High-Level and Efficient Stream Parallelism on Multi-core Systems with SPar for Data Compression Applications

Dalvan Griebler[1], **Renato B. Hoffmann**[1], **Junior Loff**[1], **Marco Danelutto**[2], **Luiz Gustavo Fernandes**[1]

[1] Faculty of Informatics (FACIN), Pontifical Catholic University of Rio Grande do Sul (PUCRS), GMAP Research Group, Porto Alegre, Brazil.

[2]Department of Computer Science, University of Pisa (UNIPI), Pisa, Italy.

`{dalvan.griebler, renato.hoffmann, junior.loff}@acad.pucrs.br, marcod@di.unipi.it, luiz.fernandes@pucrs.br`

Abstract. *The stream processing domain is present in several real-world applications that are running on multi-core systems. In this paper, we focus on data compression applications that are an important sub-set of this domain. Our main goal is to assess the programmability and efficiency of domain-specific language called SPar. It was specially designed for expressing stream parallelism and it promises higher-level parallelism abstractions without significant performance losses. Therefore, we parallelized Lzip and Bzip2 compressors with SPar and compared with state-of-the-art frameworks. The results revealed that SPar is able to efficiently exploit stream parallelism as well as provide suitable abstractions with less code intrusion and code re-factoring.*

1. Introduction

Over the past decade, vendors realized that increasing clock frequency to gain performance was no longer possible. Companies were then forced to slow the clock frequency and start adding multiple processors to their chips. Since that, software started to rely on parallel programming to increase performance [Sutter 2005]. However, exploiting parallelism in such multi-core architectures is a challenging task that is still too low-level and complex for application programmers. Consequently, parallel programming has been reserved for specialized people. On the other hand, application programmers are more concerned with algorithmic strategies and application constraints instead of supporting multi-core parallelism in their applications [Griebler et al. 2017a].

Data compression applications are one of the most important ways to save storage space. In fact, twenty years ago it was already reported that streaming applications were the most computing intensive applications [Rixner et al. 1998]. They are related to a wide set of application, such as video, big data, deep-learning, and among others. Introduce parallelism in these applications is necessary for obtaining high-performance on current multi-core systems. Therefore, combined with the performance needs and high-level parallelism abstraction requirement, new frameworks were proposed to reduce programming effort. However, it is still a challenge to balance abstraction with performance. Most of the languages are inflexible to model modern stream applications, complex for programmers, which has lead them to deal with low-level hardware optimization, or are inefficient to implement real applications [Benkner et al. 2012, Beard et al. 2015].

The so considered state-of-the-art runtimes and parallel programming interfaces for expressing the stream parallelism are Thread Building Blocks (TBB) [Reinders 2007] and FastFlow (FF) [Aldinucci et al. 2014]. They aim to abstract parallelism through parallel patterns with building blocks. Among a set of patterns provided, they also support the parallelism implementation on streaming applications. In addition, an emergent DSL (Domain-Specific Language) named SPar that promise to provide suitable and higher-level abstractions for expressing stream parallelism [Griebler et al. 2017a]. Our goals are to assess the programmability and performance of SPar on multi-core systems for real-world data compression applications. Therefore, we extended studies of the Bzip2 parallelization that were focused only on productivity analysis of SPar in our previous work [Griebler et al. 2017b] and we parallelized Lzip for this paper with SPar, TBB, and FastFlow. Thus, our main contributions are the following:

- The code parallelization of Lzip with SPar, FastFlow, and TBB.
- A comparative analysis of two important performance metrics (memory usage and completion times) and programmability (qualitative and quantitative) of SPar, TBB, FastFlow and POSIX-Threads for Bzip2 and Lzip.

In this paper, Section 2 describes the related works. In Section 3, we introduce SPar DSL. Section 4 discusses two real-world loss-less data compressors (Lzip and Bzip2), considering programming and specific aspects related to the high-level stream parallelism. Then, we detail the experiments evaluating programmability and performance in Section 5. To finalize, our conclusions will be presented in Section 6.

2. Related Work

As we focus on data compression applications, we will discuss related works whose frameworks/libraries are able to exploit stream parallelism on multi-core systems and C++ programs as well as previous works that parallelized Lzip and Bzip2 compressors. For parallel programming, FastFlow [Aldinucci et al. 2014], RaftLib [Beard et al. 2015], and TBB [Reinders 2007] are the ones. FastFlow is a framework created in 2009 by researchers at the University of Pisa and the University of Turin in Italy. It provides stream parallel abstractions adopting an algorithmic skeleton perspective and its implementation is on top of efficient fine grain lock-free communication mechanisms. We mainly used FastFlow as the target of our SPar parallelism implementation because it provides ready to use parallel patterns with high-level C++ templates for stream parallelism.

