Analyzing C++ Stream Parallelism in Shared-Memory when Porting to Flink and Storm

Renato B. Hoffmann, Leonardo G. Faé, Isabel H. Manssour, Dalvan Griebler
School of Technology, Pontifical Catholic University of Rio Grande do Sul (PUCRS)
Email: (renato.hoffmann,leonardo.fae)@edu.pucrs.br, (dalvan.griebler,isabel.manssour)@pucrs.br

***Abstract*—Stream processing plays a crucial role in various information-oriented digital systems. Two popular frameworks for real-time data processing, Flink and Storm, provide solutions for effective parallel stream processing in Java. An option to leverage Java's mature ecosystem for distributed stream processing involves porting legacy C++ applications to Java. However, this raises considerations on the adequacy of the equivalent Java mechanisms and potential degradation in throughput. Therefore, our objective is to evaluate programmability and performance when converting stream processing applications from C++ to Java while also exploring the parallelization capabilities offered by Flink and Storm. Furthermore, we aim to assess the throughput of Flink and Storm on shared-memory manycore machines, a hardware architecture commonly found in cloud environments. To achieve this, we conduct experiments involving four different stream processing applications. We highlight challenges encountered when porting C++ to Java and working with Flink and Storm. Furthermore, we discuss throughput, latency, CPU, and memory usage results.**

I. Introduction

Stream processing is essential for information-driven digital systems in healthcare, transportation, logistics, stock-market, telephony, and many others [1]. It encompasses the gathering, processing, filtering, and analyzing of high-volume, heterogeneous, continuous data streams. With a wide applicable range in many science areas, the stream processing paradigm has the core characteristic of continuously processing data as it becomes available. Therefore, computer scientists must employ efficient processing techniques to meet the computational demands of stream processing applications, often with real-time constraints. To that end, coordinating parallel hardware resources is essential.

Flink [2] and Storm [3] are open-source stream processing frameworks for real-time data processing. They provide low latency, high throughput, fault tolerance, scalability, and parallel programming solutions. Each one presents an API that allows the user to define a topological graph representing the stream processing data flow; the frameworks handle the execution via an independent controller process. Users of Flink and Storm need not handle process management, communication, synchronization, and other complex details. Although they serve the same purpose and share many design goals, Flink and Storm differ in their algorithmic solutions.

The C++ programming language is a prevalent alternative for applications that require efficiency and low-level hardware control. On the other hand, Java is a platform-independent language with extensive support. Java applications can run on any platform that supports a Java Virtual Machine, making it a popular choice for developing enterprise-level readily deployable applications [4], [5]. Therefore, porting legacy C++ applications to Java is an option to leverage Java's mature ecosystem for distributed stream processing domain [6]. However, porting parallel stream processing applications from C++ to Java raises considerations on the adequacy of the equivalent mechanisms and potential degradation in performance [7].

Multiple nodes work together in a cluster to perform computational tasks distributed among parallel machines. Beyond the necessary distributed computing techniques, modern clusters also count with many independent multicore computing nodes. In this context, cloud computing allows users to deploy their applications without maintaining their clusters [8]. When utilizing the services of cloud computing providers, distributed application processes may be allocated, partially or entirely, on a shared-memory manycore machine [9], [8]. Furthermore, vertical shared-memory scaling has benefits over solely using horizontal Cloud scaling [10]. Therefore, examining the performance impacts of Java distributed frameworks in a multicore environment is essential to gain insights into these implications.

Our goals are threefold: assess the programmability and qualitative aspects of porting parallel stream processing applications from C++ to Java, evaluate the performance impact of the ported Java applications, and analyze the performance implications of executing Java stream processing frameworks solely on shared-memory multicore architectures. We also summarize our two main scientific contributions: (1) description of steps, difficulties, and features required to convert C++ applications to Java and parallelize them with Flink and Storm; (2) performance analysis of C++ and Java applications running in shared-memory as well as a detailed explanation of the outcomes.

