October 2017

Higher-Level Parallelism Abstractions for Video Applications with SPar

Dalvan Griebler [a,1], Renato B. Hoffmann [a], Marco Danelutto [b], Luiz G. Fernandes [a]
[a] *Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre, Brazil*
[b] *Computer Science Department, University of Pisa, Italy*

Abstract. SPar is a Domain-Specific Language (DSL) designed to provide high-level parallel programming abstractions for streaming applications. Video processing application domain requires parallel processing to extract and analyze information quickly. When using state-of-the-art frameworks such as FastFlow and TBB, the application programmer has to manage source code re-factoring and performance optimization to implement parallelism efficiently. Our goal is to make this process easier for programmers through SPar. Thus we assess SPar's programming language and its performance in traditional video applications. We also discuss different implementations compared to the ones of SPar. Results demonstrate that SPar maintains the sequential code structure, is less code intrusive, and provides higher-level programming abstractions without introducing notable performance losses. Therefore, it represents a good choice for application programmers from the video processing domain.

Keywords. High-Level Parallel Programming, Stream Parallelism, Video Processing, Domain-Specific Language, C++11 Attributes

1. Introduction

For many years parallel computing has been mainly considered in specialized supercomputing centers. The situation dramatically changed in the last decade because of the many-core and multi-core architectures that are now available outside of high-performance computing centers. There are different challenges that need to be faced by application programmers to achieve performance and productivity in video streaming applications [4,18]. Compilers such as GCC are not able to automatically parallelize code from high-level C++ language abstractions [17]. Moreover, from the compiler's point of view, only limited cases of vectorized code can be automatically parallelized, while other higher-level code (viewed as coarse-grained code regions) do not provide the necessary semantic information for the compiler to perform code parallelization [10]. Consequently, developers are forced to restructure their applications by using low-level and architecture-dependent libraries to efficiently exploit parallelism.

On the other hand, traditional video applications have a predictable pattern of behavior. Video streams are generated from cameras or read from a video file. Then, a sequence of filters are used to improve quality, decode, detect objects, extract and write information, and/or use other custom filter types. Lastly, the results are reproduced in a screen or in an output video file. The most commonly used library for these applications is OpenCV [8], which already has data parallelism support for GPU and CPU architectures in some of its routines, using internally Threading Building Blocks (TBB) [15] and Compute Unified Device Architecture (CUDA) [13]. However, stream parallelism can only be achieved by using parallel programming frameworks that support pipeline pattern imple-

[1] Corresponding Author: dalvan.griebler@acad.pucrs.br

mentation. Examples are TBB and FastFlow [3], which are considered general purpose state-of-the-art alternatives and also support the implementation of stream parallelism. Although they simplify parallel programming through parallel patterns, application programmers still have to deal with low-level mechanisms and code re-factoring/rewriting, which we will discuss in detail below.

We designed SPar, a C++ internal Domain-Specific Language (DSL) for exploiting parallelism in streaming applications [10,11]. It address the problem of supporting the application programmer with higher-level and productive stream parallelism abstractions. Our primary design concern was to simplify the work of application programmers by enabling them to avoid performing sequential code rewriting and by only requiring them to introduce code annotations. SPar provides annotations to denote stream parallelism regions that the compiler may process to generate parallel code for multi-core architectures automatically. SPar is "de facto" a internal/embedded DSL as it preserves the original semantics of the host language by using the standard C++11 attributes mechanism [1]. In this paper, our goal is to assess SPar's abstractions and performance for traditional video applications. The main contributions are summarized as follows:

- We parallelize two traditional video applications with three parallel programming solutions (SPar, TBB, and FastFlow) and analysis of the implementation details.
- We compare and assess the performance, code intrusion, and higher-level parallelism abstractions of SPar with the one achieved with TBB and FastFlow.

This paper is organized as follows. Section 2 presents related works. Section 3 introduces SPar. Section 4 introduces the video streamming applications used in this work. Section 5 demonstrates the parallelization of video streaming applications. Section 6 discusses the performance experiments, code intrusion, and abstraction results. Lastly, Section 7 concludes the paper.