Another available tool is TBB (Threading Building Blocks), an Intel C++ library for general purpose parallel programming. It emphasizes scalable and data parallel programming while completely abstracting the concept of threads by providing a concept of task. TBB builds on C++ templates to offer common parallel patterns (map, scan, parallel_for, and among others) implemented on top of a work-stealing scheduler [Reinders 2007]. More recently, RaftLib [Beard et al. 2015] is a C++ library designed to support pipeline and data parallelism together. The idea is 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 performed by using lock-free queues. FastFlow and TBB, RaftLib are lower-level parallelism abstractions to the final application programmer with respect to SPar, but they are considered runtimes for SPar.

To the best of our knowledge, there are no previous works that parallelized Lzip compressor with TBB, FastFlow, or RaftLib. However, the work of [Benkner et al. 2012] implemented a parallel version of Bzip2 with TBB using a pipeline of three stages. They compared performance and lines of code for TBB and the original POSIX-Threads version (Pbzip2). Moreover, Bzip2 was also parallelized in [Aldinucci et al. 2011] with FastFlow framework by using a `Farm` and software accelerator feature. In contrast, we provided a completest analysis and comparison of programmability and performance for Bzip2 and Lzip with SPar, TBB, and FastFlow.

3. SPar

SPar is a C++ embedded DSL designed to provide higher-level parallelism abstractions for streaming applications without significant performance losses [Griebler et al. 2017a, Griebler 2016]. SPar uses the standard C++ attribute mechanism [ISO/IEC 2014] for implementing its annotation-based language so that coding productivity is improved. To express stream parallelism with SPar, it offers five different attributes. These attributes are used as parameters of annotations that describe key features of a streaming application such as stages, data consumption and degree of parallelism. The SPar compiler will recognize a SPar annotation when at least the first attribute of a double brackets annotation is specified (`[[id-attr, aux-attr, ...]]`). This first attribute must be an identifier (ID) attribute, where a list of auxiliary (AUX) attributes may be specified if necessary. The `ToStream` and `Stage` annotations are ID while `Input`, `Output` and `Replicate` are AUX.

The SPar compiler was developed with the CINCLE (A Compiler Infrastructure for New C/C++ Language Extensions) support tools [Griebler 2016]. It generates parallel code with calls to the FastFlow library. SPar uses the Farm and Pipeline interfaces and customizes them for its particular needs. More details regarding the SPar usage may be found in [Griebler et al. 2017a, Griebler et al. 2017b]. In addition, 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 item distribution policy 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 preserve the stream items order. FastFlow provides us a built-in function for this purpose so that 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 it.

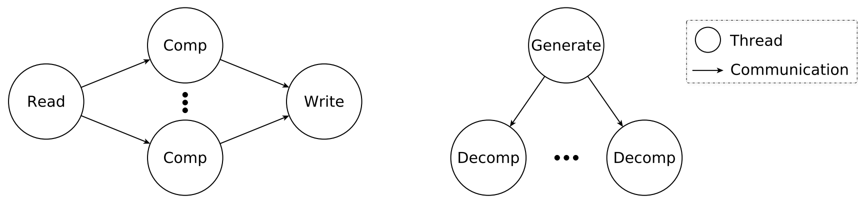
4. Data Compression Applications

The applications used in our studies are Lzip [Diaz 2017] and Bzip2 [Seward 2017]. Although both are compressors, these applications differ from each other on the encoding/decoding algorithm and details about its parallel implementation. We will extend the discussion on both of these compressors in the following sections.

4.1. Lzip

Lzip is a C/C++ lossless data compressor based on the Lempel–Ziv–Markov chain algorithm (LZMA) used by the zip family compressors. There is a standard POSIX-Threads implementation called Plzip that we used to compare to our SPar parallelization concerning efficiency and programmability. The compression mode splits the input file into blocks with a fixed number of bytes. Then, it will apply the LZMA algorithm to encode the blocks, subsequently reassembly them in the resulting compacted file. Similarly, the decompression mode will perform the same tasks, except that instead of encoding, the blocks will be decoded. Plzip adopts a different approach when generating the data-flow for the compression mode. Firstly, Lzip outputs a compressed file of a single large block. As multiple smaller blocks are required for parallel processing, Plzip must adapt the compression so that it operates in multiple blocks instead of one. This entails on 0.4% to 2% larger files and files compressed with Lzip can not benefit from the parallelism of the Plzip decompression.

Figure 1 illustrates the parallel activity graphs of Plzip, which can be viewed as a Pipeline and Farm parallel patterns. Figure 1(a) is arranged in the stream parallelism fashion and the computation is abstracted into three entities represented by `read`, `comp` and `write`. In the decompression mode (Figure 1(b)), the `decomp` entity performs `read`, decoding, and `write` entities. The `generate` entity, only distributes a unique data scope to perform the `decomp` to each thread. The communication between all these entities is performed through queues with the same direction as the arrows are indicating.