Section II presents the literature study for the remainder of this document. Then, Section III describes in detail our benchmark implementation. Section IV presents our experiments' details and the obtained results. We discuss programmability aspects in Section V and, finally, Section VI concludes with the final remarks.

979-8-3503-8160-3/23/$31.00 ©2023 IEEE
DOI 10.1109/SBAC-PADW60351.2023.00017

II. Related Work

Sungho Lee [11] proposed a JNI program analysis technique. They employ a C analyzer to extract semantic summaries of C functions and transform them into Java code that is analyzed by another Java analyzer. The Java analyzer constructs call graphs that help identify possible bugs in JNI code. They found similar bugs to this work (as we discuss in Section III), including challenges that our applications do not evidence, such as exception handling. On the other hand, they do not explore parallelism considerations. Afonso et al. [12] proposed a framework for Java, C, and OpenCL integration, emphasizing GPGPU parallelism, which brings different challenges than distributed computing. Their main goal is to make GPU acceleration on Java more approachable by combining their framework with JNI calls to C/C++ OpenCL code. They took the opposite porting direction, focusing on converting Java to C++ instead of C++ to Java as we do; the challenges and shortcomings are different. They circumvent the porting challenges by imposing a streamlined Java framework that limits troublesome conversion issues. Regarding programmability, they only measure source lines of code, whereas we also estimate complexity with Halstead.

Voicu et al. [13] proposed a SPark-based framework for integrating GPGPU and FPGA parallelism using automatically generated JNI code. Their methodology allows seamless integration for the Java developer on the SPark big data distributed processing framework. They use the Spark-JNI SPark plugin to port Java code to C++ JNI code, but the authors must highlight the conversion challenges. The results showed up to 12 times faster than pure Java code. Venugopal et al. [14] developed a distributed stream processing engine in C++. They compare their client-to-client solution with traditional Flink and Spark master-client implementation. Functionally, the main difference is that they do not support complex control mechanisms such as fault tolerance or backpressure mechanisms like Flink and Spark. They also do not support dataflow, opting for an in-place directed acyclic graph execution model. However, they consistently outperform Java solutions. Ultimately, we analyze the opposite way we sacrifice C++ performance for Java's robust streaming systems management structures.

Yang et al. [10] estimate the viability of using existing streaming engines for big data streaming applications like Apache Storm, Apache Flink, and Spark Streaming. Similar to this work, they evaluate Java stream processing solutions in a multicore shared memory environment. Their experiments revealed that the shared-memory version achieved 44 times higher than the maximum Storm throughput, claiming that vertical scaling has benefits over simply using Cloud horizontal scaling. Ekanayake et al. [15] evaluated Spark, Flink, and Java MPI scaling on nodes with manycore machines. They compared Java solutions to C implementations of K-means and Multidimensional Scaling algorithms for experiments. The main factor that helps Java reach C-comparable performance are optimizations to thread models, affinity patterns, and communication mechanisms. Mencagli et al. [16] also compared a C++ solution with Flink and Storm, where they achieved lower latency and higher throughput. Windflow is a multicore shared memory solution in which they come to the same conclusion as we do in explaining the performance pitfalls of the Flink and Storm model. However, their comparison is with entirely different implementations between Java and C++ that do not use the same base code with the JNI as ours.

SPar [17] is a C++ domain-specific language for parallel stream processing. Its main goal is to provide a high-level abstraction layer between the user and the low-level details of parallelism. SPar uses C++'s source code annotations containing five attributes representing common stream processing computational features. Then, SPar's compiler transforms the code annotations into the proper parallel code of a specific parallelism framework or runtime. Therefore, SPar's performance is dependent on the underlying runtime. The source-to-source code transformations take place directly in the C++ abstract syntax tree. Currently, SPar supports transformations for FastFlow [17], [18], TBB (Intel Threading Building Blocks) [19], OpenMP [20], distributed [21], and GPGPU [22]. In this work, we use SPar to compare our Java Flink and Storm versions with a C++ programming solution that supports multiple backends. Since we consider shared-memory multicore architectures, we use SPar versions that generate FastFlow, TBB, and OpenMP code.