2. Related Work

Because this work is focused on video streaming, we present related work exploiting stream parallelism on multi-core architectures and C++ programs. Among them, FastFlow is a framework created in 2009 by researchers at the University of Pisa and the University of Turin in Italy [3]. It provides stream parallel abstractions adopting an algorithmic skeleton perspective and it is implementated on top of efficient fine grain lock-free communication mechanisms [2]. FastFlow was primarily used as the target of our SPar parallelism implementation, because it provides ready to use parallel patterns through high-level C++ template classes.

TBB (Threading Building Blocks)is an Intel C++ library for general purpose parallel programming. It emphasizes scalable and data parallel programming while abstracting the concept of threads through the concept of task. TBB builds on C++ templates to offer common parallel patterns (map, scan, parallel_for, among others) implemented on top of a work-stealing scheduler [15]. RaftLib [5] is a more recent C++ library designed to support both pipeline and data parallelism. It is based on the idea that the programmer implements sequential code portions as computing kernels, where custom split/reduce can be implemented when dealing with data parallelism. Moreover, there is a global online scheduler that can use OS scheduler, round-robin, work-stealing, and cache-weighted work-stealing. Yet, the communication between kernels is implemented by using lock-free queues. Similar to FastFlow and TBB, RaftLib results in a lower-level parallelism abstraction for the final application programmer than SPar. Therefore, all them are considered suitable runtimes for SPar.

SPar provides parallelism abstractions by using the C++ attributes annotation mechanism. Other researchers from the REPARA project[2] used the same mechanism. However, they did not produce a true DSLs and they focused on a different design methodology [6,9,16]. The general goals of REPARA are source code maintainability, energy efficiency, and performance on heterogeneous platforms. Annotations are provided with a parallel pattern semantic (farm, pipe, map, for, reduce, among others) to produce efficient parallel code targeting heterogeneous architectures including multi-core, GPU, and FPGA-based accelerators. Both the [6,7] present a methodology for introducing parallel patterns/data stream processing via code annotation as well as rules to implement parallelism using parallel programming frameworks. In contrast, SPar has its own compiler that pre-processes the attributes and generates automatic parallel code, mainly focused on stream parallelism. Other works [9,16] provide a methodology for generating REPARA annotations and detect parallel patterns in sequential source code. This methodology may be used for generating SPar annotations.

3. The SPar Domain-Specific Language

SPar [10,11,12] is an internal C++ DSL designed on top of the C++ attributes annotation mechanism [1]. It provides high-level parallelism abstractions that are close to the streaming application's domain vocabulary. An annotation is performed by using double brackets `[[id-attr, aux-attr, ..]]` that eventually specify a list of attributes. A SPar annotation is considered valid when at least the first attribute of the list is specified. The first attribute is an identifier (ID) while the others are auxiliary attribute (AUX). Table 1 presents all available attributes as well as their type and basic functionality.

Table 1. SPar Attributes

Attribute	**Type**	**Functionality**
ToStream	ID	Marks the stream region scope.
Stage	ID	Specifies a computational stage inside the stream.
Input(...)	AUX	Indicates the data consumed by a given stage.
Output(...)	AUX	Indicates the data produced for a subsequent stage.
Replicate(...)	AUX	Extends the degree of parallelism of a given stage.

The primary rule is that every `ToStream` code block must include at least one `Stage` annotation as illustrated in Figure 1. The stream management stage is represented by the code between `ToStream` and the first `Stage`, where the programmer generates and manages the end of the stream. For instance, by introducing a stop condition that breaks the loop. Also, this is the only piece of code inside a `ToStream` that can be left out of a `Stage` scope. The ID attributes can be used in the loop body to define its scope. Another restriction is to use `Replicate` only with a `Stage` to define the degree of parallelism. Note that it is up to the programmer to identify which stages can be safely replicated.

