Int J Parallel Prog (2019) 47:253–271
https://doi.org/10.1007/s10766-018-0558-x

High-Level and Productive Stream Parallelism for Dedup, Ferret, and Bzip2

Dalvan Griebler[1] · **Renato B. Hoffmann**[1] · **Marco Danelutto**[2] · **Luiz G. Fernandes**[1]

Received: 22 September 2017 / Accepted: 20 January 2018 / Published online: 1 February 2018

Abstract Parallel programming has been a challenging task for application programmers. Stream processing is an application domain present in several scientific, enterprise, and financial areas that lack suitable abstractions to exploit parallelism. Our goal is to assess the feasibility of state-of-the-art frameworks/libraries (Pthreads, TBB, and FastFlow) and the SPar domain-specific language for real-world streaming applications (Dedup, Ferret, and Bzip2) targeting multi-core architectures. SPar was specially designed to provide high-level and productive stream parallelism abstractions, supporting programmers with standard `C++-11` annotations. For the experiments, we implemented three streaming applications. We discussed SPar's programmability advantages compared to the frameworks in terms of productivity and structured parallel programming. The results demonstrate that SPar improves productivity and provides the necessary features to achieve similar performances compared to the state-of-the-art.

Keywords High-level parallelism · Parallel programming · Stream processing · Parallel patterns · Pipeline parallelism · Streaming applications

1 Introduction

Parallel programming has been intensively studied to provide higher-level abstractions for both application and system programmers. The separation of these two concerns is important since both scenarios have different requirements and perspectives. Usually

✉ Dalvan Griebler
dalvan.griebler@acad.pucrs.br

[1] Faculty of Informatics, Pontifical Catholic University of Rio Grande do Sul, Porto Alegre, Brazil

[2] Computer Science Department, University of Pisa, Pisa, Italy

the system programmer intends to extract the best performance from the parallel architecture, and he is therefore concerned with architecture-dependent code optimization and flexibility. The application programmer is more focused on developing the application and may take advantage of the parallelism provided by the target architecture. Moreover, he is not expert in system level programming or parallelism strategies.

Parallel design patterns and algorithmic skeletons were proposed to help application programmers in the difficult task of parallel programming [5,15,16]. Although new tools have emerged over the years based on these approaches, there is still a lack of more suitable and friendly alternatives for the application programmer. In this context, the design pattern methodology does not separate the concerns discussed above, requiring application programmers to learn with manuals designed for system programmers. In contrast, the algorithmic skeleton frameworks/libraries abstract several system details, supporting the programmer with flexible and efficient solutions and high-level skeleton abstractions (see for instance FastFlow [2], GrPPI [7], and TBB [19]). However, they lack coding productivity, because the application programmer must restructure the business logic code of these programs and be aware of several details related to the library's constraints.

An alternative is to provide specialized parallel programming interfaces for a specific application domain [9,22,23]. This approach aims to increase productivity as well as provide higher-level abstractions, more friendly interface, and simpler environments [9]. The state-of-the-art example is Streamit [23], which is a programming language designed to express parallel data stream processing applications. It has a unique language that is independent of the target architecture (cluster and multi-core). The compiler can automatically transform to C or Java code based on the needs of distinct hardware devices [21,24]. StreamIt is not considered a robust language [23], but it is much more productive for exposing parallelism and communication than traditional C/C++ libraries. However, when dealing with legacy code written in robust and general purpose programming languages like C/C++ and Java, the programmer must rewrite the code into StreamIt language.

Recently, a new way to exploit stream parallelism was proposed through SPar [11], which is an internal C++ Domain-Specific Language (DSL). It was designed to support application programmers with friendly code annotations, which are expressed with the standard `C++-11` attributes [12]. The SPar's compiler interprets the annotations and performs a source-to-source transformation, producing a C++ parallel code by using the FastFlow library. Although this idea was designed for C++, annotation mechanisms are also present in other robust languages like Java, which makes this approach feasible for other languages. Though it is hard to compare StreamIt and SPar's productivity due to their distinct design goals, they provide similar benefits. Whereas SPar preserves the sequential C++ code structure, StreamIt provides a simpler syntax than the C/C++ language.

In this paper, the goal is to assess the programmability and performance of the SPar DSL for robust streaming applications such as Dedup, Ferret, and Bzip2. Moreover, we will compare it with state-of-the-art implementations to highlight the main differences and provide new insights and analysis. Consequently, we are making the following contributions:

- We implement support for high-level stream parallelism in Dedup, Ferret, and Bzip2 applications using the SPar DSL.
- We provide a comparative analysis of performance and programmability among SPar, Pthreads, TBB, and FastFlow for Dedup, Ferret, and Bzip2 applications.
- We discuss the seamlessness of refactoring structured parallel code in these three applications when using SPar.

This paper is organized as follows. Section 2 discusses, compares, and contrasts this work with related works. In Sect. 3, we present the basis of the SPar DSL features. Next, in Sect. 4 we demonstrate how these three applications were implemented with SPar, highlighting the structured parallel programming aspects related to the high-level stream parallelism abstractions. Section 5 provides useful insights regarding the programmability and performance of SPar compared to other frameworks. Finally, Sect. 6 concludes our work and presents future research.

2 Related Work

In the literature, different state-of-the-art implementations of Dedup, Ferret, and Bzip2 have been implemented and tested. The majority have concentrated on evaluating a single parallel programming frameworks (e.g., TBB and FastFlow). The novelty of our work is that it adds the implementation of the SPar DSL programming model and provides a performance and programmability analysis that extends the discussions of previous studies towards a structured parallel programming approach.

Dedup and Ferret applications have been mainly analyzed and studied by Navarro et al. [17] and Reed et al. [18]. Initially, Navarro et al. [17] created an analytical model for pipeline parallelism based on queuing theory in order to characterize performance and efficiency. As a use case, they applied their model for Dedup and Ferret, using Pthreads and TBB. In their experiments, they found load imbalance and I/O bottlenecks, which were solved by collapsing stages and implementing dynamic scheduling. Later, Reed et al. [18] provided an implementation of pipeline parallelism using different TBB constructs, demonstrating what successes and failures. They especially targeted Dedup, Ferret, and x264 PARSEC applications. They concluded that it is not possible to parallelize x264 with the default TBB constructions. Also, the TBB version of Dedup achieved better performance than Pthreads, and for Ferret the results were similar to Pthreads. In contrast to these works ([17] and [18]), we approach high-level stream parallelism using SPar, analyze and compare programmability, structured parallel programming aspects, and performance.