(a) Compression. (b) Decompression.

Figure 1. Parallel activity graph of Plzip.

Listing 1 presents the SPar annotations in the sequential version of Lzip compression mode, which will generate the activity graph presented in Figure 1(a). As we can see in line 1, the first annotation uses a `ToStream` attribute to mark the stream region and an `Input` attribute to indicate the data consumed by it. In our example, the `other_data` variable is used as a generic representation of the necessary parameters of the region. In line 4, the second annotation was used to specify the compression computation. In addition to `other_data`, this `Stage` will consume the `data` variable read in the previous `Stage`. At the same time, it will produce the compressed `data` for the subsequent stage (see `Input` and `Output`). None of the blocks have shared variables, therefore, they can be safely processed in parallel (see the use of `Replicate` on line 4). Although it is possible to fragment the compression stage into two other stages, this diminished the performance in out tests. Because most of the processing would be executed by one stage, leaving less work for the other stage. Finally, the last annotation in line 7 will consume the compressed data from the previous stage and write it to the output file. In parallel processing, the order of the queue insertion in the last stage is non-deterministic by default. As we want the output file to be equivalent to the input one, the stream must maintain its original order. In SPar, this is achieved by appending the `-spar_ordered` flag to the compilation command.

Unlike compression, the decompression function of Plzip adopts a data parallelism approach that generates the activity graph presented in Figure 1(b). This is why most of the operations computed in the decompress function involve the `file_index` class. An instantiation of this class will read the input file and split it into several blocks. Knowing the number of available blocks previously, enables a static partitioning of the the blocks to be processed by the parallel threads. Moreover, a finer grain is used to balance the irregular block sizes between threads in the decompression mode. Also, `file_index` contains the offset needed to reorder the blocks in the writing stage. After creating the `file_index` object, the POSIX-Threads implementation will spawn the parallel threads that will decompress and write the data assigned to them.

```
[[spar::ToStream,spar::Input(other_data)]]
while(true) {
 // Reading stage
 [[spar::Stage,spar::Input(data,other_data),
    spar::Output(data),spar::Replicate(
    num_workers)]]{
 // Compression stage
 }
 [[spar::Stage,spar::Input(data)]]{
 // Writing stage
 }
}
```

Listing 1. SPar annotations for the compression mode.

```
1 [[spar::ToStream,spar::Input(num_workers,
     other_data)]]
2 for(int worker_id = 0; worker_id <
     num_workers; ++worker_id){
3   [[spar::Stage,spar::Input(worker_id,
     num_workers,other_data),spar::Replicate(
     num_workers)]]
4   for( long i = worker_id; i < num_blocks; i
       = i+num_workers){
5   //Full Decompression of i indexed block.
6   }
7 }
```

Listing 2. SPar annotations for the decompression mode.

In Listing 2, there is a representation of the SPar annotations for the Lzip decompression mode. Although the decompression mode uses a data parallelism approach instead of stream, it is possible to introduce data parallelism with SPar. The `ToStream` is used to indicate the stream region, which in this case, simply forwards the data to the next stage that is replicated (it has the `Replicate` attribute). Then, the second annotation in line 3 is used to specify the stage that will perform the decompression computation over the assigned data. This stage will consume the `worker_id`, `num_workers` (both used to determine the data computed by the stage) and `other_data`. Following the execution flow, the available input blocks will be iterated in line 5, and each stage will process the i indexed data that is statically assigned to it. This mode does not require `-spar_ordered` because the stream order is obtained with the support of the `file_index` class as previously explained.

4.2. Bzip2

First introduced in 1996, Bzip2 [Seward 2017] is another lossless data compression application. It uses the Burrows–Wheeler transform and Huffman coding algorithms. We adopted the Pbzip2 [Gilchrist 2004] (that is developed with POSIX-Threads) in our studies to compare with our parallel implementations. Both compression and decompression modes in Pbzip2 have the same activity graph as illustrated in Figure 1(a). Therefore, this can be viewed as a pipeline with three stages, where the first stage will split the input file into independent blocks (100,000 - 900,000 bytes) that are forwarded to the parallel (de)compression stage. The remaining stage will be sequentially writing the resulting blocks to the output file.

The original Pbzip2 implementation maintains global queues protected by lock mechanisms to communicate between stages. Initially, a single thread will start to split the input file and fill the first queue with the generated blocks. Meanwhile, the spawned parallel (de)compress threads will query the queue for an available block. Then, the resulting block of the parallel threads will be inserted in the last queue. Meanwhile, the final thread will be constantly checking the queue for the next block to be written in the resulting file. Furthermore, this last thread will reorder the blocks with the help of an auxiliary vector used to store the blocks arriving out of order.

