.

[15] Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill. 2005. *Patterns for Parallel Programming.* Addison-Wesley, Boston, USA.

[16] Michael McCool, Arch Robison, and James Reinders. 2012. *Structured Parallel Programming: Patterns for Efficient Computation.* Elsevier Science.

[17] A. Navarro, R. Asenjo, S. Tabik, and C. Cascaval. 2009. Analytical Modeling of Pipeline Parallelism. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*. 281–290.

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

[19] Rephrase. accessed october 1, 2019. *The Rephrase project.* https://rephrase-eu.weebly.com/