III. Applications

This Section describes the steps, difficulties, and features required to convert stream applications from C++ to Java. We also discuss parallelization considerations for Flink and Storm, comparing them to the original C++ strategy. There are 3 applications: *Face Recognizer*, *Bzip2*, and *Imgcmp*. Face Recognizer is a video stream processing application that detects faces and recognizes them based on a set of images. Bzip2 is a compression and decompression application that uses the Burrows-Wheeler Transform algorithm. Imgcmp searches a set of images for similar images.

The first challenge is translating the program's necessary C/C++ `struct` to Java `class` using the JNI framework. This is further aggravated by the fact that performing type-checking in the boundary between the languages is impossible. We summarize some of the main challenges: (1) certain C concepts have no obvious Java equivalent (also noted by [11]) such as circular compositions, `void` pointers, and `union`; (2) converting C `structs` into Java `classes` in a serialization-friendly manner demands converting every value the `struct`'s pointers point to into their Java counterparts; (3) increased contact surface between native C and Java codes incurs in some extra overhead.

The parallel versions' general processing strategy is a pipeline with replicated middle stages. In this case, Reader and Writer stages perform sequential I/O at the beginning and end of the pipeline, respectively. The pipeline continuously reads input data in frames, images or chunks of bytes sends each data item to an independent worker, and then collects

and writes the result back to the disk. *Face Recognizer* and *Bzip2* have one middle parallel stage while *Imgcmp* has three.

Implementing pipeline stream parallelism with Flink and Storm is similar to each other. The parallelism strategy is similar to C++, albeit with extra considerations. First, any non-primitive data sent through the pipeline must be serialized/de-serialized. That is because Flink and Storm communicate with messages between different processes, even when running locally on a single machine. Another difficulty in Flink and Storm is re-ordering the output, which is necessary since the non-deterministic nature of parallelism yields scrambled video frames that lose their coherency. On Flink and Storm, we have to implement a re-ordering algorithm manually [23].

Another challenge of Storm and Flink is that they are frameworks that are not conceived to operate in shared memory environments. Their internal workers communicate through messages. Therefore, if the application relies on a mutable global state, the only way to rewrite it in these frameworks is by recreating and synchronizing the state in each parallel process. From a programming point of view, one must look into the application's internal code, often needing more good, extensive documentation; from an execution time performance point of reference, this is computationally costly.

IV. Experiments

In this Section, we evaluate the execution time performance of the Flink and Storm Java versions compared to the original C++ solutions. First, we present the experimental methodology in Section IV-A, followed by a discussion of the experimental results in Section IV-B.

A. Methodology

We executed the experiments in a multicore server that contains two Intel(R) Xeon(R) Silver 4210 2.20GHz processors, totalling 20 physical cores with 40 threads and 140 GB of RAM. The operating system is Ubuntu 20.04.4 LTS (kernel version 5.4.0-105-generic). We use Java version 11, Storm version 2.2.0, Flink version 1.12.0, and G++ version 9.3.0. Furthermore, we specified the target C++ standard version to be 2014, as that is the most recent C++ version SPar supports, and all C++ code uses the optimization flag `-O3`.

Regarding metrics, our experiments consider Latency in seconds, throughput in items/second or MB/second, average CPU usage in percentage, and total Memory usage in MB. We collect CPU usage by sampling the execution. Memory usage is the consumed RAM sampled every 1 ms, and Throughput is the total volume of data or items processed from start to finish of the application. Our CPU usage sums all logical CPU usage and divides it by the total number of logical CPUs, resulting in the average CPU usage of the entire multicore machine. The CPU usage samples every 1 ms.