SPar's compression and decompression mode were annotated similarly to Listing 1, which performs like the parallel activity graph in Figure 1(a). Consequently, the SPar's version will have the first stage sequential for splitting the input file into blocks. The second stage was annotated with `Replicate`, performing like a poll of threads that receive from the previous stage the data blocks to apply the (de)compression and deliver the resulting block to the to next and last stage. It will write the (de)compressed blocks to the output file. The order of the stream in SPar is guaranteed by adding the `-spar_ordered` flag in the compilation.

5. Experiments

Our experiments aim to evaluate the programmability and performance of SPar compared to FastFlow, TBB and POSIX-threads in the parallel implementations of Lzip and Bzip2. To measure the programmability we used two different metrics: (i) Cyclomatic Complexity Number (CCN) [Laird and Brennan 2006], which represents the number of linearly independent paths within a source code; and (ii) Source Lines Of Code (SLOC). In our performance evaluations, we used a 704.2 MB ISO file as the workload. The number of workers represented in the graphs does not actually represent the real number of threads spawned by the system. However, it represents the degree of parallelism. For instance, a streaming application developed with a pipeline structure will have a pool of replicated stages. This degree of parallelism plotted in all performance graphs represents this pool, which we also refer to as the number of workers. To obtain the results, we ran each version from 1 up to the max number of threads in the target machine. We executed each degree of parallelism 10 times and obtained the average execution time. Standard deviations were plotted in the graphs by using error-bars.The machine in which the tests were executed was equipped with 24GB of RAM memory and two processors Intel(R) Xeon(R) CPU E5-2620 v3 2.40GHz, with 6 cores each and support to hyper-threading, totalling 24 threads. Its operating system was Ubuntu Server 64 bits with kernel 4.4.0-59-generic. Moreover, we used PBzip2 (1.1.13), Plzip (1.6), GCC 5.4.0 with `-O3` compiler flag, TBB (4.4 20151115), and FastFlow (r13).

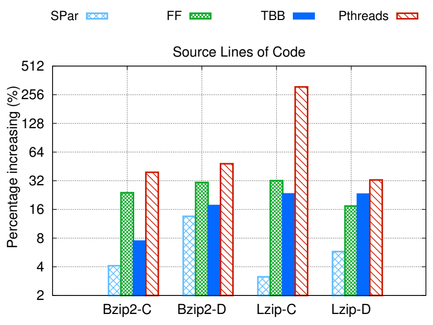
5.1. Programmability

Since we have not presented the FastFlow and TBB versions of Bzip2 and Lzip, we will briefly describe them here:

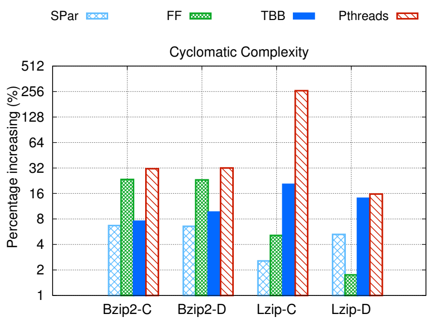
- *Lzip-TBB*: In this implementation, we used pipeline pattern, where the stages of the pipeline are abstracted into virtual functions of a TBB filter subclass. Each one of these filters is constructed with a parameter provided by the programmer. In the compression mode, this parameter was set up with `serial_in_order` in the first and last stages to maintain the original order of the stream. The middle stage was set up with `parallel` to extend the degree of parallelism. Also, we must give a number of maximum tokens that runs on-the-fly through the pipeline. We also tested a number of tokens equal to the degree of parallelism, but we opted to exclude it from our final version since it degrades the performance due to the writing stage bottleneck. The decompression mode implements a pipeline with two stages, where the second stage will perform the computation. The activity graph generated by both versions is the same as the one presented in Figure 1.

- *Lzip-FastFlow*: Although other algorithmic skeletons could be employed, we used FastFlow's *Farm* template. For this template, we must map the stages into FastFlow's *ff_node* sub-classes that represents emitter, workers and collector elements. This generates an activity graph similar to the one presented in Figure 1(a). Although FastFlow abstracts the communication between stages and thread creation, we still had to re-factoring the code and setting up the *Farm* template. One advantage of FastFlow is that it supports an optimized `Farm` template that preserves the order of the stream. Unlike the compression mode, the decompression `Farm` was developed only with the workers node and executed with an offload FastFlow function. The generated activity graph is the same as presented in Figure 1(b).
- *Bzip2-TBB and Bzip2-FastFlow*: Both TBB's and FastFlow's implementation of the compression and decompression modes produce an activity graph such as the one depicted in Figure 1(a). In FastFlow parallelization, we used a *Farm* template while for TBB, we used its *Pipeline* template with a number of tokens equal to ten times the number of workers.

