The Journal of Supercomputing (2022) 78:7655–7676
https://doi.org/10.1007/s11227-021-04182-9

OpenMP as runtime for providing high-level stream parallelism on multi-cores

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

Accepted: 27 October 2021 / Published online: 3 January 2022

Abstract

OpenMP is an industry and academic standard for parallel programming. However, using it for developing parallel stream processing applications is complex and challenging. OpenMP lacks key programming mechanisms and abstractions for this particular domain. To tackle this problem, we used a high-level parallel programming framework (named SPar) for automatically generating parallel OpenMP code. We achieved this by leveraging SPar's language and its domain-specific code annotations for simplifying the complexity and verbosity added by OpenMP in this application domain. Consequently, we implemented a new compiler algorithm in SPar for automatically generating parallel code targeting the OpenMP runtime using source-to-source code transformations. The experiments in four different stream processing applications demonstrated that the execution time of SPar was improved up to 25.42% when using the OpenMP runtime. Additionally, our abstraction over OpenMP introduced at most 1.72% execution time overhead when compared to handwritten parallel codes. Furthermore, SPar significantly reduces the total source lines of code required to express parallelism with respect to plain OpenMP parallel codes.

Keywords Parallel programming · Stream processing · C++ · Parallel patterns · Pipeline · Code transformation

✉ Dalvan Griebler
dalvan.griebler@pucrs.br

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

[2] Laboratory of Advanced Research on Cloud Computing (LARCC), Três de Maio Educational Society (Setrem), Três de Maio, Brazil

1 Introduction

Multi-core processors are ubiquitous in modern hardware architectures. They expose parallelism in the form of replicated cores. Developers aiming to extract higher performance must deal with parallel programming. Developing parallel code for multi-core processors means that programmers must deal with error-prone concepts such as thread creation and management, cache optimizations, communication mechanisms, load balancing, data dependencies, critical or mutual exclusive data access, etc. Also, regular developers do not have the skills or knowledge to develop efficient parallel code. Instead, they are interested in easily achieving good performance without entering in low-level and architecture-specific details. That is where parallel programming abstractions apply. In particular, domain-specific languages (DSL) are computer languages that allow the programmer to specialize on a specific domain [8, 27]. They can be an efficient alternative to achieve better performance through parallel computing while still allowing the developer to mainly focus on specific algorithmic solutions to its own field of work or study.

When adopting parallel abstractions, one may consider structured or non-structured parallelism. The structured approach equips the developers with patterns or algorithmic skeletons (i.e. *Map*, *Reduce*, *Pipeline*). These patterns are generally developed by experts in the parallel computing field and are made to be readily available for a series of well-defined situations commonly found in the development of parallel programs. For multi-cores, several application programming interfaces (API) have been developed with this intent such as Intel Threading Building Blocks (TBB) [25] and FastFlow [2]. They require restructuring the sequential code into their respective API and the necessity for programmers to understand the semantics and characteristics of each parallel pattern. This way, it is possible for the application programmer obtaining a parallel version without entering into low-level parallelism details. The other approach for parallel programming is adopting non-structured techniques for implementing ad hoc parallelism. Common examples are OpenMP [23] and Pthreads [15], although they show contrasting complexity and flexibility levels. In this case, the programmer can freely write parallel code as long as it is correct regarding the respective API syntax and semantics. However, it requires the programmer to deal with low-level parallelism aspects such as critical sections, scheduling, data management, and others.

Stream processing applications are generally computationally intensive. Therefore, they must leverage parallel computing to complete processing in a feasible time. This class of applications is characterized by a data stream flowing through a sequence of independent computational stages or filters. Some common examples can be found in audio and video processing, sensor monitoring, cryptography, data analysis, statistics, etc. When developing parallelism for these applications, a linear or nonlinear Pipeline [18] pattern can typically be employed to represent most common situations [2].

Parallel APIs follow distinct design principles that may impact positively or negatively in programmability and productivity aspects of writing parallel code

for a given application domain. For example, OpenMP is popular for data parallelism; however, it requires a non-trivial implementation when it is used to develop parallel stream processing applications [8, 24]. Extra mechanisms must be sought in order to correctly and efficiently develop stream applications with OpenMP, as will be discussed in Sect. 2. The lack of higher-level abstractions to parallelize stream processing applications defeats the simplicity purpose of OpenMP's pragma directive approach.

To tackle this problem, our goal is to employ SPar[1] [9] as a DSL that aims at providing parallel abstractions for stream processing applications through the use of standard C++ code annotations. The idea is to use SPar's own compiler to transform code annotations into a supported runtime parallel system directly on the AST (Abstract Syntax Tree). This way, efficient parallel code may be generated from simple code annotations that do not require significant sequential code refactoring from the programmer neither apriori parallel pattern know-how. In prior works, we successfully managed to automatically generate parallel code for the FastFlow [9] and TBB [13] runtimes, which provide a similar level of abstraction because both of them employ a structured parallel programming approach. In contrast, this work posed a new challenge: automatically generating OpenMP stream processing code without changing the original syntax and semantic rules for SPar annotations. We innovate by investigating the methodology proposed in this work for using a high-level language like SPar (contains only five annotation attributes) to perform automatic and efficient OpenMP parallel code generation in stream processing applications. On the other hand, other works mainly focus on C++ templates, new parallel patterns, and/or extensions to current libraries for improving parallel programming.

We made the following contributions: (1) Efficient OpenMP linear and nonlinear Pipeline implementation for parallel stream processing; (2) new compiler definitions and transformation rules for generating OpenMP parallel code, which were implemented in the SPar compiler; and (3) performance and programmability analysis of the proposed solution with a set of representative applications.

The remainder of this paper is organized as follows. Section 2 explained difficulties and requirements for developing stream processing applications with OpenMP and presented a structured Pipeline template for it. Then, Sect. 3 described our methodology for generating lower-level OpenMP parallel code using SPar's language. The experimental analysis was conducted in Sect. 4 considering performance and programmability aspects. Subsequently, we described and discussed related work in Sect. 5. Finally, Sect. 6 concluded this study and presented some final remarks.

2 OpenMP pipeline for stream processing

The goal of this Sect. 2 is demonstrating the challenges introduced by OpenMP in stream processing applications as well as demonstrating an efficient solution. In Sect. 2.1, we first reasoned about the challenges, possible solutions, and

[1] https://gmap.pucrs.br/spar.

requirements for developing stream processing applications with OpenMP. Then, in Sect. 2.2, we employ these concepts and demonstrate how to efficiently develop an OpenMP Pipeline.

2.1 OpenMP for stream processing

Writing parallel code using OpenMP is difficult when targeting stream processing applications [24]. To approach this issue, possible solutions are creating language extensions, deploying parallel patterns, or DSLs (domain specific languages). A discussion about the solutions that were already investigated in the past can be found in the related work (Sect. 5). In short, none of these works have solved the programmability gap between OpenMP and stream processing. Instead, we propose a new approach by combining the Pipeline parallel pattern strategy with a domain specific language suitable for stream processing. This way, we leverage the best of both worlds: parallel patterns efficiency and programmability with DSL's higher level of abstraction.