We measure latency from end to end by introducing time stamps at the exit and the arrival of every message in the pipeline; then, we measure latency as the difference between the two timestamps. We finally sum up all stages' latency, yielding the end-to-end latency representing the total communication cost in seconds between the nodes in the pipeline. In this document, we shortly refer to end-to-end latency simply as latency. However, it is essential to note that this method does not measure the network's latency exclusively because some tuples may be buffered.

In latency and throughput graphs, the X-axis represents the total number of workers, and the standard deviation appears as error bars. The number of workers does not necessarily correspond to the number of active processes or threads in the system. The number of workers represents how many parallel processes or threads spawn. There are extra threads/processes for Sink and Source operations, and Flink and Storm spawn one or more processes to coordinate computation. Given that w is the number of workers, and n is the number of parallel stages, every application has at least $(n * w) + 2$ processes/threads. Bzip2 and Face Recognition have one parallel stage, while Imgcomp has three parallel stages.

The Face Recognizer experiments use a 15-second MPEG-4 input video with 450 frames, a 640x360 resolution, and 10 150x150 training set images. For Bzip2, we take the Canterbury Corpus lossless compression benchmark input and concatenate it into an 800MB file. The decompression input is a 180 MB `bzip2` file, which is the compression output. The input for *Imgcmp* is 114 J-PEG images (sampled from Ferret's native dataset) with 128x96 resolution totalling 684KB of data.

B. Discussion of Results

This Section discusses each application's throughput, latency, CPU, and memory usage results.

1) Face Recognizer: Figure 1 shows the average latency (Y-axis) of all *Face Recognizer* versions. Except for TBB and Flink, all versions increase latency with parallelism. To explain the latency results of each interface, we must understand some considerations about each interface. TBB's task scheduler only operates when the system can immediately continue processing them. TBB only buffers stream tuples up to a maximum number of user-defined tokens, which is ten times the number of workers; therefore, the latency is lower (in our experiments, close to 0) because fewer items are concurrently buffered. FastFlow performs static data buffering (default is 512), which means multiple data items may be alive inside the stream but not currently under processing. OpenMP also does static buffering in the same manner as FastFlow, but the default size of the internal queue buffering is 100 instead of 512. Therefore, it has lower latency than FastFlow because fewer items are alive in the stream awaiting processing resources.

Flink has very low latency due to its internal backpressure mechanism. Even though Flink buffers data, the backpressure mechanisms do not allow many stream tuples concurrently live and halt the execution of non-bottleneck processes. Therefore, the Source only creates stream items if Flinks understands the system can process more stream tuples. Flink's runtime automatically controls the backpressure mechanism.

Storm's backpressure relies on tokens-in-flight, which is similar to TBB. Since we keep default values, Storm has