We can see the code intrusion (SLOC) and complexity (CCN) of each version of Lzip and Bzip2 in Figures 2(a) and 2(b), respectively. The y axis is the percentage increase of the metric regarding the sequential code while the x axis represents the application (Bzip2 or Lzip) followed by the suffix C for compression and D for decompression. Note that all POSIX-Threads versions and both the compress and decompress of Bzip2 implemented with FastFlow were already available in [Aldinucci et al. 2011].



(a) Source Lines of Code.



(b) Cyclomatic Complexity.

Figure 2. SLOC and CCN metrics increase with respect to the sequential version.

When collecting the programmability metrics, we only considered the files that effectively implement parallelism. Figure 2 shows that Bzip2 decompression mode has considerably increased the amount of source code needed for all versions. We credit this to the addition of a more complex decompression function used for the parallel versions. This change allows Bzip2's decompression mode to be parallelized since it separates the reading and decompression stage, which were executed by a single external library called in the original sequential code. The compression mode however, could be developed over the original sequential compression function, therefore, the SLOC and CCN increase is less significant. Still observing Bzip2 in Figure 2, we notice that POSIX-Threads has the highest programmability metrics when compared to the sequential code. That is because POSIX-Threads has to manually handle stream reordering, communication protocols, and threads creation. In Bzip2, FastFlow had the second worst result because it reused most of the POSIX-Threads structures, which could be further abstracted by FastFlow.

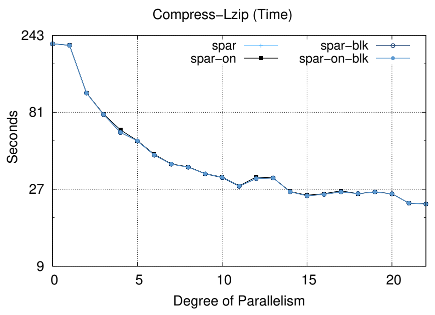
We can observe a similar configuration of SLOC and CCN increase in Figures 2(a) and 2(b) in Lzip parallel versions. Despite FastFlow presenting a higher amount of code needed for decompression compared to SPar, it also presents a lower CCN. That is because FastFlow implements more numerous, smaller, and simpler functions while SPar implements on a single big function, which impacts the CCN negatively. At the same time, FastFlow has a SLOC increase more similar to TBB, since now it takes full advantage of FastFlow's abstractions. One example of this is the abstraction of the producer pipeline stage with a software accelerator that offloads data from the main thread in the decompression function. Also, we highlight that other high-level solutions could be employed by TBB as well (use of `parallal_for` template) for the decompression function, although the need of a different syntax can make this more difficult. Finally, all decompression mode versions achieved a smaller CCN and SLOC increase due to the simpler parallelism strategy employed.

In addition, the fragmentation of the code into `read`, `comp`, and `write` stages is only obtained using parallel programming. This means that the application programmer does not have those details in mind when developing an application. Because of that, when programming FastFlow and TBB over the sequential code, even though low-level details like communication protocols and scheduling options are abstracted, we end up returning to the original POSIX-Threads structure of three separate stages, and need to re-factor the sequential code. SPar on the other hand, does not require the code to be re-shaped/re-written/re-factored in these studied applications, maintaining the sequential code structure.

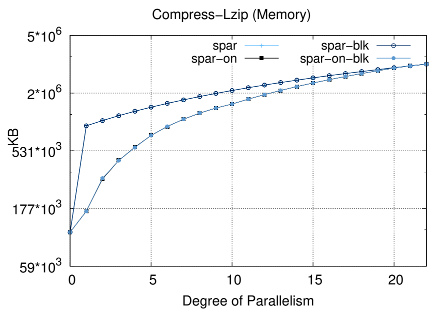
5.2. Performance

We present the performance results in Figures 3 and 4 obtained for the Lzip application and in Figures 5 and 6 for the Bzip2 application. Our parallelizations were compared with the related works for the Lzip and Bzip2 POSIX-Threads version, and Bzip2 with FastFlow. We evaluated execution time, and memory usage for all parallel versions with SPar (`spar`) combining the `spar_ondemand` (`on`) and `spar_blocking` (`blk`) compiler flags. Also, we plotted the POSIX-Threads (`Plzip` and `PBzip2`), FastFlow (`ff`), and TBB (`tbb`) versions. As we can observe, the standard deviation was negligible in almost all the cases because it is not visible through error-bars. In Figure 3, we present the results obtained for all versions of SPar with the possible combinations of the optimization flags. We highlight that these optimizations modify memory consumption while having almost no impact in the total execution time of the Lzip application.