Unlike C++ meta-programming for DSL design and implementation, C++ attributes must be implemented at the compiler level. The SPar compiler is designed to recognize our DSL and to generate parallel code with proper calls targeting the lower level parallel programming library. It was developed using the CINCLE (A Compiler Infrastructure for New C/C++ Language Extensions) support tools [10]. The compiler parses the code (specified by a compiler flag called `spar_file`) and builds an AST (Abstract Syntax Tree) to represent the C++ source code. Subsequently, all code transformations are made directly in the AST, where proper calls to the FastFlow library will be introduced. Once

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

all SPar annotations are properly transformed, the compiler generates a file with C++ code and FastFlow calls, which is compiled by invoking the GCC compiler to produce a binary output.

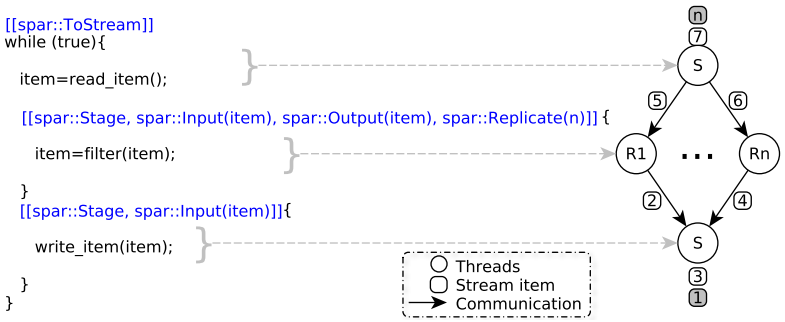


Figure 1. High-level representation of the SPar runtime parallelism.

In parallel code generation, SPar mainly uses the FastFlow's Farm and Pipeline objects. In Figure 1, the first stream region (code in between `ToStream` and the first `Stage`) and the last `Stage` are executed in two threads explicitly spawned for this purpose. The `Stage` that has the `Replicate` attribute (degree of parallelism support) spawns as many threads as the `Replicate` parameter that executes the same code portion. The underlying runtime system will then automatically distribute the stream items (specified through the `Input` and `Output` attributes) to these spawned threads. The communication is implemented using lock-free queues that are interconnected to adjacent stages. By default, the input items are distributed to the replicated `Stage` threads in a round-robin fashion and the stream order is not necessarily preserved in the output. This distribution is non-blocking, which means that the scheduler will be actively trying to put items in the queues (by default the queue size is set to 512). SPar assumes that stages have stateless operators when there is a `Replicate` attribute, while stateful operators must be manged by the programmer. Moreover, it supports other options that can be activated when desired (singly or combined) through the following compiler flags:

- `spar_ondemand`: generates an on-demand stream item scheduler by setting the queue size to one. Therefore, a new item will only be inserted in the queue when the next stage has removed the previous one.
- `spar_ordered`: makes the scheduler (on-demand or round-robin) preserve the stream items input/output order. FastFlow provides us a built-in function for this purpose so that SPar compiler simply generate proper calls to this functionality.
- `spar_blocking`: switches the runtime to behave in passive mode (default is active) blocking the scheduler when the communication queues are full. FastFlow offers a pre-processing directive so that the SPar compiler may easily support it.

4. Video Streaming Applications

Two typical video applications were studied and described in the following sections: one related to autonomous vehicles and one used to recognize people.

4.1. Lane Detection

This application is depicted in Figure 2. The $Capture(...)$ function generates an infinite sequence of image frames $F = \{f_1, ..., \infty \mid f$ is an image frame$\}$ from a camera device

and applies three computer vision algorithms. First, the $Segment(...)$ function splits each f_i into three horizontal parts to reduce and focus the processing on the lower area, where the street lane usually appears. Second, the application applies the $Canny(...)$ filter to detect edges, which computes $\delta_i = Canny(f_i)$, and produces $\Delta = \{\delta_1, ..., \delta_n \mid \delta_i$ is the result of $Canny(f_i)\}$ where n represents the maximum number of frames or ∞. Third, the application applies $HoughT(...)$ to detect straight lanes, where the potential of detecting these lanes is increased with the Canny filter. The $HoughT(...)$ algorithm receives each element of Δ and produces a δ_i and an λ_i subset of detected lanes l, where m is the number of lanes detected in each frame, $\lambda_i = \{l_1, ..., l_m \mid l$ is a detected lane in $HoughT(\delta)\}$ and $\Lambda = \{\lambda_1, ..., \lambda_n \mid \lambda$ is a subset of lanes$\}$.