Although OpenMP provides task-based parallelism (*section* and *task* directives), it lacks two essential key aspects that are not supported by the standard API. The first one is MPMC (Multiple Producer Multiple Consumer) parallel thread communication. OpenMP has *depend* clauses to express task producer–consumer relations. However, it cannot be used with stream processing applications since it is not efficient for producer-consumer cycles (i.e. for loop generating items) and lacks synchronization. Therefore, an external communication queue must be implemented or imported. The second key aspect is the necessity for efficient thread runtime synchronization to enable concurrent MPMC communication. This must be sought from external sources. Based on these aspects, we defined the following requirements that need to be fulfilled for our *Pipeline* template in OpenMP:

1. *Efficient MPMC queues* Each communication queue must be able to handle any positive number of producers and consumers greater than 0.
2. *Synchronization* It is necessary to synchronize insert and remove operations in the queue. To that end, we used default C++ `conditional_wait` mechanisms coupled with `mutex`.
3. *Mutually exclusive access* To avoid concurrency issues, we used a single default C++ `mutex` mechanism. Standard OpenMP locks could not be used since; with it, it would not possible to synchronize producers and consumers.
4. *End of processing signals* When terminating execution, the queue must propagate end-of-processing signals through all associated producers and consumers. Additionally, when end-of-processing is propagated through the stream, it must guarantee no data item is left unprocessed.
5. *Unpredictable Workload* The queue must handle varying or starving workload.
6. *Non-blocking behavior* Producers or consumers unable to operate on the queue must not maintain a busy-waiting or blocking status. They should halt execution and wake up only when work can resume. This was achieved with default C++ `conditional_wait` mechanisms.

7. *Customizable buffer size* Configurable and restrained buffer size. A very small buffer size may turn into a stream bottleneck, whereas a huge buffer size may use unnecessary extra memory.
8. *Circular FIFO access pattern* First-in-first-out data access in a circular pattern.
9. *Ordering constraints* Due to the non-deterministic nature of parallel processing, stream items may reach certain processing stages in a different order from which they were generated. This is a problem for a number of applications, where the integrity of the output depends on the order of the stream items (e.g. frame order on an output video). For that, the implementation must accommodate the possibility to maintain the original stream order on a sequential or output stage. For that purpose, we used the algorithm proposed in [12], which adds a tagging module to the stream generator and a re-ordering module to the stream consumer.

To fulfill these requirements, we created an MPMC synchronized communication queue named `SParSharedQueue`, which is a concurrent queue implemented with standard C++ mechanisms. It was used in our Pipeline template. Note that, although alternative MPMC queues can be employed, there are specific mechanisms that still must be manually implemented by the programmer such as communicating the end of stream or re-ordering data items. Effectively, this would expose the developer to many low-level parallelism details that are difficult to reason about in the context of multi-core architecture and systems. `SParSharedQueue` helps to ease the process of integrating these features. Alternatively, `SParSharedQueue` could be easily adapted to be used as a skeleton for integrating another communication queue.

2.2 OpenMP pipeline

Regularly, OpenMP implementations are not classified as structured parallel programming. However, we specified a strictly structured approach in Listing 1 that is a Pipeline pattern. Here, we did not propose a new Algorithm; rather, we exemplify with this pseudo-code how to compose a Pipeline using OpenMP in C++. We have studied different OpenMP mechanisms for implementing stream parallelism, and this was the most efficient. We can observe that it is a verbose code. Many mechanisms are exposed to the programmer, and it is difficult to separate the application code from the parallelism strategy. Later, we used this Pipeline template as the generation target code for SPar in Sect. 3. This code is equivalent to the manual implementation when using OpenMP for stream processing applications, plus the `SParSharedQueue` mechanisms that sum approximately 88 source lines of code. The code in Listing 1 parallelized a generic three stages stream processing application.

Regarding Listing 1, the communication queues are defined in lines 1 and 2. In this example, we have multiple Pipeline stages that communicate with each other using two different queues: one for the consumer and other for the producer relation. These queues take as initialization arguments the maximum buffer size (`q_size`) and the number of producers. Next, we create a parallel region in line 3 where the queues are defined as shared arguments.

```
1  SParQueue<data> * queue1 = new SParQueue<data>(q_size,1);
2  SParQueue<data> * queue2 = new SParQueue<data>(q_size,
       par_threads);
3  #pragma omp parallel shared(queue1,queue2)
4  #pragma omp single nowait
5  #pragma omp taskgroup
6       for(int thread=0; thread < 1; thread++){
7         #pragma omp task
8         {
9           struct data * stream_item;
10          for(/*Generator loop*/){
11            //Stage 1 Code
12            queue1->Add(stream_item);
13          }
14          queue1->NotifyEOS();
15      }   }
16      for(int thread=0; thread < par_threads; thread++){
17        #pragma omp task
18        {
19          struct data * stream_item;
20          while(1){
21            stream_item = queue1->Remove();
22            if(/*End of Stream*/) break;
23            //Stage 2 Code
24            queue2->Add(stream_item);
25          }
26          queue2->NotifyEOS();
27      } }
28      for(int thread=0; thread < 1; thread++){
29        #pragma omp task
30        {
31          struct data * stream_item;
32          while(1){
33            stream_item = queue2->Remove();
34            if(/*End of Stream*/) break;
35            //Stage 3 Code
36      }  }  }
```

Listing 1: OpenMP 3 stages Pipeline template pseudo-code example.

In lines 4 and 5, OpenMP directives `single nowait` and `taskgroup` are used to guarantee that the parallel region will execute once and all tasks finish before proceeding. Then, lines 6 to 15 implement the first stage of the Pipeline, where the parallel generation loop in line 6 creates as many threads as specified. This is typically the generation stage that must iterate the stream items. Items are en-queued (queue `Add` method) in line 12 inside the first stage. After the stage has finished generating items, it signals the end of the stream in line 14 using `NotifyEOS` method.

The first stage is the only unique one, while every subsequent stage follows a similar pattern between each other. Therefore, we only detailed the stage in lines

16–27. It all starts with its parallel generation loop (line 16) which creates as many parallel threads as specified. This number must be defined by the developer. Sequential stages, such as the one in line 28, only spawn one thread. In line 21, it defines a temporary stream item for local computations. Then, in lines 20–25, it begins an infinite iteration statement that will constantly try to consume items from the queue in line 21 (queue `Remove` method). The consume operation from the queue is non-blocking. Then, in line 22 it verifies if the end of the stream was reached and if so breaks from the iteration statement and notifies the end of stream to the next queue in line 26. In line 23, the regular stage code is executed. After it is done processing, it produces and adds the item to the next communication queue (line 24 `Add` method). The only exception is the last stage, which needs to propagate end of stream signals.