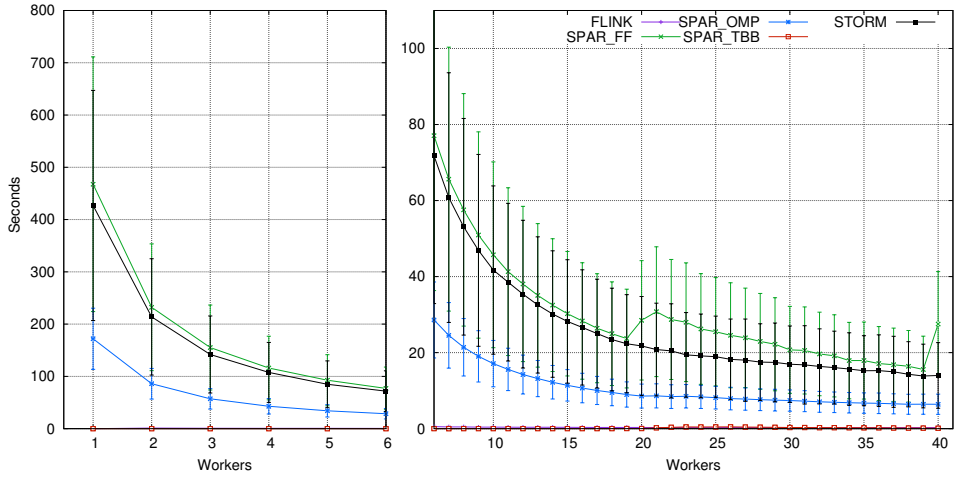


Fig. 1: Face Recognizer latency

default 1000 tokens-in-flight, which is the highest buffering rate of all parallel versions and explains why it has high average latency. Larger buffer sizes entail higher latency standard deviation. Ultimately, buffering or chunking data is detrimental to latency, a vital concern for real-time applications. Improvements include performing backpressure or directly reducing the number of buffered elements. One final consideration regarding latency is that as the number of parallel workers increases, the average latency also tends to reduce. Latency decreases for all versions that keep many items buffered because more parallel workers can process the buffered items faster.

Figure 2 depicts the average *Face Recognizer* throughput. Expectedly, Flink and Storm do not scale as well as the C++ versions in the shared-memory environment. That is because Flink and Storm introduce a significant overhead with their management systems and fault tolerance guarantees that the C++ back-ends do not implement. Another factor detrimental to performance is the Java garbage collector and the cost of communicating between processes instead of threads. These mechanisms could theoretically be disabled to increase performance, but since we consider that they possibly execute on distributed clusters containing manycore, this is likely, not possible. Interestingly, the Java sequential execution yielded 10.57% more throughput than the C++ sequential version. The Java compiler may have some runtime optimization for this specific application that the C++ optimizations could not do.

One extra consideration from Figure 2 is that the FastFlow version has a dip in performance after 19 or 39 workers; the reason is due to a workload imbalance caused by FastFlow's behaviour of pinning threads to cores that end up pinned to virtual cores. Flink does allow setting affinity – equivalent to pinning – but we do not configure it; therefore, it continues scaling on hyper-threading without a performance dip. Storm does not allow pinning since the underlying resource manager controls the resources.

Figure 3 shows the CPU and memory usage with 40 workers. Regarding CPU usage, Flink and Storm have a considerable initialization period, evidenced by the CPU working with less than 20% capacity for 4 seconds in Flink and 8 seconds in Storm. On the other hand, C++ versions achieve close to 100% CPU usage almost immediately after starting execution. CPU usage also reveals that FastFlow's low throughput is due to it operating at 10% of the total CPU's capacity at the end of the stream, which depicts an imbalance in the workload for the reasons mentioned earlier.

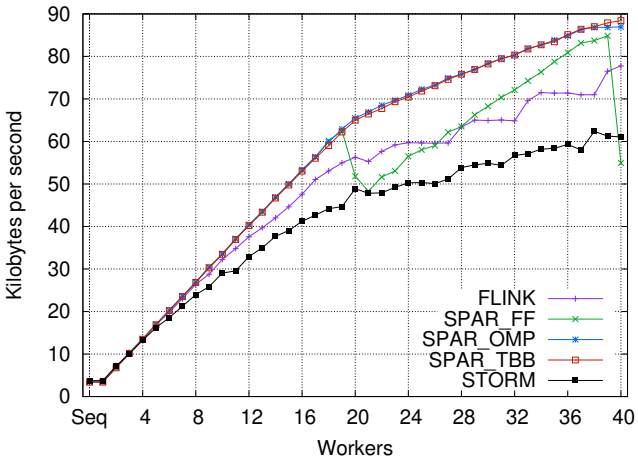
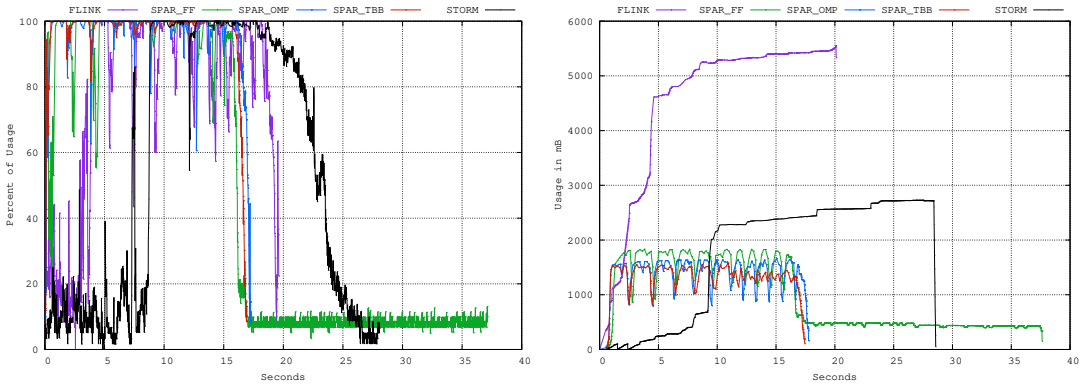