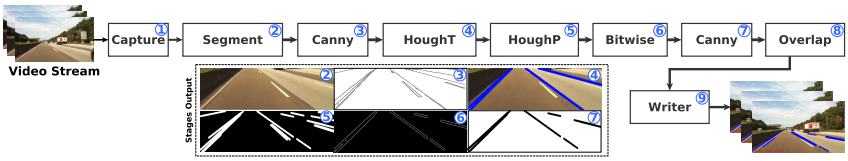


Figure 2. Lane detection application workflow.

Since $HoughT(...)$ does not detect the lane extremities, this application also applies the Probabilistic Hough Transform in fourth step, giving the beginning and end of the detected lanes. The $HoughP(...)$ function also receives elements of Δ, but produces another set of lanes called Ω, where $\Omega = \{\omega_1, ..., \omega_o \mid \omega$, which are the result of $HoughP()\}$ and o represent the number of elements in Ω. In the fifth step, the $Bitwise(...)$ operation is applied to each δ_i element to use its common lanes in Λ and Ω. Thus, the resulting lanes Γ_i for each frame δ_i are produced taking into account that $\Gamma = \{\omega \mid \forall\lambda \in \Lambda, \forall\omega \in \Omega, \lambda = \omega\}$. Finally, each Γ_i element (lane) is overlapped with δ_i (frames) to produce the image with the detected lanes and the $Writer(...)$ function writes it to a file.

4.2. Person Recognition

The Person Recognition application workflow is illustrated in Figure 3. First of all, it receives a sequence of frames $F = \{f_1, ..., \infty\}$ from the $Capture(...)$ operation. Then, all f elements are processed by $Detector(...)$, which produces a set of faces $\Delta = \{\delta_1, ..., \delta_m\}$ where m corresponds to the total number of faces detected in a single f element of F. All detected faces are outlined with a red circle as demonstrated in step ❷ of Figure 3.

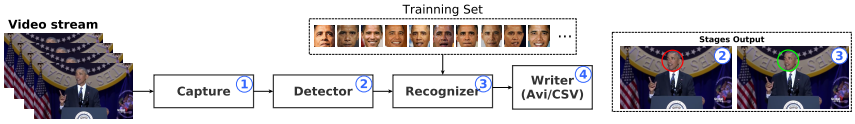


Figure 3. Person recognition application workflow.

The $Recognizer(...)$ step uses the previous set of image faces (called training set $T = \{t_1, ..., t_n \mid t$ is a training face image $\}$) in order to compare all Δ (the faces detected in f) to T (the faces training set). The result of the $Recognizer(...)$ is $D = \{d \mid d \in T, d \in \Delta\}$. For each element of D, the application outlines this position in f with a green circle as demonstrated in step ❸ output in Figure 3. Lastly, the $Writer(...)$ function write in the output the frames and the information about the faces into two different file formats (CSV and AVI).

5. Parallelization

This section discusses the parallelization of Lane Detection and Person Recognition applications, using SPar, TBB, and FastFlow. We first used the Lane Detection application (explained in Section 4.1) as a baseline example (Listing 1). This is a high-level description of the actual source code in order to demonstrate the extra code introduced by each one of the tools. Our parallel code modeling strategy was the same for the two applications. There were three pipeline stages, where all independent operations were put together in a single stage that was replicated as many times as necessary. We also tried to model the application with more stages, but the overall performance was decreased due to communication overhead.