3 Abstracting OpenMP stream parallelism with SPar

The goal of this section was to describe how SPar may be used for abstracting the complexity added by OpenMP in stream processing applications. In short, we aimed to leverage high-level SPar annotations while automatically generating efficient OpenMP parallel code using the Pipeline template discussed in Sect. 2. With that, our goal was to simplify the process of developing OpenMP stream processing applications. Initially, we described SPar's language and compiler methodology in Sect. 3.1. Then, we presented OpenMP code generation in SPar in Sect. 3.2.

3.1 SPar abstractions

SPar (an acronym for Stream Parallelism) is a domain-specific language compatible with C++ structured parallel programming [9]. SPar language provides a set of five attributes that describe different aspects commonly found in stream processing applications. The attribute mechanism is supported by standard C++. With this set of attributes, SPar uses the CINCLE compiler infrastructure support to perform source-to-source code transformations directly on the AST [8]. This strategy allows SPar's compiler to automatically generate parallel code targeting a parallel runtime system (e.g. FastFlow or TBB). It does so based on a set of transformation rules unique to each underlying interface. By default the parallel runtime system is FastFlow. We did not explain FastFlow and TBB transformation rules in this paper, but more details may be found in [8] and [13]. The underlying parallel runtime system is responsible for the performance gains, and both FastFlow and TBB have ongoing active projects. With SPar, the programmer can parallelize C++ sequential code with an extra abstraction layer while maintaining the original code structure. Ultimately, SPar's goal is to increase parallel code development productivity at the lowest possible performance cost. The annotations were explained in Sect. 3.1.1 and code generation in Sect. 3.1.2.

Table 1 SPar attributes

Attribute name	Type	Short description
ToStream	ID	Defines the scope of the parallel stream region.
Stage	ID	Defines the scope of one processing stage or filter inside the parallel stream region.
Input	AUX	Defines the data consumed by stream or stage regions.
Output	AUX	Defines the data produced by stream or stage regions.
Replicate	AUX	Parallelizes the associated stage region.

3.1.1 SPar annotations

After ISO C++11, the C++ language provided default support for attribute mechanisms [14]. They can be used with an annotation, which is delimited by double brackets and takes a list of attributes as input [[attr-list]]. Furthermore, the C++ standard grammar 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 whether it can be used for types, objects, code blocks, etc. Each C++ compiler may support different attributes, but every compiler must be able to parse them and place them in the AST. For that purpose, SPar has its own compiler. In SPar's language, five different attributes have been defined for expressing common stream processing concepts. They are briefly described in Table 1 where ID stands for identifier and AUX for auxiliary attributes. Every attribute list must begin with an ID attribute which may be followed by a list of complementary AUX attributes. In this work, we leveraged these existing attributes. An exhaustive syntax and semantics detailed description may be found in [8].

Relative to the semantics, some rules must be respected to generate correct parallel code. First, every annotation must be placed in front of an iteration or compound statement. This ensures the correct definition of each computational stage. Also, there can be no nested `ToStream` attributes (stream of streams), as it's not possible to generate a stream flowing inside another stream in SPar's language. However, the number of `Stage` inside a `ToStream` region is not restricted. Both `Input` and `Output` must contain at least one argument where the arguments must match for producer-consumer relations. Finally, `Replicate` can only be used with a `Stage` attribute, which must be a stateless operator. This means the programmer can not leave unattended critical/atomic code regions inside a replicated Stage.

An example of SPar language usability is demonstrated in Listing 2. It is a C++ algorithm that discovers and prints palindrome numbers between 1 and 1000. `ToStream` is used to mark the stream region, which is the scope of the for loop. The code situated between `ToStream` and the first `Stage` attribute is always the stream generation stage. In lines 4 and 10, `Stage` is used to specify two stream computing regions. The first one uses `Input` and `Output` to, respectively, consume (from the previous stage) and produce (to the next stage) `num` and `rev`. Specifying every `Stage` is a task that must be performed by the developer.

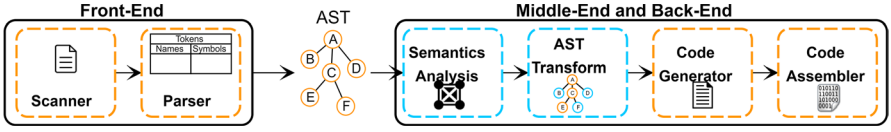


Fig. 1 SPar compilation flow. Extracted from [8]

The code in Listing 2 also has a `Replicate` that effectively spawns 4 parallel threads that compute that stage's code. In this case, this is a safe replication, since this is a stateless stage. In SPar, safe replications must be ensured by the programmer. Numbers are processed independently from one another since they do not require access to past variable states. Additionally, this number of stages must be defined by the user. This is an important part that may have high impact on performance. To ease this burden, it has been developed a research in self-adaptive parallelism to automatically define this number during execution time for a different parallel runtime library [28], which can be extended to OpenMP in the future. Lastly, `Stage` in line 10 consumes `num` and `rev` from the previous stage with `Input` checks if the number is palindrome and prints it. It is a sequential stage since it does not have a `Replicate` attribute.

```
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 }    }
```

Listing 2: Example for using the code annotations.

3.1.2 Parallel code generation

After the code is properly annotated, SPar's compiler begins the parallel code generation using CINCLE's support. CINCLE (Compiler Infrastructure for new C/C++ Languages Extensions) is a compiler infrastructure for generating new C/C++ embedded DSLs. It provides basic features and a simple interface to enable AST transformations, semantic analysis, and source-to-source code generation. Figure 1 depicts the compilation flow of SPar's compiler using CINCLE. In it, the orange

dotted lines represent CINCLE modules, while cyan blue dotted lines represent SPar's compiler modules.

The compilation process starts with a call to GCC compiler, which performs a semantic and syntax analysis of basic C++ code. After that, the code is scanned by CINCLE and the resulting tokens are parsed in order to create an AST representing the entire semantics of the C++ ISO. This AST is fully accessible, allowing for complete control of every node. This way, the code transformations can be performed directly on the AST during compile time. Nodes can be removed or shifted freely, allowing for any possible code transformation required to generate the calls to the abstracted underlying parallel runtime system. This is one of the main advantages of CINCLE, since GCC compiler does not support direct AST transformations [8].

Subsequently, the first SPar compiler module that performs semantic analysis checks the AST for semantic errors in the annotations implemented by the programmer. Inconsistencies are reported back to the developer. If everything is correct, later, the code transformation step performs the appropriate transformations directly on the AST. For that, it switches the annotation nodes in the AST for an internally generated sub-tree that contains the proper code with the underlying parallel runtime system. However, this step must be done with extensive care to the C++ ISO [14], as errors can easily break the entire code. Finally, GCC is called again to produce the final binary file, which contains parallel code automatically generated by SPar compiler. There are also a few compilation flags that can be used to change the behavior of the runtime system. They are:

- `spar_ondemand`: used to generate on-demand scheduling [2].
- `spar_blocking`: used to activate FastFlow blocking mode.
- `spar_ordered`: used to guarantee that output stream elements are delivered respecting the original input order.