Fig. 2: Face Recognizer average throughput of 10 executions.

About memory usage in Figure 3, all SPar's versions are similar and less than 2000 MB. On the other hand, Flink and Storm demand more memory, while Flink requires double that of Storm. This behaviour remains consistent throughout all applications we examine, so we do not show more memory consumption graphs. Face Recognizer is the most favourable memory consumption application for Java versions compared to C++ versions. In summary, Flink consistently consumes more memory than Storm in our applications, which consume more than C++.



(a) CPU. (b) Memory.

Fig. 3: Face Recognizer CPU and memory usage with 40 workers.

2) Bzip2: We perform *Bzip2*'s experiments for compression and decompression.

Concerning latency in Figure 4, *Bzip2* follows the same overall trend as *Face Recognizer*. The main difference is that `SPAR_FF` delivers latencies much closer to the best versions (Flink and TBB). In this case, the difference is that we statically set all buffer sizes to 1, proving that reducing the communication buffer sizes can improve latency. Every other version remains with the same configurations and has the equivalent results to *Face Recognizer* in latency.

Different from *Face Recognizer*, Figure 5 reveals that Flink and Storm struggle to compare to the C++ frameworks in throughput. In the worst case, the compression throughput is half that of SPar versions and eight times lower for decompression. The throughput difference between `SPar_TBB` and `Flink` is 41.55% in *Face Recognizer* and 47.05% in *Bzip2*'s compression. Furthermore, the scalability is relatively similar between these two applications. Bzip2's decompression scales less than compression because it has less work to perform, and the communication costs start to outweigh the parallelism benefits. Figure 6b shows that the decompression CPU usage is lower than compression.

Beyond what was explained in Face Recognizer, two other reasons contribute to Flink and Storm achieving lower throughput than C++. First, Storm and Flink's versions have a higher communication cost because serialization and process communication is costlier than threads. Second, Storm and Flink use considerably more memory than C++, which increases the communication costs associated with a NUMA architecture and makes Java's garbage collector have to handle larger memory allocation. Another relevant factor is that Bzip2 Java sequential compression throughput is 0.18% superior to C++ sequential, which was mostly better for Java in the Face Recognizer application.

The graphs from Figure 6 showcase Bzip2's resource usage with the maximum number of workers (40). These results confirm that Flink and `SPar_FF`, the two versions with the most unstable throughput measurements, struggle to maintain consistently high CPU usage. We also see that Storm has close to 100% CPU usage for a limited amount of time, which also contributes to explaining its low scalability with a high number of processors.

3) Imgcmp: The latency graph in Figure (7) shows the *Imgcmp* latency. We highlight less latency scaling as Workers increase and more standard deviation or erratic behaviour in latency across all versions. This behaviour is because the *Imgcmp* pipeline depth is higher than Bzip2, so there is more communication. While Bzip2 or Face Recognizer have two producer-consumer communication stages, *Imgcmp* has four. Furthermore, the only versions of *Imgcmp* that keep scaling are the C++ versions. That is because Flink and Storm do not increase throughput as the number of workers increases, as will be discussed later. Flink is the only version that tends to increase latency as the number of Workers increases. In conjunction with the reasons mentioned for the other applications, as the total number of Workers increases, the control and communication get more costly as more parallel agents are involved.