The study of Chasapis et al. [4] evaluated and applied task-based strategies on 10 PARSEC applications by using the OmpSs programming model. They compared their implementation concerning performance and programmability (lines of code) to the original POSIX threads implementation. Instead, we are concentrating only on streaming applications such as Dedup and Ferret from PARSEC, we added Bzip2 in our analysis and compared our approach (SPar) with others beyond the original POSIX threads version. We also considered the Cyclomatic Complexity Number (CCN) and structured code refactoring.

In [14], an extension for the Cilk programming model was proposed, named Cilk-P. They designed it to express on-the-fly pipeline parallelism through pre-processing compiler directives. The central point of the investigations was an efficient scheduling algorithm integrated into a work-stealing scheduler runtime. The experiments were carried out on Dedup, Ferret, and x264 PARSEC benchmarks, compared to Pthreads and TBB versions. In contrast, our research focuses more on the programmability aspects and high-level parallelism abstractions than efficient runtime design. Also, our performance analysis included the Bzip2 application and FastFlow implementation.

Recently, the research of Danelutto et al. [6] has presented a different perspective. The goal was to introduce the idea of pre-built parallel patterns that can be easily instantiated by the programmer. Therefore, they proposed P^3ARSEC, a benchmark suite for parallel pattern-based frameworks consisting of five PARSEC applications parallelized with the FastFlow library. Their work also analyzed performance and programmability with respect to the original POSIX thread version of PARSEC. On the other hand, we have focused on high-level stream parallelism aspects of Dedup and Ferret, and compared their implementation to our version, which in turn is more productive and has a better CCN. Moreover, we considered the Bzip2 implemented in [1] in our analysis, using FastFlow in order to compare it with SPar, TBB, and Pthread versions.

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

SPar is an internal DSL embedded in the C++ language, capable of modeling high-level parallelism for streaming applications [9–11]. It was implemented with the standard C++ attribute annotation mechanism [12]. The programmer only needs to introduce annotations in the sequential source code rather than having to actually rewrite it to exploit the parallelism available on multi-core systems. The following sections will describe the SPar language, skeletal library, and runtime.

3.1 SPar Language

When using SPar, the programmer will find user-friendly abstractions that are closer to the streaming application domain's vocabulary. All properties are represented by language attributes in the annotated regions. An annotation is expressed by using double brackets `[[id-attr, aux-attr, ..]]`, where a list of attributes can be specified if necessary. When at least the first attribute of the annotation is specified in the attributes' list, it is considered a SPar annotation. We named the first attribute identifier (ID) and the others auxiliary (AUX). The full description of SPar's available attributes can be found in [9]. In summary, `ToStream` and `Stage` are classified as IDs whereas `Input`, `Output`, and `Replicate` are AUXs.

In Listing 1, we can appreciate SPar's usability with the Prime Numbers algorithm. Semantically, each `ToStream` must have at least one `Stage` annotation. In SPar, the code between `ToStream` and the first `Stage` becomes a stream management stage, where the programmer needs to explicitly manage the full stream, including the end of the stream. We can easily achieve this by introducing a stop condition that breaks the loop. In the case of Listing 1, the end of stream is dictated by the annotated loop in line 3. Also, this is the only piece of code inside a `ToStream` region that can be left out of the `Stage` scope limits. Both the ID attributes can use the loop body to define their scope as we performed the annotation in line 5. Another restriction is that the `Replicate` can only be used with `Stage`.

```
 1 int prime_numbers(int n){
 2   int total=0;
 3   [[spar::ToStream, spar::Input(n)]] for(int i=2; i<=n; i++){
 4     int prime=1;
 5     [[spar::Stage, spar::Input(i,prime), spar::Output(prime), spar::Replicate(workers)]] for (int j
         =2; j<i; j++){
 6       if (i%j==0){ prime=0; break; }
 7     }
 8     [[spar::Stage,spar::Input(prime),spar::Output(total)]]
 9     { total=total+prime; }
10   }
11   return total;
12 }
```

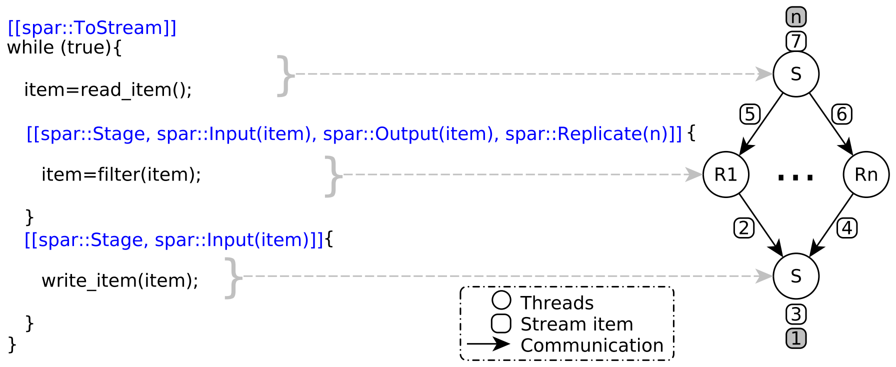
Listing 1 Prime numbers algorithm annotated with SPar

The SPar compiler is designed to recognize our language and generate parallel code. It was developed by using the CINCLE (A Compiler Infrastructure for New C/C++ Language Extensions) support tools [9]. The compiler parses the code (which is specified by a compiler flag named `spar_file`) and builds an AST (Abstract Syntax Tree) to abstractly represent the C++ source code. Subsequently, all code transformations are made directly in the AST, where calls are generated to the FastFlow library. Once all SPar annotations are properly transformed, another C++ code is generated, which is compiled by invoking the GCC compiler to produce a binary output. The next section presents further details on the functioning of the runtime parallelism.

3.2 SPar Runtime Parallelism

As aforementioned, SPar takes advantage of FastFlow to generate parallel code support. SPar mainly uses the Farm and Pipeline interfaces and customizes them to meet particular needs. Figure 1 is a high-level representation of a given annotated code (left hand side) with the respective runtime parallelism behavior (right hand side). Note that the annotated code is reading stream items infinitely, which are subsequently filtered and then written to the standard output. On the right hand side of Fig. 1, the code between `ToStream` and the first `Stage` annotation is a single process running this portion of code.