SPar's Lane Detection parallel implementation is presented in Listing 2. Although we have identified that *Segment*(), *Canny*(), *HoughT*(), *HoughP*(), *Bitwise*(), and *Overlap*() may compute in parallel, we have opted to maintain them in a single sequential function in order to avoid load imbalance. When comparing the SPar and Sequential versions, we can observe that the source code structure is maintained. As we can see in Listing 2, we only had to add annotations in lines 3, 5 and 6 to support parallelism.

```
1| int main(){
2|  v = Open(video);
3|  while(!v.eof()){
4|    f = Capture(v);
5|    f = Overlap(Canny(
 |       Bitwise(HoughP(
 |       HoughT(Canny(
 |       Segment(f)))))))
 |       ;
6|    Write(f);
7|  }
8| }
```

Listing 1 Sequential.

```
1| int main(){
2|  v = Open(video);
3|  [[spar::ToStream,spar::Input(v)]] while(!v.
 |       eof()){
4|    f = Capture(v);
5|    [[spar::Stage,spar::Input(f),spar::Output(
 |       f),spar::Replicate()]]{ f = Overlap(
 |       Canny(Bitwise(HoughP(HoughT(Canny(
 |       Segment(f)))))));  }
6|    [[spar::Stage,spar::Input(f)]]{Write(f);}
7|  }
8| }
```

Listing 2 SPar.

The stream communication between the stages is handled by the `Input` and `Output` annotations, therefore the programmer must only identify and indicate these data dependencies. Moreover, SPar's runtime system also handles thread and queue management. However, lane Detection requires the original stream order to be maintained in the last stage so that the output file maintains a correct frame sequence. In SPar, this can be achieved with the addition of the `-spar_ordered` compiler directive.

FastFlow and TBB require programmers to restructure the sequential code by using the Farm and Pipeline parallel patterns, as presented in Listing 3 and 4. These two frameworks look similar in terms of code re-factoring and expressiveness. While TBB only offers Pipeline, FastFlow supports both parallel patterns. Although FastFlow allows us to combine Pipeline with Farm, it already provides us the most efficient and less intrusive parallelization by using only the Farm. In Listing 3, the main details for FastFlow's implementation version are presented. We used *Capture*() inside the emitter (the input frame stream scheduler), *Writer*() in the collector (gathering/reducing results from the workers), and the sequence of filters were assigned to the Farm workers. After building the classes, we created a vector of worker replicas (`S2`) and initialized the Farm template to preserve the order of the stream items (`ff_OFarm`). Also, we distinguished between the collector and emitter classes (lines 29 and 31) to run Farm (line 32). Note that FastFlow requires implementing the business logic code inside the `svc` method, which is a virtual member function of the `ff_node` class. The runtime calls this method and when it returns, the pointer is stored in a lock-free queue managed by FastFlow runtime.

```
1| struct td { /*set of data*/ };
2| class S1:public ff_node_t<td> {
3| td* svc(td*){
4|  while(!v.eof()){
5|   f = Capture(v);
6|   td * t = new td(f, ...);
7|   ff_send_out(t);
8|  }
9|  return EOS;
10| }};
11| class S2:public ff_node_t<td> {
12| td* svc(td* t){
13|  t->f = Overlap(Canny(Bitwise(
  |      HoughP(HoughT(Canny(Segment
  |      (t->f)))))));
14|  return t;
15| }};
16| class S3:public ff_node_t<td> {
17| td* svc(td* t){
18|  Write(t->f);
19|  delete t;
20|  return GO_ON;
21| }};
22| int main(){
23|  v = Open(video);
24|  std::vector<unique_ptr<ff_node
  |      >> workers;
25|  for(int i=0; i < nthreads; i
  |      ++)
26|   workers.push_back(std::
  |      make_unique<S2>(...));
27|  ff_OFarm<td> ofarm(move(
  |      workers));
28|  S1 E(v, ...);
29|  ofarm.setEmitterF(E);
30|  S3 C(...);
31|  ofarm.setCollectorF(C);
32|  ofarm.run_and_wait_end();
33| }
```