3.2 OpenMP with SPar

In Sect. 2, we showed how to implement stream parallelism in OpenMP using the Pipeline parallel pattern. In this section, our contributions are the new definitions, transformation rules, and compiler algorithm we created to abstract low-level OpenMP stream parallelism details. We explained and characterized the transformation phase necessary to automatically generate stream parallelism code with OpenMP using SPar's language and compiler.

We define the Pipeline as: $pipe(stage_0, queue_0, stage_1, ..., queue_{n-1}, stage_n)$. This is not de-facto OpenMP code, but rather a high-level representation to guide the compiler in generating correct structured code. Each stage can behave sequentially (*seq*) or parallel (*par*). Therefore, a parallel stage is denoted $par(stage_n)$ and $seq(stage_n)$ for sequential stage. Each $stage_n$ is an independent computational stage. Typically, the Pipeline parallel pattern defined in the literature does not specify the queues inside its argument list [19]. The queues are implemented by the parallel pattern and abstracted to the programmer. Instead, OpenMP does not follow

Table 2 Definitions of the transformation rules

D_0	An additional gatherer stage ψ and a queue are generated when the last block of code is annotated with S that contains in its attribute list R_n and its associated T contains an O.
D_1	A queue must be generated before each S inside the T attribute list.
D_2	A generic block of code is a sequential stage if its annotation list S does not contain the attribute R_n.
D_3	A generic block of code with an annotation list S containing an R_n attribute may be a parallel stage.
D_4	A T annotation is transformed into a *pipe* when the attribute list contains at least one S and its block of code is the sequential stream generator.

a structured parallel programming approach and this implies that communication queues are explicit to programmers. With our Pipeline pattern definition, we allow that non-structured runtimes such as OpenMP and possibly others (C++ standard threads or Pthreads) follow a pseudo-structured approach.

In addition to the Pipeline template, we provided a set of rules to perform code transformations and support automatic parallel code generation. The transformation is successful when it can translate the annotations into a valid *pipe* template structure, such as described in Sect. 2. This set of rules must be able to accomplish the semantics described in Sect. 3.1.1. Each supported underlying API has its own set of rules. In this work, we presented new rules for automatically generate OpenMP stream processing code. To clarify the rest of this Section content, we defined some terms. A $\square$ is a generic block of code, and the scope of the sentences is represented by {...}. Generic blocks of code can contain any valid C++ code, including function calls. Annotations were marked as [[...]] and may contain a list of attributes as an argument. The ID attributes are T (`ToStream`) and S (`Stage`). The AUX attributes are I (`Input`), O (`Output`), and R (`Replicate`). ψ is a special case `Stage` sometimes required. In Table 2, we summarized the definitions to generate the transformation rules specific for OpenMP Pipeline (Sect. 2.2). It is important to specify that more than one definition D_i may apply to the same annotation.

The flow in Fig. 2 is a visual depiction of the decision process of applying these definitions. Starting the process, it checks the first annotation, which, if correct, must always be a `ToStream`. Then, it unconditionally applies D_4 and moves back to question which is the next annotation. Every subsequent annotation should be a `Stage`, which always applies D_1. This is the queue generation that happens for every stage. If this `Stage` is not replicated, it applies D_2 and goes back to the initial step. Otherwise, it applies D_3, which considers that as a parallel stage. The following decisions are solely to apply or not D_0. If the replicated stage is not the last one or if `ToStream` does not have an `Output`, it does not apply D_0 and goes back to the start. Otherwise, D_0 is applied and it goes back to the next annotation decision. From this point forward, if there is no other annotation, the process is ended. In summary, this whole process iterates all code annotations and decides the correct parallel Pipeline template to be generated.

To better understand the use of these definitions, we demonstrated their application in Rule 1. It refers to $[[T_0]]\{\square_0, [[S_1, I_1, O_1, R_1]]\{\square_1\}, [[S_2, I_2]]\{\square_2\}\}$. Remember that according to our previous definition T is a `ToStream`, S is a

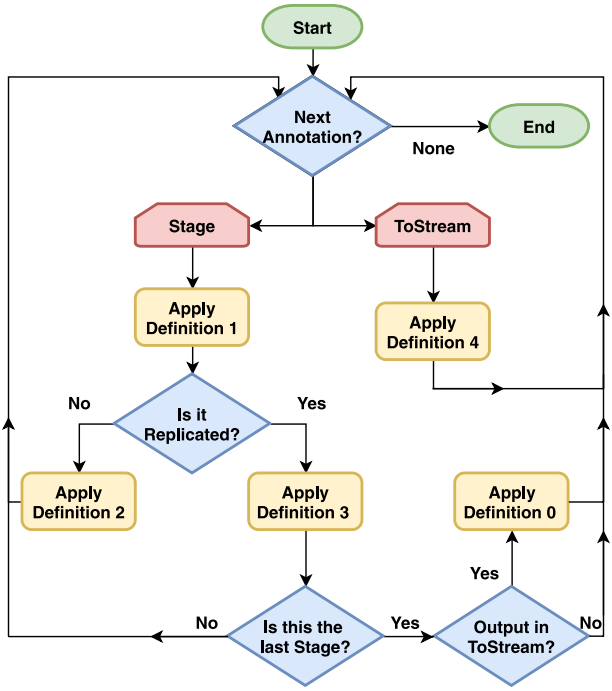


Fig. 2 OpenMP definitions flow

`Stage`, and so forth. Annotations are parsed starting from the first `ToStream` up to the last `Stage`. Initially, $[[T_0]]$ applies for definition D_4, which generates a Pipeline with $\square_0$ as the first sequential stage. The first `Stage` $[[S_1, I_1, O_1, R_1]]$ applies D_1, which generates the first queue, and D_3 to generate a parallel stage. Then, the last annotation with $[[S_2, I_2]]$ will be the a sequential stage to conform with definition D_2 while generating its queue for D_1. In this case, D_0 does not apply. D_0 is used in rarer cases, where the output of a replicated stage might be used outside of the stream region.

$$\begin{gathered}[[T_0]]\{\square_0, [[S_1, I_1, O_1, R_1]]\{\square_1\}, [[S_2, I_2]]\{\square_2\}\} \\ \Downarrow \\ pipe(seq(\square_0),\ queue_0, par(\square_1),\ queue_1,\ seq(\square_2))\end{gathered} \tag{1}$$

Rule 2 refers to $[[T_0]]\{\square_0, [[S_1, I_1, O_1]]\{\square_1\}, [[S_2, I_2, R_2]]\{\square_2\}\}$. The $[[T_0]]$ annotation matches with D_4, which generates a *pipe* receiving as first sequential stage $\square_1$. Then, the $[[S_1, I_1, O_1]]$ annotation matches with the definition D_2, generating a sequential stage, while D_1 generates its queue. Then the $[[S_2, I_2, R_2]]$ annotation matches for D_3 to generate a parallel filter and the unconditional D_1 to generate the stage communication queue.

$$\begin{gathered}[[T_0]]\{\square_0, [[S_1, I_1, O_1]]\{\square_1\}, [[S_2, I_2, R_2]]\{\square_2\}\} \\ \Downarrow \\ pipe(seq(\square_0),\ queue_0, seq(\square_1),\ queue_1,\ par(\square_2))\end{gathered} \tag{2}$$