In SPar, the `Stage` that has a `Replicate` attribute corresponds to spawning many threads with the same code portion (see first stage Fig. 1). The underlying runtime system will abstractly distribute stream items (specified through the `Input` and `Output` attributes) to these spawned threads, which have a lock-free communication queue connected to the previous stage. If there are incorrect or unspecified parameters, SPar will report a compiler error. By default, the items are distributed in a round-robin fashion and input/output ordering is not preserved. This distribution is non-blocking, which means that the scheduler will be actively trying to put items in the queues (by



```
[[spar::ToStream]]
while (true){

  item=read_item();

   [[spar::Stage, spar::Input(item), spar::Output(item), spar::Replicate(n)]] {

    item=filter(item);

  }
  [[spar::Stage, spar::Input(item)]]{

    write_item(item);

  }
}
```

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

default the queue size is 512). The last `Stage` is a single thread, which has a queue connected to the previous stage. Observe that it is up to the programmer not to replicate the stages corresponding to stateful operators. SPar is not able to guarantee sequential code equivalence if the programmer makes a mistake or does something wrong. SPar also supports other options through compiler flags that can be activated when desired (individually or combined) as follows:

- `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 order of the stream items. FastFlow provides us a built-in function for this purpose so that the SPar compiler can simply generate it.
- `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 this feature.

4 Streaming Applications

In this section, we describe real-world streaming applications that commonly run on multi-core systems. We detail our challenges to introduce SPar annotations and demonstrate the ease and productivity of our DSL for programmers.

4.1 Dedup

Dedup is a PARSEC application designed to compress data streams based on the deduplication method. It combines local and global compression to achieve high compression ratios [3]. The original Pthreads implementation is based on the pipeline parallel pattern, using five different stages and communicating through queues with fixed sizes. Figure 2 (left hand side) illustrates the activity graph of the original Dedup

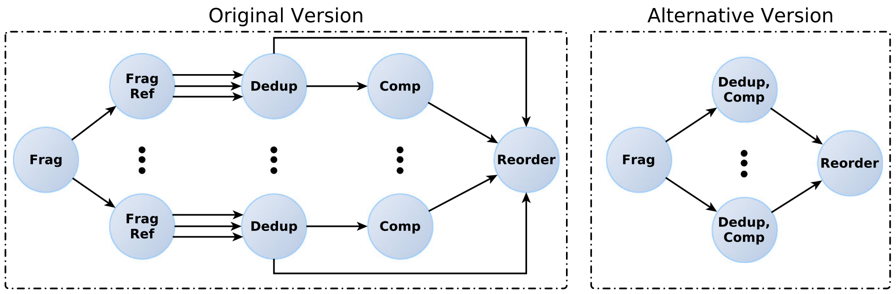


Fig. 2 Activity graph of the Dedup implementation versions

implementation. The first and last stage are performing a single thread input and output operations. The three middle stages have a thread pool, where the middle one uses lock mechanisms to avoid access conflicts in concurrent hash table entries. Each stage can be described as follows:

- *Fragment* Here, the data stream is read and partitioned into coarse-grain chunks so that the rest of the stage processes them individually.
- *Fragment Refine* This stage receives a coarse-grain chuck and further partitions it into finer grain chunks. Consequently, it produces smaller chunks for the next stages.
- *Deduplicate* This stage analyzes if there are duplicated chunks so that they can be eliminated and only unique chunks are stored in the hash table.
- *Compress* This stage compresses the unique chunks.
- *Reorder* This stage writes the compressed chunks into a file and also reorders the chunks as they are received to match the original file.

Two particularities distinguish this pipeline from a traditional one: the *Fragment Refine* stage receives only one input and generates several outputs and the *Compress* stage is bypassed if the chunk is duplicated. Also, because the output order must be equivalent to the input and there is no determinism in the parallel processing, a reordering algorithm for the stream items was implemented for the Pthread version in the *Reorder* stage. This algorithm reconstructs the stream order according to two different identifier (ID) numbers attached to the chunk data structure in the two fragmentation stages. A search tree is used as a reordering structure for the first (*Fragment*) fragmentation level and a heap-based data structure for the second (*Fragment Refine*). Chunks that are received in order are promptly written, while those that are out of order are stored in the reordering structures so that they can be removed later and/or written in the output file [3]. This additional code is not required by the sequential version, which will never process a new chunk without having already written the previous one. More importantly, SPar offers a compiler flag that automatically abstracts and guarantees the stream order (`spar_ordered`).

For our SPar implementation, we started from the sequential code version. However, we used the same Pthreads lock mechanisms that were in the auxiliary function of the *Deduplication* stage to prevent problems with concurrent access to the hash table entries. The *Fragment Refine* stage is a Pthreads optimization. Although it was implemented with SPar initially, we have opted to not include it in the final version primarily because the results of our experiments showed that it slightly degrades performance. Therefore, the alternative version we present on the right hand side of Fig. 2 has coarser-grain chunks, which is partially derived by refactoring the code.

```
1  while(1){
2    //Load parts of the input data file in a large memory buffer
3    //Break if there is nothing left to read
4    [[spar::ToStream,spar::Input(chunk,split_data)]] while(1){
5      //Fragment the big memory buffer into coarse–grain chunks
6      //Break the loop if we can't split the chunk anymore
7      [[spar::Stage,spar::Input(chunk),spar::Output(chunk),spar::Replicate(n)]]{
8        //Deduplicate and Compress
9      }
10     [[spar::Stage,spar::Input(chunk)]]{
11       //Write the final result
12     }
13   }
14 }
```

Listing 2 High-level representation of Dedup implementation using SPar annotations.

Another advantage of the alternative version is that we achieved sequential code equivalence [15], which means that SPar and sequential versions produced the exact same results. In this case, the sequential and SPar versions of the program were more effectively compressed files. For instance, for the PARSEC's native input, SPar and sequential versions achieved 664.3 MB output while Pthreads and FastFlow program versions achieved 668.2 MB. This difference is related to the *Fragment Refine* stage due to the granularity change in partitioning the chunks.

The activity graph of the alternative version in Fig. 2 was produced by the code in Listing 2. As can be observed, SPar's annotations compose a three stage pipeline. The first being part of the *Fragment* stage, the second formed by a combination of the the *Deduplicate* and *Compress* stages, and the last by the *Reorder* stage. We also added the `Replicate` attribute that is similar to a thread pool in the original Pthread version. From the SPar perspective, it means replicating this annotated code portion as many times as necessary to increase the degree of parallelism. Then, the SPar runtime will send different chunks to each one of the replicas created. In addition, the SPar compiler recognizes the stream items by the declaration of the `Input` and `Output` attributes.