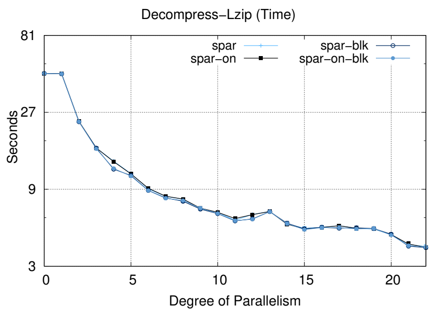
In Figures 3(b) and 3(d), however, the memory usage variation is caused by the `spar_ondemand` flag. Here, an on-demand scheduling is generated by setting the stages queue size to one, meaning that the workload will be distributed dynamically. Each thread will only receive a new item in its queue once it has already removed the previous one. This way, we reduce the concurrent number of total active items, and the memory demand. Figure 4 depicts the comparison of the best SPar version with respect to the best FastFlow, TBB, and POSIX-Threads implementations. We observe that all versions presented a similar completion time. Concerning memory usage in Figure 4(b), the FastFlow compression mode presented a slightly higher demand up to the fifteenth number of workers. That is because FastFlow does not use the on-demand scheduling, which means that more concurrent items will exist in the queues. This could be improved by enabling the on-demand scheduling within code. The memory usage of the decompression mode presented in Figure 4(d) shows that all versions achieved almost identical results, which is



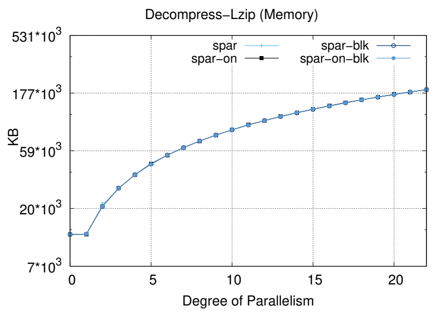
(a) Completion time.



(b) Memory usage.



(c) Completion time.



(d) Memory usage.

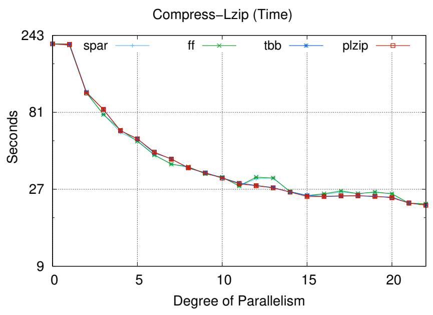
Figure 3. Lzip performance with SPar options.

expected since they all use the same class to manage file splitting and block distribution.

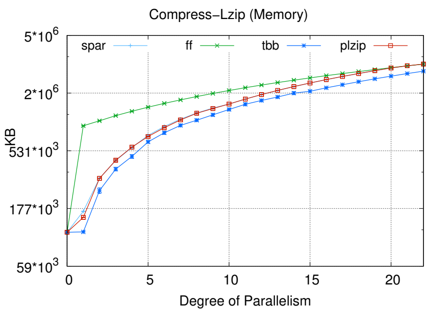
Regarding the Bzip2 application, we can visualize the results of the SPar versions in Figure 5. The completion time achieved a similar result between all SPar versions. However, for the memory consumption we observed similar behavior in Lzip with the SPar versions not using the `spar_ondemand` (on). The same justification of Lzip applies here, the versions with *on* generate less concurrent items in the queues. Even so, in Bzip2 the memory consumption is constant and the disparity between the *on* and non *on* versions is higher. That is because Bzip2 creates larger blocks than Lzip and the whole input file will be in the queues whereas on the *on* versions only part of it will be in the communication queues.

We can compare Bzip2 results in Figure 6. All parallel versions presented very similar performance between them. Our experiments also revealed a pattern of behavior between Lzip and Bzip2 relative to the performance for the parallel versions. In the end, SPar generated efficient FastFlow parallel code, sometimes slightly better than the hand tuned versions such as for the memory usage metric.

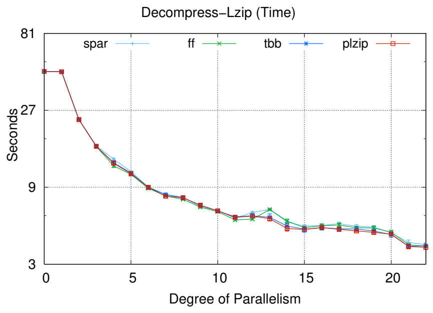
Table 1 presents the best speed-ups (***S***) and its respective degree of parallelism need to achieve (Size) for the parallel implementations of Lzip and Bzip2. As expected, SPar presented a slightly lower performance (6% in the worst case) compared to the original Pthreads implementation. Though SPar generates FastFlow code, there small differences between these two versions for both applications. That is because the FastFlow and SPar versions are not necessarily the same. FastFlow was hand-coded whereas SPar generates the code automatically. This indicates that the code generation overhead is negligible and SPar is able to efficiently abstract stream parallelism for Lzip and Bzip2.