Regarding throughput in Figure 8, the C++ versions continue scaling as the number of Workers grows, but the Java versions do not. The C++ sequential execution yielded 18.35% more throughput than the Java sequential version. Therefore, scalability inefficiency is entirely associated with the Flink and Storm parallelism runtimes. *Imgcmp* is an application that has double the communication channels. Further, *Imgcmp* has more complex serialization than all the other applications (see Section III). Another factor is that *Imgcmp* behaviour severely oversubscribes the system when there are many workers. At 13 workers, the application already oversubscribes the system's logical cores. Strictly at about 13 Workers is when Flink and Storm scalability halts. Flink and Storm operate with processes instead of threads like C++. From the Operational System perspective, switching the context between threads is cheaper than processes, so the over-subscription of cores has benefits in C++ threads but not in Java processes.

Figures 9 offer more insight since it shows that Flink and Storm cannot fully utilize the processor's capacity. The graph shows that the Java frameworks struggle to maintain almost

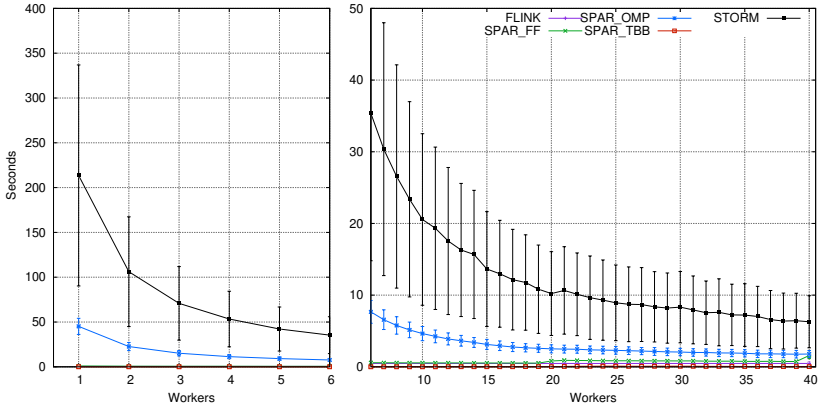


Fig. 4: Bzip2 Compression Latency.

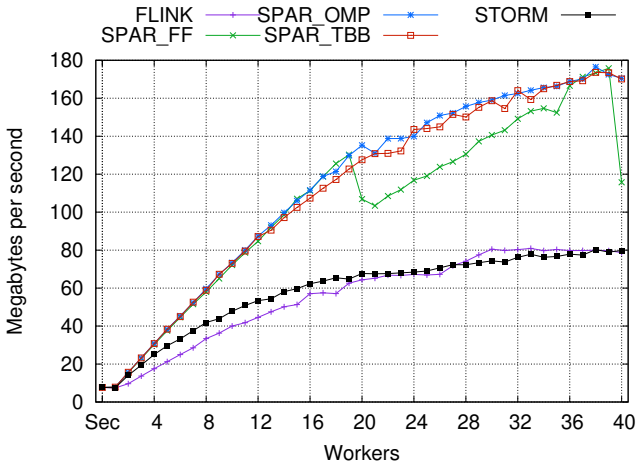


Fig. 5: Bzip2's Compression Throughput.

80% of CPU usage, unlike the SPar versions. Furthermore, it displays an oscillation pattern, whereby Flink and Storm go from around 75% to around 45% CPU usage. This further corroborates that latency (and therefore communication costs) plays a significant role in Flink and Storm's reduced throughput. The frameworks do some processing and then halt during communication. The cost of oversubscribing processes and the higher communication costs, evidenced by higher latency, showcase challenges for Flink and Storm in a shared-memory multicore environment.