Note that the SPar version preserved the original sequential code version and provided high-level parallelism abstractions. Highlighted benefits include thread pool management targeted by only specifying the `Replicate` attribute, communication between stages by simply indicating input and output dependencies, stream reordering support through a compiler flag, and structured parallel programming with minimal source code modifications.

We must highlight that it is simple to produce different kinds of parallelism exploitation versions once the stream processing region and source code annotated with SPar is identified. Obtaining another possible version from Listing 2 would separate *Deduplicate* and *Compress* into two different stages. The initial step is to add one more `Stage` annotation before line 10 so that these two operations will be performed in different stages. Moreover, as SPar does not support specific stage communications, in this case a control variable is required. It will be updated if the chunk is duplicated in the

Deduplicate stage and sent to the *Compress* stage so that it checks for bypassing when needed. Our experiments did not reveal the need for any performance improvements for this version, and the complexity of the implementation was increased. Hence, we used the original unified stage for our experiments in Sect. 5.

An underlying factor in our parallelism implementation is the full execution of one pipeline each time the program loads parts of the input data file in a large memory buffer (line 2). For PARSEC's native input set this occurs six times. At first glance, this unnecessary overhead might look like a severe performance degradation. Nevertheless, as shown in Sect. 5, good performance could be maintained, primarily because there are few iterations. Moreover, because as previous studies have already pointed out, Dedup's bottleneck is the last stage of writing the results in the output file [4,6, 14]. We measure that bigger input file sizes in the Dedup application may generate overheads. Since programmers may benefit from SPar's simplicity, they will need to move the `ToStream` annotation before the loop (line 1), and perform some code refactoring to deal with jumps, especially known as "go to", which are not suitable for structured parallel programming, as [6] also mentioned. Consequently, programmers may face complexities regarding the original source code. However, the currently best performance-aware version is presented in Listing 2.

4.2 Ferret

Ferret is a PARSEC application intended for a content similarity search in data such as video, audio, and images [3]. This application, originally parallelized with Pthreads, implements the pipeline parallel pattern using six stages. The first and last stages are in charge of input and output. Therefore, they need to execute sequentially. The four middle stages may perform each query in parallel as the activity graph in Fig. 3 (left hand side) demonstrates. The stages are briefly described as follows:

- *Load* Responsible for reading the image that is going to be consumed by the subsequent stages.
- *Segmentation* Divides the image into different objects.
- *Extraction* Computes a 14-dimensional array for each previously detected object. It describes features such as color, shape, and area.
- *Vectorization* Tries to match a set of candidate images from the database of indexed images.
- *Rank* Computes and ranks the images using the EDM (Earth Mover's Distance) metric for each element of the database.
- *Ouput* Writes the results of the previous stage.

The original implementation in the PARSEC benchmark using Pthreads has a dedicated thread pool for each middle stage. The communication uses queues, where the finished stage pushes the result to the next stage. Lock mechanisms are used to avoid race conditions when pushing to the queues. On the right hand side of Fig. 3, we provide an alternative version of this application, which is the coarser-grain computation. The Ferret program structure allows programmers to exploit parallelism in different ways. If the operation sequence is preserved, it is possible to combine the middle

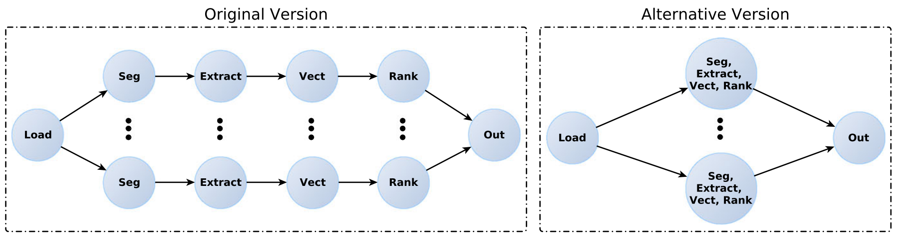


Fig. 3 Activity graph of the Ferret implementation versions

stages. Additionally, the output is not required to be written the same way as the input order.

Before introducing parallelism with SPar in the Ferret source code, we must organize it in a structured way, remove unnecessary jumps, and merge some functions. Our modification does not require the programmer to have previous knowledge or experience with parallel programming, and the final program behavior is the same and unaltered with respect to the original version. We simply make it easier and more structured to introduce parallelism such as the authors of [4,6,14] have done in their codes. Listing 3 represents the Ferret parallelism implementation using the SPar annotations in a high-level codification. This code will behave similarly to the activity graph presented on the left hand side of Fig. 3, which also represents the Pthread version. When using SPar, we only needed to find the stream region in order to annotate it with the `ToStream` attribute and identify the computing stages necessary for putting the `Stage` annotation and its data dependencies through the `Output` and `Input` attributes. Moreover, we added `Replicate` to the middle stage to increase the degree of parallelism.

```
 1 [[spar::ToStream]] while(1){
 2   //Load stage code
 3   [[spar::Stage,spar::Input(image),spar::Output(seg_image),spar::Replicate(n)]]{
 4     //Segmentation stage code
 5   }
 6   [[spar::Stage,spar::Input(seg_image),spar::Output(extract_data),spar::Replicate(n)]]{
 7     //Extract stage code
 8   }
 9   [[spar::Stage,spar::Input(extract_data),spar::Output(vect_data),spar::Replicate(n)]]{
10     //Vectorization stage code
11   }
12   [[spar::Stage,spar::Input(vect_data),spar::Output(rank_data),spar::Replicate(n)]]{
13     //Rank stage code
14   }
15   [[spar::Stage,spar::Input(rank_data)]]{
16     //Output stage code
17   }
18 }
```

Listing 3 High-level representation of Ferret implementation using SPar annotations.

Different pipeline compositions can be created by simply adjusting the existing SPar annotations. For instance, to target the same activity graph of Ferret's alternative version illustrated on the right hand side of Fig. 3, all of the 4 middle stages need to be merged into one large sequential stage. Based on the example in Listing 3, this can be achieved by simply commenting in lines 5, 6, 7, 9, 11 and 12, and adding `rank_data`

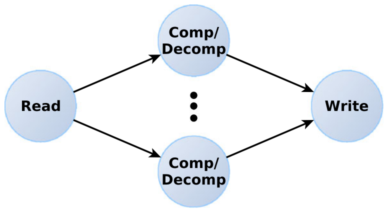


Fig. 4 Activity graph of the Pbzip2 implementation versions

in place of `seg_image` for the `Output()` annotation in line 3. We observed that SPar can easily support different structured parallel code in this application, by simply moving the SPar annotations to the source code. This is an important feature for programmers because they are able to quickly reconfigure pipeline granularity. We identified that other input queries may generate different workloads. Thus, SPar's flexibility along with its simplicity make it an excellent choice for programmers who aim to achieve customized performance-aware solutions in future generation search engines, which also include audio and video similarity searches.