Rule 3 is a special case where D_0 applies. $[[T_0, O_0]]\{\square_0\}, [[S_1, I_1, R_1]]\{\square_1\}\}$ applies for D_0 since the last stage $[[S_1, I_1, R_1]]$ contains a R and $[[T_0, O_0]$ contains an O. Definition D_1 applies for $[[S_1, I_1, R_1]]$ since its a *Stage* while also applying D_3 since it is replicated. Finally, $[[T_0, O_0]]$ applies D_4 which is unconditional for T.

$$\begin{gathered}[[T_0, O_0]]\{\square_0\}, [[S_1, I_1, R_1]]\{\square_1\}\} \\ \Downarrow \\ pipe(seq(\square_0),\ queue_0, par(\square_1),\ queue_1,\ seq(\psi))\end{gathered} \tag{3}$$

Once the correct Pipeline pattern is assembled from the definitions, the compiler executes the final step which is responsible for automatically generating OpenMP parallel code. From this point on, the code generation is performed directly on the AST using source-to-source transformations that will ensure the full semantics of the sequential program annotated with SPar. The compiler will replace every SPar annotation from the C++ program and generate actual OpenMP code leveraging the Pipeline template previously defined in Sect. 2.2. Since OpenMP is non-structured, we implemented a new compiler algorithm to perform this compiler step. The algorithm supports automatic code generation for the Pipeline template and that employs the new definitions and transformation rules previously presented. Besides, we had to manually develop the reordering algorithm (introduced in [11]) for the SPar compiler because OpenMP does not offer task ordering.

Regarding SPar's compilation flags presented in Sect. 3.1.2, the new compiler algorithm for OpenMP handles them as follows:

- `spar_ondemand [integer_size]`: used to generate on-demand scheduling. Sets different `SParSharedQueue` buffer/queue size since the access pattern is already on-demand. Default buffer/queue size is 100 items and can be changed during compile-time.
- `spar_blocking`: used to activate blocking mode. This flag was not supported by OpenMP SPar compiler since the implemented `SParSharedQueue` is non-blocking.
- `spar_ordered`: used to say that output stream elements must be delivered respecting the original input order. Behavior left unaltered.

In summary, in this Section, we created and explained new SPar transformation rules for OpenMP stream processing that could be used in SPar for other non-structured parallel programming API's such as C++ threads. The main reason we supported OpenMP on SPar is that it performs better under certain circumstances, as will be shown in Sect. 4. Consequently, a C++ program already annotated with SPar may significantly improve performance only by changing a compilation flag to select the OpenMP runtime.

4 Experiments

The goal of this Section was to assess the performance of the parallel code automatically generated by the SPar compiler for OpenMP stream processing. We also evaluated programmability aspects for handwritten OpenMP code compared to SPar language. First, we described the experimental setup in Sect. 4.1. The performance was evaluated in Sect. 4.2 and programmability in Sect. 4.3.

4.1 Experimental setup

For the experimental analysis, we selected four different stream processing applications that represent different computational characteristics. They have different I/O intensity, varied data granularity, critical access requirements, throughput, and stream data reordering. They are: LaneDetection [12], Person Recognition [12], Bzip2 [7], and Ferret [3]. The parallel implementation details are beyond the scope of this paper. However, we described their basics as well as referenced where more detailed information can be found below:

- Ferret [3]: application for content similarity search in feature rich data such as video, audio, and images. In this case, the version utilized was adapted for image search. Its algorithm [3] finds the top 50 similar images. It was originally parallelized with Pthreads using a 6 stages Pipeline. The first stage implements sequential input and the final one sequential output, respectively. The four middle stages execute the bulk of the processing and are all executed in parallel.
- Bzip2 [7]: is a widely used lossless compression program in Linux-based distributions. It has independent modules for compression and decompression. The original Pthreads implementation developed a three stages Pipeline and a handwritten reordering algorithm in the output filter to maintain the integrity of the file.
- Lane Detect [12]: reads an input video stream from a camera recording a lane in front of the vehicle. Then, it outputs the detected lanes after a sequence of filters. It utilizes Canny filter and Hough Transform, which are provided by the OpenCV C++ image processing library. Accuracy of the system is dependent on OpenCV [4] detection algorithms. The parallel implementation technique uses a three-stage Pipeline where the first and last stages perform sequential I/O and the central one processes individual frames in parallel. The frame order must be guaranteed to keep the integrity of the video.
- Person Recognition [12]: reads a video input stream from a camera positioned in front of a point of interest. Then, it detects faces and uses an image database to recognize if that face belongs to a specific person. This database is pre-processed and derives from a series of training images from one person of interest. This application is also supported by OpenCV image processing runtime. Accuracy of the system is dependent on OpenCV [4] detection algorithms. The parallelism implementation consists of a three-stage Pipeline with a middle parallel stage and the output video frames must also be reorganized before outputting them.

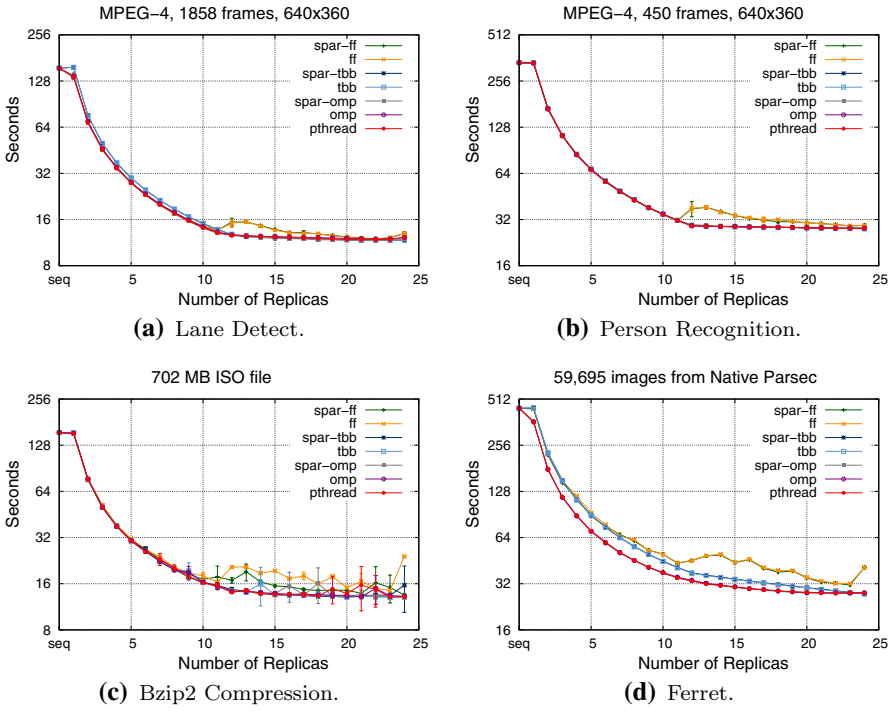
All tests were carried out in a machine equipped with 32 GB of RAM and two processors Intel(R) Xeon(R) CPU E5-2620 v3 2.40 GHz (total of 12 physical cores and 24 threads with Hyper-Threading). The operating system was Ubuntu Server 64 bits kernel 4.15.0-88-generic, and GCC 7.4.0. The compilation was also performed with the -O3 optimization flag. Other software details are OpenCV version 2.4.13.6, TBB (interface version 9107), and FastFlow version 3.0 for all applications except Bzip2 and Ferret, which are version 2.2.0 due to missing features in newer versions. Additionally, FastFlow versions did not use default thread mapping with LaneDetect and PersonRecognizer applications to increase performance. Moreover, we guaranteed the correctness of the parallel versions by comparing the hash value of the output with the sequential version.