Listing 3 FastFlow.

```
1| struct td { /*set of data*/ };
2| class S1:public filter{
3| S1(...):..., filter(filter::
  |      serial_in_order){}
4| void* operator()(void*){
5|  while(!v.eof()){
6|   f = Capture(v);
7|   td * t = new td(f, ...);
8|   return t;
9|  }
10|  return NULL;
11| }};
12| class S2:public filter{
13| S2(...):..., filter(filter::parallel){}
14| void* operator()(void*in){
15|  td * t=static_cast<td*> (in);
16|  t->f = Overlap(Canny(Bitwise(HoughP(
  |      HoughT(Canny(Segment(t->f)))))));
17|  return t;
18| }};
19| class S3:public filter {
20| S3(...):..., filter(filter::
  |      serial_in_order){}
21| void* operator()(void*in){
22|  td * t=static_cast<td*> (in);
23|  Write(t->f);
24|  delete t;
25|  return NULL;
26| }};
27| int main(){
28|  v = Open(video);
29|  pipeline p;
30|  S1 s1(v, ...);
31|  p.add_filter(s1);
32|  S2 s2(...);
33|  p.add_filter(s2);
34|  S3 s3(...);
35|  p.add_filter(s3);
36|  task_scheduler_init init_parallel(
  |      nthreads);
37|  p.run(nthreads);
38| }
```

Listing 4 TBB.

For the TBB parallelization (Listing 4), we used a data structure because the `operator()` method supports only one argument. The idea is similar to FastFlow. In contrast, the classes are modeled using the particular `operator()` member function and specifying the filter type in the class constructor. To maintain the order of the stream items, *Capture*() and *Write*() are necessary `serial_in_order` while the sequence of filters are `parallel`. The communication between classes occurs through pointers that are stored in a global queue when a pointer returns inside `operator()`. The final step is to initialize the pipeline, add the stage classes, and execute the pipeline (line 37). The Person Recognition application followed a similar code structure in the Lane Detection for SPar, TBB, and FastFlow. In this second case, *Capture*() and *Write*() were made the first and last stages while the middle stage processed *Detector*() and *Recognizer*() in sequence over distinct f from $F = \{f_1, ..., \infty\}$. These two applications differ in the sequence of filters applied over the stream items and workloads. The next section discusses the differences regarding performance, source lines of code, and cyclomatic complexity.

6. Experiments

Our experiments were carried out to assess parallel programming abstractions and performance of SPar (`spar`), TBB (`tbb`), and FastFlow (`ff`) for Lane Detection and Person Recognition applications. The experiments were performed with the default parallelization presented in Section 5 and the combination of the SPar compilation flags: `-spar_blocking` (`spar-blk`), `-spar_ondemand` (`spar-ond`), and their combinations (`spar-ond-blk`). We measured the Cyclomatic Complexity Number (CCN) [14] and the Source Lines of Code (SLOC) to evaluate code intrusion. These metrics give us valuable insights regarding the amount of coding needed and additional complexity to support parallelism in the applications.

Instead of reading from a camera device, we adapted the code to read from a video file to test our parallelizations. We used a MPEG-4 video of 5.25MB (640x360 pixels) for Lane Detection and a MPEG-4 video of 1.36MB (640x360 pixels) along with a training set of 10 image faces of 150x150 pixels for Person Recognition. The machine used was equipped with 24GB of RAM memory and a dual socket Intel(R) Xeon(R) CPU E5-2620 v3 2.40GHz (24 threads with Hyper-Threading). The operating system was Ubuntu Server 64 bits with kernel 4.4.0-59-generic. We used the following tools: GCC 5.4.0, libraries TBB (4.4 20151115), FastFlow (revision 13) and OpenCV (2.4.9.1). We compiled using the -O3 flag. All values represent averages of 10 different testing cycles.