4.3 Bzip2

The parallel version of Bzip2 [20] data compressor is called Pbzip2 [8]. This application is widely used in Linux-based distributions. It can be viewed as two independent structures, one for decompressing and other for compressing files. Figure 4 illustrates a generic activity graph of the original Pthreads implementation, represented as a simple pipeline with three stages inspired by the producer/consumer model. The first stage (*Read*) is responsible for partitioning the input file into independent blocks (the default block size is 900,000 bytes) that are further consumed by the parallel *Compress/Decompress* stage. The final stage (*Write*) writes to the results in the output file.

```
[[spar::ToStream,spar::Input(bytes_left, file_data)]] while(bytes_left>0){
  //Producer stage code
  [[spar::Stage,spar::Input(block_data),spar::Output(compressed_block),spar::Replicate(n)]]{
    //Compress/Decompress stage code
  }
  [[spar::Stage,spar::Input(compressed_block)]]{
    //Writer stage code without stream reordering
  }
}
```

Listing 4 High-level representation of Bzip2 implementation using SPar annotations.

Although this application has two structures, they apply the same parallelism strategy and its output must preserve the input order. In the Pbzip2 implementation,[1] the communication among the stages is performed by using global FIFO queues protected with lock mechanisms. For the middle stage, a thread pool is created so that they consume data blocks to compress or decompress data. The last stage only has to retrieve data blocks from the queue, reorder them, and write the results in the output

[1] http://compression.ca/pbzip2/

file. Pbzip2 implements this control with a reordering algorithm that restructures the data blocks appended to the queue. It uses an auxiliary vector to store the blocks that arrive out of order. The algorithm always checks whether the arriving blocks and those in the vector should be written. Our implementation using SPar for the compression mode was based on the original Pzip2 code. However, it was not possible to express the parallelism directly in Bzip2's sequential decompression code because the *Read* and *Decompress* stages were processed using the same library function. The same problem is reported by Gilchrist [8]. In our case, all of Pthreads' queue management, lock mechanisms, and stream reordering were removed since they are all handled by SPar at runtime and completely abstracted from the application programmer.

Listing 4 depicts the high-level SPar annotation layouts. In general, parallelism annotations in this application were limited to the activity graph in Fig. 4 because the compression/decompression functions are externally implemented. Because this is a legacy code application, we must treat the return operations inside the function that are used for error exception handling. This is because SPar does not allow return routines inside a stream parallelism region (`ToStream`/`Stage`) to prevent the program from crashing during the execution.

In this application, we again observed that SPar guides programmers to structured parallel programming, provides high-level and productive parallelism abstractions, and abstracts all details related to load balancing strategies and synchronization mechanisms such as mutex-based implementations. Consequently, application programmers are able to concentrate on developing smart solutions, because SPar takes care of generating parallel code through the annotations provided in the application code. To measure SPar's feasibility, we performed experiments to evaluate its programmability (Sect. 5.1) and performance (Sect. 5.2), comparing it with state-of-the-art implementations and frameworks.

5 Experiments

Our experiments aimed to assess the programmability and performance of SPar compared to Pthreads, TBB, and FastFlow in the Dedup, Ferret, and Bzip2 applications. We used (a) the Cyclomatic Complexity Number (CCN) [13], used to measure the number of linearly independent paths in a source code, and (b) the Source Lines of Code (SLOC) metrics, to measure the programmability. To evaluate performance, we defined the native input set of PARSEC benchmark for Dedup and Ferret. For Bzip2, we used also Dedup's native input set, which is representative of a real-world workload.

The thread number parameter does not represent the actual number of threads spawned and run in the system. For instance, streaming applications run in a pipeline fashion and each stage may have a thread/replica pool, consequently, this number determines their sizes. This parameter is called the degree of parallelism in the graphs. Moreover, to obtain the execution time metric, we collected the default benchmark time measurements and maintained the original timestamp positions in the source code. We ran each application from 1 up to the total number of cores for the degree of parallelism, repeating the execution of these samples 10 times to get the average execution time. The parallelism degree 0 represents the sequential execution time. The

Table 1 Coding productivity and complexity of the best performance versions

Version	(a) Dedup		(b) Ferret		(c) Bzip2	
	SLOC	CCN	SLOC	CCN	SLOC	CCN
Sequential	454	105	223	30	1278	276
SPar	471	111	274	33	1404	285
FF	974	237	369	37	1607	342
TBB	–	–	376	39	1483	297
Pthreads	1294	288	623	81	1917	366

standard deviation was plotted in the graphs using error bars, which is not visible in most cases because it was negligible. In addition, we implemented Dedup and Ferret as PARSEC plug-ins, allowing programmers to simply use SPar implementation versions through the `parsecmgt` command line tool. Through this feature, we plan to release our codes as an option in the PARSEC benchmark suite along the already supported Pthreads and TBB versions.

The machine was equipped with 24 GB of RAM memory and two Intel(R) Xeon(R) CPU E5-2620 v3 2.40 GHz processors (24 threads with Hyper-Threading). The operating system was an Ubuntu Server 64 bits with the kernel 4.4.0-59-generic. Other software details are: GCC 5.4.0, libraries TBB (4.4 20151115), FastFlow (revision 13), Pbzip2 (1.1.13), and PARSEC benchmarks (3.0). Also, we compiled the programs using the -O3 flag.

5.1 Programmability

Although we implemented several versions, we picked only those with the best performance to compare coding productivity (SLOC) and complexity (CCN). We also highlight that all the evaluated codes for Ferret and Bzip2 employ the same parallelism strategy. Dedup will be discussed later in this section. Table 1 presents the results evaluating and comparing the implemented versions. The first column contains the best versions of each framework considered while the others present the absolute numbers (SLOC and CCN) for each one of the applications. Because the TBB version of Dedup did not work, we will also not consider it in our programmability discussions. Moreover, since the implementation of TBB and FastFlow in the streaming applications has not been discussed previously (Sect. 4), we will describe them briefly here:

- *Dedup-ff* FastFlow provides a template abstraction for pipeline and farm parallel patterns. Although it is possible to implement different skeletons for this application such as described in [6], the best performance version was the `ofarm` in our experiments. For this version, the programmer has to convert the different stages into virtual functions of `ff_node` subclass and set the nodes to different farm roles (emitter, workers and collector), producing an activity graph similar to the alternative version of Fig. 2. The code re-refactoring is simpler than when using Pthreads, since FastFlow's runtime handles stage communication queues, and thread pool management. Another advantage of FastFlow is that the reordering algorithm was removed because it supports the programmer with a farm template that preserves the order of stream items.