We tested handwritten versions for FastFlow (`ff`), TBB (`tbb`), our OpenMP Pipeline (`omp`), and Pthreads (`pthread`). These are consolidated parallel programming interfaces from industry and academia. These will be further explained later in Sect. 5. Both FastFlow and TBB can be generated with SPar [9, 13], and, therefore, we also tested SPar generated code (`spar`) for the supported interfaces and OpenMP introduced in this work. This way, there can be a fair comparison to assess SPar's impact on the performance regarding the handwritten implementation. Note that the annotated source code is the same for every different `spar` version, so performance differences relate solely to code generation and underlying runtime parallel system. Save for `omp`, every other handwritten implementation, as well as SPar's annotated source code, is either from external [1, 3, 7] or previous studies [10, 11].

For every application, the execution time was chosen to assess performance. With the execution time, workload information (depicted in the title of the graphs), and the number of replicas, most other performance metrics can be derived. Each plotted value in the graphs was obtained from the arithmetic mean of 5 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 some cases as it is mostly negligible. Concerning the graphs, the x axis is always the number of replicas (related to the `Replicate` attribute), which goes from 1 up to 24 (number of cores with hyper-threading in the system). Another important aspect to consider is that the number of replicas in the x axis may not represent the actual active thread count in the system. Rather, each runtime may spawn one dedicated thread for each stage, or in the case of parallel stages, a thread pool is determined in size by the value of the number of replicas. In the graphs, the y axis is the execution time in seconds, using a logarithmic scale 2.

4.2 Performance

The Lane Detect results depicted in Fig. 3a showcased that the application scaled up to the 12th replica. This same behavior repeated for every other application. That is because the test machine only had 12 physical cores. Another detail is that the FastFlow versions deteriorated performance after the 11th replica since they suffered from load balancing issues due to FastFlow's static scheduling. Even though the machine has 12 physical cores, it happened at 11 replicas because FastFlow sets



(a) Lane Detect. **(b)** Person Recognition.

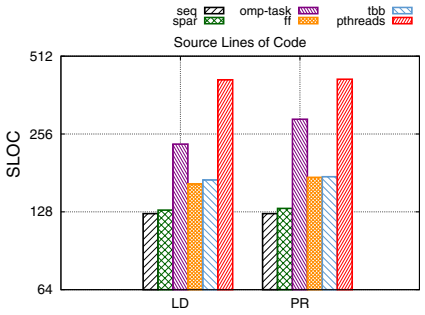
(c) Bzip2 Compression. **(d)** Ferret.

Fig. 3 Experimental results for execution time

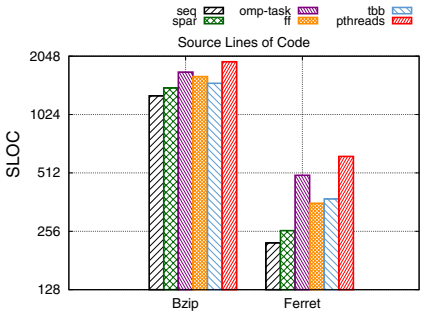
aside a dedicated thread for stage communication. This did not happen in the `tbb` versions, which generate work-stealing scheduling. Regarding `omp`, among all applications, it performed up to 0.64% less than `pthread`, indicating that our new compiler algorithm is efficient since the OpenMP code automatically generated is comparable to a handwritten low-level Pthreads program.

Figure 3b shows the results for Person Recognizer, which are mostly similar to Lane Detect. This demonstrated the consistency of the implementation under drastically varying workload. In this case, Lane Detect had a maximum throughput of 157 FPS (frames per second) while Person Recognizer had 16 FPS. Again, for Person Recognizer FastFlow versions, the same load balancing issues happened after 11 replicas for the same reasons as Lane Detect. Relative to OpenMP and TBB versions, SPar code automatically generated performed very similarly to its corresponding handwritten implementation. In this application, `spar-tbb` and `tbb` had 0.17% different performance wise while `spar-omp` and `omp` were at most 1.77% apart. Also, `spar-ff` and `ff` only differed in execution time by 0.74%. Despite the comparable performance, the advantage of SPar over the handwritten implementations is the higher-level parallel programming interface. This is further demonstrated in the programmability discussion in Sect. 4.3.

Now, Fig. 3c shows the results for Bzip2 application. The analysis here is more complex due to the higher standard deviation. Generally, results were mostly similar, but during standard deviation spikes, they differed by up to 10%. This behavior was



(a) LaneDetect (LD) and PersonRecognition (PR).



(b) Bzip2 and Ferret.

Fig. 4 Source lines of code

also observed in [13]. This may be caused by variations in disk access or machine configuration. In fact, we confirmed this by performing tests with the same software details in a different machine, where the standard deviation was mostly negligible. Here, `spar-omp` and `omp` were only 1.24% separated.

The final Ferret results are shown in Fig. 3d and highlight the main contributions of our work. The results demonstrated a clear performance difference between OpenMP, TBB, and FastFlow versions. The best versions were `pthread` and `omp | spar-omp` with a maximum observed difference between the two of 1.72%. OpenMP performed better than TBB, which in turn performed better than FastFlow. In hard numbers, at the 12th replica, TBB was 10.1% worse than OpenMP, FastFlow was 17.04% worse than TBB, and FastFlow was 25.42% worse than OpenMP. Again, SPar versions were very similar to their handwritten implementations counterparts, with performance differences lower than 2.49%. Despite the higher-level of abstractions, the execution time is comparable.

4.3 Programmability

In this section, the goal was to evaluate programmability aspects for each parallel version evaluated in the performance Sect. 4.2. In Fig. 4, we present the total SLOC[2] (Source Lines of Code) for each version, including the sequential implementation `seq`. The `spar` version only contains one entry since both `spar-ff`, `spar-tbb`, and `spar-omp` used the same annotations. SLOC does not consider blank lines and comments, only valid programming language syntax. Alone, SLOC is not expressive enough to conclude which version is more productive to code with. However, it provides an overview of the level of code intrusion that each API introduces. It represents how much extra code each interface requires to enable parallelism.

Among all applications, `spar` achieved the lowest SLOC value. At the highest amount of SLOC for Bzip2, SPar introduced 9.85% extra SLOC compared to `seq`.