6.1. Programming Comparison

As presented in Section 5, both Fastflow and TBB required the definition of data structures, pointers management, and code re-factoring to use the parallel patterns available as C++ templates. On the other hand, SPar preserved the source code semantics by only requiring the insertion of standard C++ annotations in the correct place. Another advantage is that SPar only uses five attributes, which are flexible enough to model different pipeline compositions. Also, three compiler flags can support different runtime system behavior as the programmer sees fit. Thus, because these applications are already structured, SPar can be implemented without increasing complexity. SPar implementation of these applications may be easily derived from code such as the one of Listing 1. In addition, we provide a quantitative analysis of the parallelized application versions using software engineering metrics such as CCN and SLOC, in Table 2 (higher numbers entail more intrusive and complex coding).

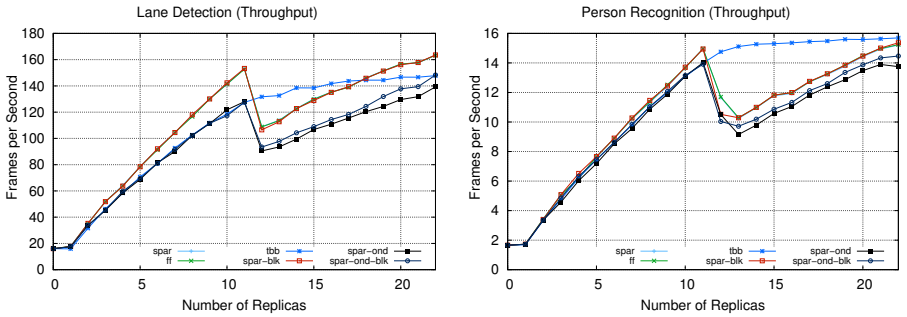
Table 2. Programming metrics.

	(a) Lane Detection		**(b) Person Recognition**	
Implementation	***SLOC***	***CCN***	***SLOC***	***CCN***
Sequential	100	14	100	12
SPar	104	14	106	12
Fastflow	138	22	148	22
TBB	144	22	149	20

For all implemented versions in both applications, SPar provides lower SLOC and CCN when compared to TBB and FastFlow. SPar does not increase the CCN with respect to the sequential code because there are not significant code intrusions or re-factoring. In contrast, CCN increases when using TBB and FastFlow due to the their classes and methods. These results evidence the higher-level parallelism abstractions of SPar with respect to TBB and FastFlow, where programmers may reduce the learning curve due to SPar's domain-oriented approach, simpler code syntax, and on-the-fly stream parallelism implementation.

6.2. Performance Comparison

The graphs in Figures 4(a) and 4(b) present the throughput achieved by Lane Detection and Person Recognition. We obtained an average of 0.85% standard deviation for Lane Detection, and 0.54% for Person Recognition. The sequential version is plotted as the 0 replica position of the *X* axis. The one replicate point is where the parallel code starts varying the degree of parallelism. In both applications, the parallelization provided good scalability before reaching hyper-thread resource usage. Then, the throughput decreases with SPar and FastFlow versions (identical results in the two). This occurred because the FastFlow runtime scheduler implements the default ordering and this significantly impacts the performance due to its limitation of ondemand scheduling. We can see this even more clearly in the Person Recognition application. TBB continues to increase the performance, but it is not able to reach the best SPar and FF performances. We only noted a small performance overhead in TBB for Lane Detection when using hyper-threading, which is expected in applications where threads compete for the ALU usage.



(a) Results of the Lane Detection versions.

(b) Results of the Person Recognition versions.

Figure 4. Performance comparison.

SPar generates customized FastFlow code. All parallelized versions can be implemented manually using the FastFlow library. Only the equivalent default version of SPar was implemented with FastFlow and plotted on the graphs. In Figure 4(a) and 4(b) we can see that SPar and FastFlow achieved identical results. We thus were able to conclude that the SPar's high-level abstraction did not affect the performance of these applications, because the parallel code generated by SPar is very similar to manually tuned code in FastFlow. In theory, every streaming application developed with SPar could achieve identical performance compared to FastFlow. Moreover, the default scheduling of SPar and FastFlow yields better results than TBB before using hyper-threading. It is better when the items are balanced such as in the Lane Detection (see Figure 4(a)). The lower performance in Lane Detection TBB is justified by its runtime scheduler that dynamically works on a global queue, thereby not necessary being suited for these kind of applications. However, in the Person Recognizer, TBB provides better load balancing when using hyper-threading resources due to its dynamism.