- *Ferret-ff* As for Dedup, FastFlow is able to support the implementation of different parallel patterns for Ferret, which is easier than using Pthreads [6]. According to our experiments, the version that achieved the best results was `pipeoffarms`, which produces an activity graph equivalent to Pthreads (illustrated on the left hand side picture in Fig. 3). To parallelize with FastFlow, all stages must be implemented inside a virtual function of `ff_node` subclass. While for each one of the middle stage a farm pattern is instantiated, the first and last stages are sequential. Finally, there is a pipeline function with a series of farm stages instantiated along with the first and last stages, which are sequential code wrappers.
- *Ferret-tbb* TBB also provides a pipeline template in addition to other parallel patterns. Similar to FastFlow, the TBB pipeline construct transforms abstract stages into virtual functions of TBB filter subclasses. Each filter is built with a parameter given by the programmer, which can be `serial_in_order` to maintain the stream order, or `parallel` to extend the degree of parallelism of the stage. The programmer cannot specifically set the number of active threads for each stage, instead, a maximal number of active tokens is given on-the-fly. Then, the TBB's runtime will automatically manage the thread distribution between the available CPUs. The only control the programmer has over the threads is the maximal number of active threads the scheduler can have at the same time. TBB's Ferret implementation [18] generates an equivalent activity graph shown on the left hand side of Fig. 3. The maximal number of tokens on-the-fly is set to 1024, and the task scheduler is set to a maximal number, which is equivalent to the degree of parallelism used in the other Ferret framework versions.
- *Bzip2-ff* The original FastFlow implementation [1] was built based on the Pthread version, with essentially the same stages. The only difference is that they removed the stage communication queues. The activity graph generated is equivalent to the one depicted in Fig. 4. In addition, we tried to implement an ordered farm (`ofarm`) version by removing the original reordering algorithm. However, our experiments showed that this made the performance slightly worse.
- *Bzip2-tbb* Bzip2 TBB was implemented by us based on the Pthread version. Again, to implement parallelism the stages have to be implemented inside a virtual function of a TBB filter subclass. The resulting activity graph is equivalent to the one previously presented in 4. The number of tokens for TBB's task scheduler was configured so that it will always be equal to ten times the total number of the threads' parameter. Another important detail was to set up the first and last filters with `serial_in_order` to guarantee that the output file is fine when the middle filter is `parallel`.

The results for each one of the tested benchmark versions, that are shown in Table 1 were collected from only the source code files that have effectively implemented parallelism, although most of the applications instantiated external libraries. For the sequential version, only the main file was considered. The Dedup application had substantial SLOC and CCN differences, as can be seen in Fig. 5. SPar achieved its exemplary programmability results mainly due to the fact that the sequential code

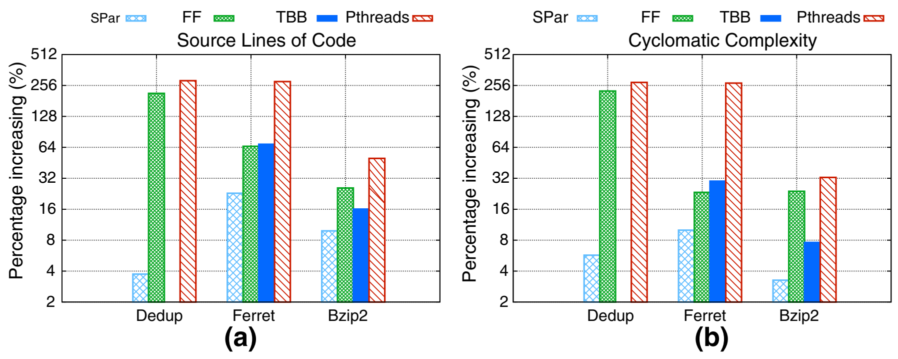


Fig. 5 Coding productivity and complexity increasing w.r.t. sequential version. **a** SLOC. **b** CCN

structure was almost unchanged. In contrast, FastFlow and Pthreads versions implemented a new chunk partitioning stage and the complete implementation of a circular queue structure to hold data during stage communication. Also, Pthreads increased the complexity by adding thread management and stream reordering. In comparision to Pthread, we highlight that the equivalent SPar implementation achieved 543 SLOC and 121 CCN. When compared to the SPar version presented in Table 1, we can observe that the Pthread equivalent implementation has a 15 and 9% higher SLOC and CCN respectively. This is explained by the addition of the extra chunk partitioning stage.

The Ferret source code was not sufficiently well structured to be simply annotated with SPar. It therefore required some code refactoring to express a direct pipeline structure. This same modification was also required by TBB. The unexpected increase in code and complexity in SPar is due to the first stage having to be restructured so that it can be used along with the other stages. Again, SPar still provides a more productive solution, which is proven by the SLOC and CCN metrics. TBB and FastFlow are similar, and present good results when compared to Pthreads because Pthreads requires thread management and queue implementations for stage communications.

In the Bzip2 application, we implemented the parallelism for compress and decompress functions. We can observe in Table 1 that all parallel versions have considerably increased the total amount of code needed. This was previously discussed in 4.3, where another decompression function was needed instead of the original sequential Bzip2 application. Additionally, extra parallelism helper features were implemented in Pbzip2 (the Pthreads version) to simplify the program interactions that were considered in all other versions, too. Once more, SPar achieved the best results with a lower percentage increase for CCN and SLOC with respect to the sequential version. FastFlow had the second worst result because it reused most of the Pthreads structure, which is not productive. On the other hand, TBB achieved better productivity compared to Pthreads and FastFlow.