[2] Obtained with SLOCCOUNT tool.

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. In fact, `pthread` introduced up to 230.95% extra SLOC regarding `seq`. The `omp-task` version was the second highest overall SLOC count (up to 123.31%). When putting these results together with the performance results previously presented, it is possible to observe that the highest SLOC implementations achieved the overall best performance results. This is because they provided more customization options that can be leveraged by an experienced developer. Beyond reducing the SLOC number, SPar reduced the complexity of switching between runtime versions (FastFlow, TBB, and OpenMP) since the SPar annotations were the same for every one of these APIs. The code can simply be recompiled with a different flag. This allows for lower time-to-deploy when changing from one system to another.

5 Related work

In this section, we selected related work that investigate OpenMP for stream processing or present code generation solutions similar to SPar. The difficulties of developing stream processing applications with OpenMP have been noticed by other researchers [8, 24]. Aiming to solve this problem, the ompSs [22] programming model proposed the use of *in, out, inout pragma* directives to express data-flow parallelism in multi-core, GPGPU, and FPGA environments. They are language extensions. However, their clauses are not sufficient to describe iterative producer item generation cycles to feed consumer stages. Another solution called OpenStream [24] extended existing *pragmas* to better represent stream processing application characteristics. They reported better performance than ompSs. The main reason is that OpenStream uses persistent tasks, while ompSs assumes lightweight tasks combined with work-stealing scheduling. Similarly, our parallel programming abstraction also opted for the persistent task approach.

Both ompSs and OpenStream share two fundamental characteristics. The first is that both used fixed OpenMP versions with a fixed GCC compiler prototype. On the other hand, our solution allows the developer to use any GCC or OpenMP available in the system. The second characteristic is that OpenStream and ompSs adopted a different stream processing concept than ours. In their case, the programmer specifies a data-flow represented by a directed acyclic graph and lets the interface handle parallelism details. On the other hand, we adopted an execution flow with a structured and domain-specific vision for stream processing.

Related to heterogeneous architectures, some researches have been proposed to help abstract the process of developing code that targets both CPU and GPGPU devices [20, 21]. Like SPar, they employ a compiler to perform source-to-source code transformations. They used an OpenMP inspired methodology as a language extension. Unlike these works, our work did not consider GPGPU generation (in SPar this was done in [26]). However, since their CPU generation only considered default OpenMP, they are limited by the same problems we previously discussed in this paper.

In the field of parallel patterns, FastFlow [2] and Intel TBB [25] are C++ libraries that provide ready to use patterns as structured parallel programming solutions. To the programmer, they provide building blocks to instantiate ready-to-use parallel patterns (e.g., Map, Farm, and Pipeline). What they all share in common is that the programmer must know the implications of each pattern and the library syntax to manually implement parallelism. GrPPI [6] is an API for existing underlying runtime parallel systems. Although GrPPI does not focus only on stream processing applications, it does provide a Pipeline pattern. The patterns are deployed as C++ templates, which means the programmer must refactor the code into their specific API. Comparatively, the annotation approach adopted by SPar allows the programmer to maintain the original code structure and also abstracts from the programmer pattern syntax details. Besides, GrPPI requires the programmer to choose the best parallel pattern, while our programming abstraction takes advantage of the transformation rules to automatically generate a suitable parallel pattern. GrPPI could be considered as another runtime to SPar.

Advances in parallel patterns have been made in [5]. In it, the authors proposed patterns suitable for data parallelism and stream parallelism. These patterns were later implemented in FastFlow, which is SPar's default underlying parallel runtime system. Another abstract parallel programming API is Intel's CILK [17]. Its simplicity is well demonstrated by its only three keywords (cilk_spawn, cilk_sync, and cilk_for). However, similarly to OpenMP, it cannot support stream parallelism. As an extension, the work of [16] has proposed on-the-fly Pipeline parallelism for CILK as well as work-stealing scheduling. However, [16] also requires sequential code refactoring and its interface has not been updated since the publication date.

More closely related to SPar, StreamIt [27] is a stream DSL. StreamIt is more productive in expressing stream parallelism and communication than traditional C/C++ libraries. It has a straightforward and flexible structure that can be composed to create complex stream graphs without requiring significant modifications to the source code. However, StreamIt requires the programmer to learn new restrictive language syntax and semantics that are specific for their stream computation concepts. In contrast, we used standard C++ embedded into the programming language syntax.

Table 3 summarizes and compares this work with the main aforementioned related research. It was partially extracted from [8]. Highlighted in red is SPar, which is the interface used in this work to abstract OpenMP stream processing code. In the table, tools specify the methodology used to make the parallelism available and stream paradigm indicates if the runtime adopts a stream parallelism or data-flow approach for stream computation. Data-flow is a directed acyclic graph expressed computation where the runtime defines when and if to parallelize some code at a lower-level. On the other hand, stream parallelism uses a parallel pattern approach to perform its computations and specify an execution-flow. The parallelism shows if the shared-memory and communication mechanisms are explicit, implicit or abstract from the users perspective.

This work is the only one that fulfills all of the presented characteristics. Only this work, OpenMP, OpenStream and ompSS are annotation based. Another important aspect in Table 3 is parallelism. It can be abstract, where the programmer does

Table 3 Related work comparison

API	Tools	Stream paradigm	Target architecture	Parallelism
SPar	C++11 attributes	Stream parallelism	Multi-core	Abstract
StreamIt	New language	Stream parallelism	Multi-core and cluster	Abstract
OpenMP	C pragma	Data-flow	Multi-Core and accelerators	Explicit
OpenStream	C pragma	Data-flow	Multi-Core	Explicit
ompSS	C pragma	Data-flow	Multi-core and Accelerators	Explicit
Cilk-Piper	C/C++ language extension	Stream parallelism	Multi-Core	Explicit

not need to code with data oriented concerns since the interface handles these aspects, or explicit where the programmer must consider data characteristics in their design. In this case, only SPar and StreamIt are abstract. However, StreamIt requires the programmer to learn a new language while SPar uses C++ standard annotation mechanisms. Finally, most parallel runtime systems design goals focus at extracting maximum performance and productivity while ignoring abstractions. They require the programmer to understand at least the basics of data-flow specification or API definitions. They could potentially be used as future target underlying SPar runtime parallel systems.

In a nutshell, our scientific contribution is not SPar itself as it was already defined in the past [9]. Rather, we proposed an efficient stream Pipeline for OpenMP and implemented a new compiler algorithm for automatically generating parallel code based in this OpenMP Pipeline template. Besides, we created new definitions and transformation rules to leverage source-to-source code transformations and that can be further extended to support other non-structured parallel APIs besides OpenMP. So, we inherit all SPar characteristics, meaning that we are the only OpenMP solution that uses C++11 attributes for multi-core stream parallelism while focusing mainly on abstractions.