7. Conclusions

This paper presented an assessment of SPar for parallelizing traditional video applications in comparison with other parallel programming frameworks (TBB and FastFlow). The experiments were performed to evaluate performance, CCN, and SLOC of these par-

allel versions, discussing aspects of the implementation, runtime behavior, and code intrusion. We concluded that SPar provides a good trade-off between high-level parallelism abstractions and performance for application programmers for these tested applications. Also, SPar may be used by application programmers to easily introduce parallelism in sequential legacy codes, especially for video applications that extend the OpenCV library. Future works will explore and evaluate how SPar can be used in combination with the internal parallelism support of OpenCV for GPUs. In addition, we plan to parallelize a set of video applications that extend other computer vision features such as deep-learning.

Acknowledgements
Authors would like to thank the partial financial support from CAPES and FAPERGS Brazilian research institutions. Moreover, this work has also received partial financial support from the EU H2020-ICT-2014-1 project RePhrase (No. 644235).

References

[1] 14882:2014-ISO/IEC. Information Technology - Programming Languages - C++. Technical report, International Standard, Geneva, Switzerland, December 2014.

[2] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati. An Efficient Unbounded Lock-Free Queue for Multi-core Systems. In *Euro-Par Parallel Processing*, volume 7484, pages 662–673, Rhodes Island, Greece, August 2012. Springer.

[3] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati. FastFlow: High-Level and Efficient Streaming on Multi-core. In *Programming Multi-core and Many-core Computing Systems*, volume 1 of *PDC*, page 14. Wiley, March 2014.

[4] H. C. M. Andrade, B. Gedik, and D. S. Turaga. *Fundamentals of Stream Processing*. Cambridge University Press, New York, USA, 2014.

[5] J. C. Beard, P. Li, and R. D. Chamberlain. RaftLib: A C++ Template Library for High Performance Stream Parallel Processing. In *6th Inter. Works. Progr. Models and App. for Multicores and Manycores*, PMAM' 2015, pages 96–105, San Francisco, USA, Febuary 2015. ACM.

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

[7] M. Danelutto, T. D. Matteis, G. Mencagli, and M. Torquati. Data Stream Processing Via Code Annotations. *The Journal of Supercomputing*, pages 1–15, 2016.

[8] S. Datta. *Learning OpenCV 3 Application Development*. Packt, 2016.

[9] D. del Rio Astorga, M. F. Dolz, L. M. Snchez, J. D. Garca, M. Danelutto, and M. Torquati. Finding Parallel Patterns Through Static Analysis in C++ Applications. *The International Journal of High Performance Computing Applications*, 2017.

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

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

[12] D. Griebler, M. Danelutto, M. Torquati, and L. G. Fernandes. SPar: A DSL for High-Level and Productive Stream Parallelism. *Parallel Processing Letters*, 27(01):20, March 2017.

[13] D. B. Kirk and W. mei W. Hwu. *Programming Massively Parallel Processors*. Morgan Kaufmann, 2013.

[14] L. M. Laird and M. C. Brennan. *Software Measurement and Estimation: A Practical Approach*. Wiley-IEEE Computer Society Pr, 1st edition, 2006.

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

[16] R. Sotomayor, L. M. Sanchez, J. G. Blas, J. Fernandez, and J. D. Garcia. Automatic CPU/GPU Generation of Multi-versioned OpenCL Kernels for C++ Scientific Applications. *International Journal of Parallel Programming*, 45(2):262–282, 2017.

[17] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *ACM Trans. on Embe. Comp. Sys. (TECS)*, 13(4):25, 2014.

[18] W. Thies and S. Amarasinghe. An Empirical Characterization of Stream Programs and Its Implications for Language and Compiler Design. In *Inter. Conf. on Par. Arch. and Compil. Tech.*, PACT '10, pages 365–376, Austria, September 2010. ACM.