Finally, we observed that it could be possible to improve the SLOC and CCN numbers in the Bzip2 and Dedup FastFlow implementations. For Bzip2, the default FastFlow library support for stream reordering can be used to completely remove the Pthreads reordering algorithm. In addition to that, in both of these applications, most Pthreads lock mechanisms could be remove. Indeed, TBB and FastFlow also

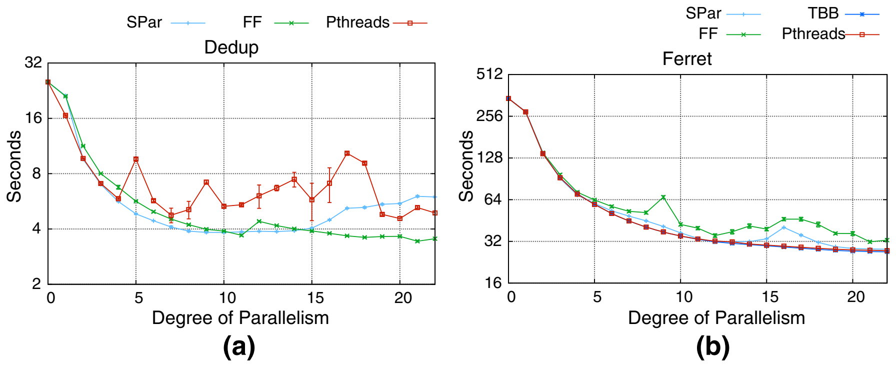


Fig. 6 Dedup and Ferret performance results. **a** Dedup execution time. **b** Ferret execution time

have the support for implementing filters with lambda functions that could improve productivity in Ferret and Bzip2 applications, although the simplicity is questionable because of the tricky syntax and code rewriting required. Even so, SPar would still be more productive, simpler, and less intrusive as we observed for other streaming applications in [9, 11].

5.2 Performance

Figures 6 and 7 present the graphs of the performance results of Dedup, Ferret, and Bzip2. The execution times collected were plotted in the graphs with the respective standard deviations, which were negligible in almost all of the tests, except for the Pthreads version on Dedup application as shown in Fig. 6a. Also, this application is the only one where Pthreads achieved the worst performance. During the experiments, it was observed that maintaining the original activity graph with more stages generates an extra overhead in communication. Therefore, as in FastFlow, for SPar we merged the middle stages into a single stage. We credit the better performance achieved in the lower degree of parallelism (up to 10) with respect to FastFlow due to the removal of the *Fragment Refine* stage. Consequently, FastFlow outperforms SPar with a greater degree of parallelism, which is a results of maintaining the *Fragment Refine* stage.

Before discussing the results of Ferret in Fig. 6b, we must highlight that the alternative version of Ferret presented on the right hand side of Fig. 3 and implemented with SPar achieved 10% worse performance. We did not expected this result, because in theory, reducing the number of stages would subsequently reduce the possibilities of overheads. However, this was not the case in the Ferret application. The results of Fig. 6b show that SPar provides equivalent performance with respect to Pthreads and TBB. As FastFlow's version implemented by the authors of Danelutto et al. [6] had lower performance. We concluded that the SPar compiler is able to generate an optimized FastFlow parallel code with respect to the hand-tuned code.

Because Bzip2 has two different streaming operations, we performed isolated experiments. The graph in Fig. 7a presents the results for the file compression, which is the most costly phase. We can observe that SPar achieved equivalent performance

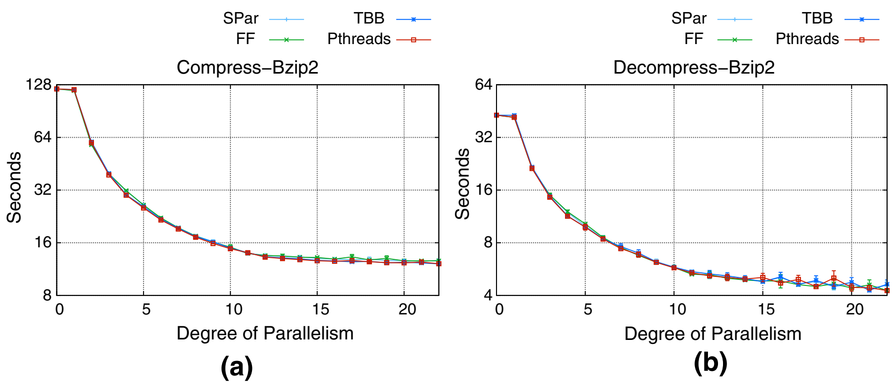


Fig. 7 Bzip2 performance results. **a** Bzip2 compression execution time. **b** Bzip2 decompression execution time

Table 2 The best Speed-ups (S) for the application versions implemented

Version	(a) Dedup		(b) Ferret		(c) Bzip2 (Com.)		(c) Bzip2 (Deco.)	
	Size	S	Size	S	Size	S	Size	S
`SPar`	10	6.59	23	12.24	23	9.80	23	9.76
`SPar_`	10	3.91	20	6.19	23	9.48	23	9.60
`FF`	21	7.35	21	10.84	23	9.74	22	10.04
`TBB`	–	–	24	12.85	22	9.98	23	10.09
`Pthreads`	7	5.30	24	12.62	24	9.91	24	10.31

with respect to the original Pthreads implementation from [8], hand-tuned FastFlow from [1], and our TBB implementation. In Fig. 7b, the results of the file decompression followed a similar trend with respect to the file compression results.

The speed-ups of the applications in our experiments achieved a similar trend compared to the related works running on different multi-core architecture machines [1,4,6,14,17,18]. We summarized the best speed-ups for each one of the applications and evaluated frameworks in Table 2, where for each version we present the size (degree of parallelism in which it achieved the best speed-up), and the speed-up number. We present two different versions of the SPar implementations: `SPar` and `SPar_` (they are in the first and the second row of Table 2). For Ferret, SPar achieved a superior performance when compared to `SPar_` by adding the `spar_ondemand` and `spar_blocking` compilation directives. In addition to the compilation directives, SPar's Bzip2 and Dedup were improved by using a customized reordering algorithm instead of the default FastFlow ordering generated by SPar with the `spar_ordering` flag. The FastFlow implementation also uses an ordering algorithm extracted from the Pthreads implementation in place of its own default ordering. These results further demonstrate SPar's good performance, which is close to the state-of-the-art framework/library implementations. Compared to Pthreads for the best speed-ups, SPar is 10% lower, in the worst cases, regarding the degree of parallelism tested. For TBB implementations, SPar achieved a 1.83% worse speed-up in Bzip2 compression and 4.74% in Ferret. Conclusively, SPar provided only a small performance degradation compared to FastFlow and TBB, in the worst cases it was less than 11%. Therefore, these are very encouraging results for the SPar compiler and its abstraction layer.