6 Conclusion

In this paper, we investigated an approach to simplify the development of OpenMP parallel stream processing applications employing source code annotations. We started by describing a template for OpenMP Pipeline processing and implementation requirements. From that, we used SPar's higher-level set of annotations to automatically generate the OpenMP Pipeline code. For that, we created and explained new SPar transformation rules for OpenMP stream processing that could be used in SPar for other non-structured parallel programming API's such as C++ threads. In

the experiments, we evaluated four stream processing applications comparing handwritten state-of-the-art code with SPar's automatically generated parallel code.

We observed that all SPar generated code performed only 2.49% less when compared to the handwritten solution. Additionally, it was possible to improve the execution time by up to 25.42%. Evaluating programmability aspects, we observed that SPar is the parallel programming API that requires the lowest amount of extra code. It is worth mention that our solution is limited to multi-core architectures. Additionally, our OpenMP Pipeline implementation performance might be degraded by lock contention. In the future, this could be addressed by using fine-grained lock access or employing lock-free multiple single producer single consumer queues. Other future works are enabling different parallel patterns to support big data applications and integrating other architecture support such as GPGPU or distributed systems.

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

References

1. Aldinucci M, Danelutto M, Kilpatrick P, Meneghin M, Torquati M (2011) Accelerating code on multi-cores with FastFlow. Euro-Par 2011 parallel processing, vol 6853. Lecture notes in computer science. Springer, Berlin, pp 170–181
2. Aldinucci M, Danelutto M, Kilpatrick P, Torquati M (2014) FastFlow: high-level and efficient streaming on multi-core. In: Programming Multi-core and Many-core Computing Systems, PDC, vol 1, p 14
3. Bienia C, Kumar S, Singh JP, Li K (2008) The PARSEC benchmark suite: characterization and architectural implications. In: 17th International Conference on Parallel Architectures and Compilation Techniques. PACT '08. ACM, Toronto, Ontario, Canada, pp 72–81
4. Bradski G (2000) The OpenCV library. Dr. Dobb's journal of software tools
5. Danelutto M, Matteis TD, Sensi DD, Mencagli G, Torquati M, Aldinucci M, Kilpatrick P (2019) The rephrase extended pattern set for data intensive parallel computing. Int J Parallel Progr 47:74–93. https://doi.org/10.1007/s10766-017-0540-z
6. del Rio Astorga D, Dolz MF, Sanchez LM, Blas JG, García JD (2016) A c++ generic parallel pattern interface for stream processing. In: Algorithms and Architectures for Parallel Processing, Springer International Publishing, Cham, pp 74–87
7. Gilchrist J (2004) Parallel Compression with BZIP2. In: 16th IASTED International Conference on Parallel and Distributed Computing and Systems. PDCS' 04. ACTA Press, MIT, Cambridge, USA, pp 559–564
8. Griebler D (2016) Domain-specific language and support tool for high-level stream parallelism. Ph.D. thesis, Faculdade de Informática - PPGCC - PUCRS, Porto Alegre, Brazil. http://tede2.pucrs.br/tede2/handle/tede/6776
9. Griebler D, Danelutto M, Torquati M, Fernandes LG (2017) SPar: a DSL for high-level and productive stream parallelism. Parallel Process Lett 27(01):1740005. https://doi.org/10.1142/S0129626417400059
10. Griebler D, Hoffmann RB, Danelutto M, Fernandes LG (2017) Higher-level parallelism abstractions for video applications with SPar. In: Parallel Computing is Everywhere, Proceedings of the International Conference on Parallel Computing, ParCo'17, IOS Press, Bologna, Italy, pp 698–707 https://doi.org/10.3233/978-1-61499-843-3-698

11. Griebler D, Hoffmann RB, Danelutto M, Fernandes LG (2018) High-level and productive stream parallelism for Dedup, Ferret, and Bzip2. Int J Parallel Progr 47(1):253–271. https://doi.org/10.1007/s10766-018-0558-x
12. Griebler D, Hoffmann RB, Danelutto M, Fernandes LG (2018) Stream parallelism with ordered data constraints on multi-core systems. J Supercomput 75(8):4042–4061. https://doi.org/10.1007/s11227-018-2482-7
13. Hoffmann RB, Griebler D, Danelutto M, Fernandes LG (2020) Stream parallelism annotations for multi-core frameworks. In: XXIV Brazilian Symposium on Programming Languages (SBLP), SBLP'20, ACM, Natal, Brazil, pp 48–55 https://doi.org/10.1145/3427081.3427088
14. ISO/IEC-14882:2011 (2011) Information technology - programming languages - C++. Technical report, International Standard, Geneva, Switzerland
15. Jacqueline FD, Buttlar BN (1996) PThreads programming. O'Reilly, Sebastopol, CA, USA
16. Lee ITA, Leiserson CE, Schardl TB, Zhang Z, Sukha J (2015) On-the-fly pipeline parallelism. ACM Trans Parallel Comput. https://doi.org/10.1145/2809808
17. Leiserson CE (2009) The cilk++ concurrency platform. In: Proceedings of the 46th Annual Design Automation Conference, DAC '09, Association for Computing Machinery, New York, NY, USA, pp 522–527 https://doi.org/10.1145/1629911.1630048
18. Mattson T, Sanders B, Massingill B (2004) Patterns for parallel programming, 1st edn. Addison-Wesley Professional, New York
19. McCool M, Reinders J, Robison A (2012) Structured parallel programming: patterns for efficient computation. Morgan Kaufmann Publishers Inc., San Francisco
20. Memeti S, Pllana S (2018) Hstream: a directive-based language extension for heterogeneous stream computing. In: 2018 IEEE International Conference on Computational Science and Engineering (CSE), pp 138–145 https://doi.org/10.1109/CSE.2018.00026
21. Nakao M, Murai H, Sato M (2019) Multi-accelerator extension in openmp based on pgas model. In: Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region, HPC Asia 2019, Association for Computing Machinery, New York, NY, USA, pp 18–25 https://doi.org/10.1145/3293320.3293324
22. OmpSs: The ompss programming model (2020). https://pm.bsc.es/ompss
23. OpenMP (2020) Open multi-processing api specification for parallel programming http://openmp.org/
24. Pop A, Cohen A (2013) Openstream: expressiveness and data-flow compilation of openmp streaming programs. ACM Trans Archit Code Optim. https://doi.org/10.1145/2400682.2400712
25. Reinders J (2007) Intel threading building blocks. O'Reilly, Sebastopol
26. Rockenbach DA, Griebler D, Danelutto M, Fernandes LG (2019) High-level stream parallelism abstractions with SPar targeting GPUs. In: Parallel Computing is Everywhere, Proceedings of the International Conference on Parallel Computing (ParCo), ParCo'19, IOS Press, Prague, Czech Republic, vol 36, pp 543–552 https://doi.org/10.3233/APC200083
27. Thies W, Karczmarek M, Amarasinghe S (2002) Streamit: a language for streaming applications. In: Horspool RN (ed) Compiler construction. Springer, Berlin, pp 179–196
28. Vogel A, Griebler D, Fernandes LG (2021) Providing high-level self-adaptive abstractions for stream parallelism on multicores. Software: practice and experience 51(6):1194–1217. https://doi.org/10.1002/spe.2948