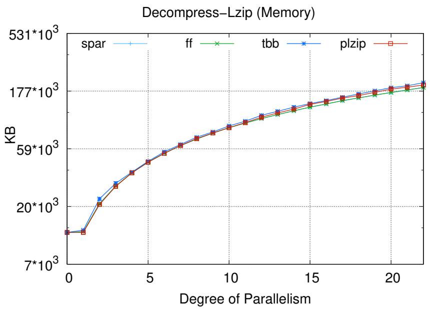
(a) Completion time.



(b) Memory usage.

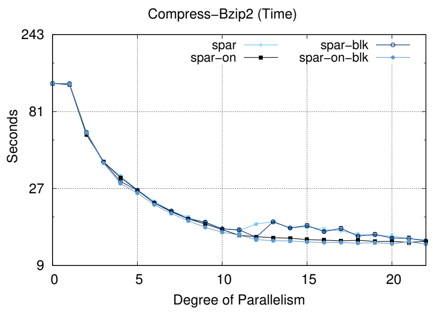


(c) Completion time.

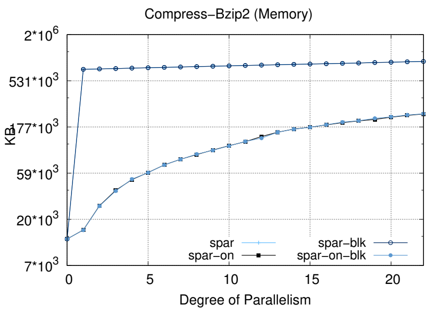


(d) Memory usage.

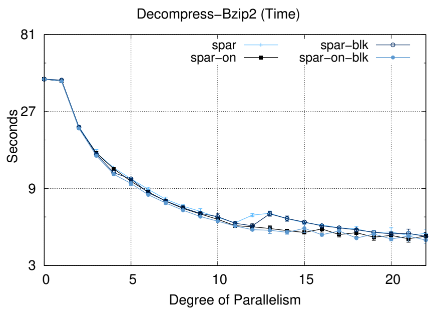
Figure 4. Lzip performance comparison.



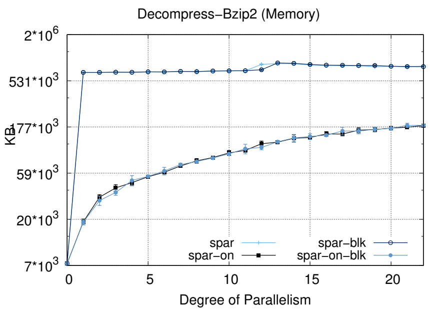
(a) Completion time.



(b) Memory usage.

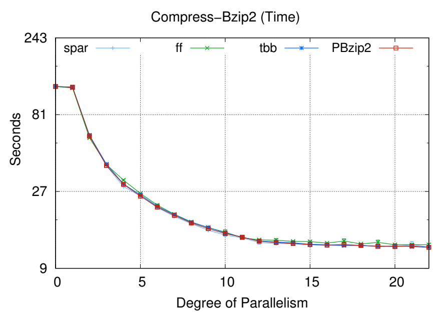


(c) Completion time.

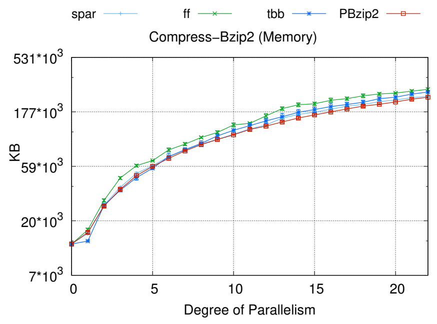


(d) Memory usage.

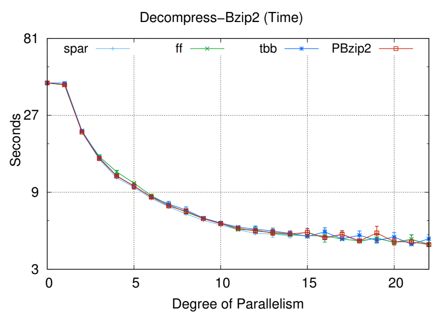
Figure 5. Bzip2 performance with SPar options.