6 Conclusion

This paper approached high-level and productive stream parallelism for representative real-world streaming applications (Dedup, Ferret, and Bzip2). We achieved the goal by implementing these applications using the SPar DSL, discussing relevant aspects concerning structured parallel programming, coding productivity, flexibility, and code refactoring. Also, state-of-the-art implementations and frameworks/libraries were selected to perform a fair comparison, and provide reliable results in our experiments. The findings demonstrated that SPar offers great improvements for coding productivity (increasing the SLOC by less than 23% and CCN by less than 10% w.r.t. sequential version in the worst cases) and high-level parallelism abstractions for the streaming application domain without significant performance degradations (less than 10% w.r.t. Pthreads in the worst case). Lastly, the small performance difference between SPar and FastFlow (11% lower than FastFlow in the worst case), highlights the negligible overhead of SPar's code generation and programming interface's abstraction layer.

Currently, we are working on the Dedup application implemented in PARSEC with TBB so that it can be included in our discussions. As future work, we aim to compare this paper's results with other state-of-the-art implementations such as [4] and [14] to assess productivity and performance as have done here in regards to SPar, FastFlow, TBB, and Pthreads implementations. Furthermore, we intend to implement high-level and productive stream parallelism using SPar in other real-world applications.

Acknowledgements Authors thank the partial financial support from the following Brazilian research institutions: CAPES and FAPERGS. Moreover, this work has been partially supported by the EU H2020-ICT-2014-1 Project RePhrase (No. 644235).

References

1. Aldinucci, M., Danelutto, M., Kilpatrick, P., Meneghin, M., Torquati, M.: Accelerating code on multi-cores with FastFlow. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011 Parallel Processing. Lecture Notes in Computer Science, vol. 6853, pp. 170–181. Springer, Berlin (2011)
2. Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: FastFlow: high-level and efficient streaming on multi-core. In: Pllana, S., Xhafa, F. (eds.) Programming Multi-core and Many-core Computing Systems, Parallel and Distributed Computing, vol. 1, p. 14. Wiley, New York (2014)
3. Bienia, C., Kumar, S., Singh, J.P., Li, K.: The PARSEC benchmark suite: characterization and architectural implications. In: 17th International Conference on Parallel Architectures and Compilation Techniques. PACT '08, pp. 72–81. ACM, Toronto, Ontario, Canada (2008)
4. Chasapis, D., Marc, C., Moretó, M., Vidal, R., Ayguadé, E., Labarta, J., Valero, M.: PARSECSs: evaluating the impact of task parallelism in the PARSEC benchmark suite. ACM Trans. Archit. Code Optim. (TACO) **12**(4), 41:1–41:22 (2016)
5. Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press, Cambridge (1989)

6. Danelutto, M., de Matteis, T., de Sensi, D., Mencagli, G., Torquati, M.: P3ARSEC: Towards parallel patterns benchmarking. In: 32nd Annual ACM Symposium on Applied Computing, SAC '17. ACM, Marrakech, Morocco (2017)
7. del Rio Astorga, D., Dolz, M.F., Fernández, J., García, J.D.: A generic parallel pattern interface for stream and data processing. Concurr. Comput. Pract. Exp. **29**, e4175 (2017)
8. Gilchrist, J.: Parallel compression with BZIP2. In: 16th IASTED International Conference on Parallel and Distributed Computing and Systems. PDCS' 04, pp. 559–564. ACTA Press, MIT, Cambridge, USA (2004)
9. Griebler, D.: Domain-Specific Language and Support Tool for High-Level Stream Parallelism. Ph.D. thesis, Faculdade de Informática - PUCRS, Porto Alegre, Brazil (2016)
10. Griebler, D., Danelutto, M., Torquati, M., Fernandes, L.G.: An embedded C++ domain-specific language for stream parallelism. Parallel computing: on the road to exascale. In: Proceedings of the International Conference on Parallel Computing, ParCo'15, pp. 317–326. IOS Press, Edinburgh, Scotland, UK (2015)
11. Griebler, D., Danelutto, M., Torquati, M., Fernandes, L.G.: SPar: a DSL for high-level and productive stream parallelism. Parallel Process. Lett. **27**(01), 20 (2017)
12. ISO/IEC.: Information Technology—Programming Languages—C++. Technical report, International Standard, Geneva, Switzerland (2014). http://www.iso.org/iso/catalogue_detail.htm?csnumber=64029
13. Laird, L.M., Brennan, M.C.: Software Measurement and Estimation: A Practical Approach, 1st edn. Wiley-IEEE Computer Society Pr, Hoboken (2006)
14. Lee, I.T.A., Leiserson, C.E., Schardl, T.B., Zhang, Z., Sukha, J.: On-the-fly pipeline parallelism. ACM Trans. Parallel Comput. **2**(3), 17:1–17:42 (2015)
15. Mattson, T.G., Sanders, B.A., Massingill, B.L.: Patterns for Parallel Programming. Addison-Wesley, Boston (2005)
16. McCool, M., Robison, A.D., Reinders, J.: Structured Parallel Programming: Patterns for Efficient Computation. Morgan Kaufmann, Burlington (2012)
17. Navarro, A., Asenjo, R., Tabik, S., Cascaval, C.: Analytical modeling of pipeline parallelism. In: 18th International Conference on Parallel Architectures and Compilation Techniques. PACT '09, pp. 281–290. IEEE, Washington, DC, USA (2009)
18. Reed, E.C., Chen, N., Johnson, R.E.: Expressing pipeline parallelism using TBB constructs: a case study on what works and what doesn't. In: ACM SIGPLAN Conference on Systems. Programming, Languages and Applications: Software for Humanity (SPLASH), SPLASH '11 Workshops, pp. 133–138. ACM, Portland, Oregon, USA (2011)
19. Reinders, J.: Intel Threading Building Blocks. O'Reilly, Newton (2007)
20. Seward, J.: A Program and Library for Data Compression. http://www.bzip.org/1.0.5/bzip2-manual-1.0.5.html (2017)
21. Soulé, R., Gordon, M.I., Amarasinghe, S., Grimm, R., Hirzel, M.: Dynamic expressivity with static optimization for streaming languages. In: 7th ACM International Conference on Distributed Event-based Systems. DEBS '13, pp. 159–170. ACM, Arlington, Texas, USA (2013)
22. Sujeeth, A.K., Brown, K.J., Lee, H., Rompf, T., Chafi, H., Odersky, M., Olukotun, K.: Delite: a compiler architecture for performance-oriented embedded domain-specific languages. ACM Trans. Embed. Comput. Syst. (TECS) **13**(4), 25 (2014)
23. Thies, W.: Language and Compiler Support for Stream Programs. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts (2009)
24. Zhang, D., Li, Q.J., Rabbah, R., Amarasinghe, S.: A lightweight streaming layer for multicore execution. SIGARCH Comput. Archit. News **36**(2), 18–27 (2008)