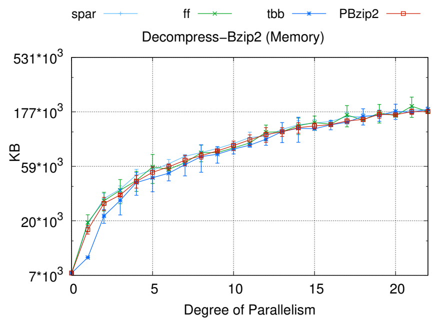
(a) Completion time.



(b) Memory usage.



(c) Completion time.



(d) Memory usage.

Figure 6. Bzip2 performance comparison.

6. Conclusions

In this paper, we assessed the high-level and efficient stream parallelism of SPar for two real-world data compression applications. Our experiments demonstrated these characteristics through performance and programmability analysis of different parallelizations for the Bzip2 and Lzip with SPar compared to POSIX-Threads, FastFlow, and TBB. Therefore, we concluded that SPar is a suitable alternative for expressing parallelism in these applications. It provides good results on completion time, and memory usage. Also, it requires less code intrusion and the qualitative discussion revealed SPar with a simpler syntax. In the future, we plan to implement parallelism with SPar in other real-world applications, including other fields such as deep-learning, network monitoring, stream processing in the fog computing, and network package inspection.

Acknowledgements

Authors thank the partial financial support from CAPES and FAPERGS Brazilian research institutions as well as FACIN/PUCRS. Also, it was supported by the EU H2020-

Table 1. The best Speed-ups (S) for the parallelized versions.

	(a) Lzip(Com.)		**(b) Lzip(Deco.)**		**(c) Bzip2 (Com.)**		**(c) Bzip2 (Deco.)**	
Version	***Size***	***S***	***Size***	***S***	***Size***	***S***	***Size***	***S***
`SPar`	22	9.85	22	11.94	23	9.80	23	9.76
`FF`	22	9.82	22	12.12	23	9.74	22	10.04
`TBB`	24	9.98	21	12.13	22	9.98	23	10.09
`Pthreads`	22	10.01	24	12.39	24	9.91	24	10.31

ICT-2014-1 project RePhrase (No. 644235).

References

Aldinucci, M., Danelutto, M., Kilpatrick, P., Meneghin, M., and Torquati, M. (2011). Accelerating Code on Multi-cores with Fastflow. In *17th International Europen Conference on Parallel and Distributed Computing*, Euro-Par'11, pages 170–181, Bordeaux, France. Springer.

Aldinucci, M., Danelutto, M., Kilpatrick, P., and Torquati, M. (2014). FastFlow: High-Level and Efficient Streaming on Multi-core. In *Programming Multi-core and Many-core Computing Systems*, PDC, page 14. Wiley.

Beard, J. C., Li, P., and Chamberlain, R. D. (2015). 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. ACM.

Benkner, S., Bajrovic, E., Marth, E., Sandrieser, M., Namyst, R., and Thibault, S. (2012). High-Level Support for Pipeline Parallelism on Many-Core Architectures. In *18th International Europen Conference on Parallel and Ditributed Computing*, Euro-Par '12, pages 614–625, Rhodes Island, Greece. Springer.

Diaz, A. D. (2017). Lzip- LZMA Lossless Data Compressor. http://lzip.nongnu.org/.

Gilchrist, J. (2004). Parallel Compression with BZIP2. In *16th IASTED International Conference on Parallel and Distributed Computing and Systems*, PDCS' 04, pages 559–564, MIT, Cambridge, USA. ACTA Press.

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

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

Griebler, D., Filho, R. B. H., Danelutto, M., and Fernandes, L. G. (2017b). High-Level and Productive Stream Parallelism for Dedup, Ferret, and Bzip2. In *10th International Symposium on High-Level Parallel Programming and Applications*, HLPP'17, pages 238–256, Valladolid, Spain.

ISO/IEC, . (2014). Information Technology - Programming Languages - C++. Technical report, International Standard, Geneva, Switzerland.

Laird, L. M. and Brennan, M. C. (2006). *Software Measurement and Estimation: A Practical Approach*. Wiley.

Reinders, J. (2007). *Intel Threading Building Blocks*. O'Reilly, USA.

Rixner, S., Dally, W. J., Kapasi, U. J., Khailany, B., Lbpez-Lagunas, A., Mattson, P. R., and Owens, J. D. (1998). A Bandwidth-Efficient Architecture for Media Processing. In *31st ACM/IEEE Inter. Symp. on Microarchitecture*, pages 3–13, Dallas, Texas, USA. 31st Annual ACM/IEEE International Symposium on Microarchitecture.

Seward, J. (2017). A Program and Library for Data Compression. http://www.bzip.org/1.0.5/bzip2-manual-1.0.5.html.

Sutter, H. (2005). The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobbs journal*, 30(3):202–210.