V. Programming Productivity Analysis

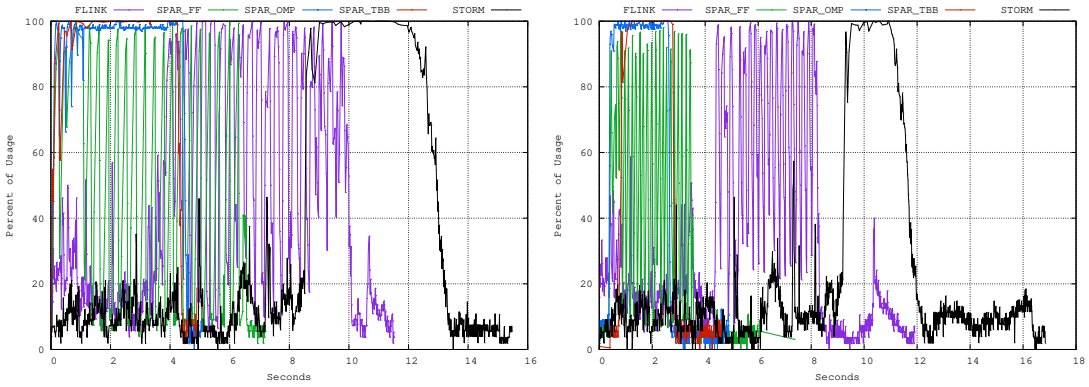
Programming productivity is a crucial factor in comparing different parallel programming interfaces. However, this is a complex task since it involves many personal considerations individual to each programmer. Therefore, one option is to use established code metrics to approximate productivity evaluation [24]. While helpful, code size and complexity evaluations cannot predict the effort required to develop a parallel application. Evaluations that estimate development efforts are more helpful but still have limitations. This work uses SLOC (significant lines of code) and Halstead complexity estimation. SLOC does not count blank or commented lines. Halstead metrics quantify software complexity by measuring program length, vocabulary size, volume, and difficulty in development estimation. For the measurements, we strip the code of any logic irrelevant to the application or parallelism (i.e., we strip latency measurement logic).

Table I shows the difference in SLOC between the parallel and sequential versions of all applications. Overall, a tendency is that Java parallel versions require much more lines of code than C++ parallel versions using SPar. This difference is due to all the extra code that Flink's or Storm's framework requires. Flink and Storm require creating a `class` that implements `interfaces` or inherits from the correct parent for every parallel stage and implementing the required functions. Furthermore, for *Imgcmp* in particular, a whole new utility `class` is required to deal with the serialization of OpenCV's `Mat structs`. Ultimately, the extra code required by the JNI framework explained the significant SLOC differences ranging from 46.88% up to 2470% between Java and C++.

Table I show the programmability metrics. Code tokens represent the sum of elementary units such as operators, operands, keywords, identifiers, literals, and punctuation marks. Halstead estimates development time in hours based on the number of tokens. SPar uses fewer tokens because the other Java versions encourage a style of programming where one declares many classes to do specific things. Fewer tokens reflect in the development time, where the SPar's versions are estimated to take up to 281.25% fewer development hours.

Bzip2 is a different situation, where Storm is estimated to take 5.96% fewer hours to develop than SPar. *Bzip2* is an application committed to portability and backward compatibility; there are many constructs with `#ifdefs` and other conditional compilation features that end up increasing the amount of work necessary to parallelize it, even if most of it is trivial (i.e., changing a header file or a function call). Therefore, for C++ SPar, the results are explained by many preprocessor conditional directives. Java also has the equivalent code, but the verbosity of Bzip2 in C++ gets similar to Java.

Comparing Storm with Flink, Face recognizer Storm takes



(a) Compression (b) Decompression

Fig. 6: Bzip2 CPU usage with 40 workers.

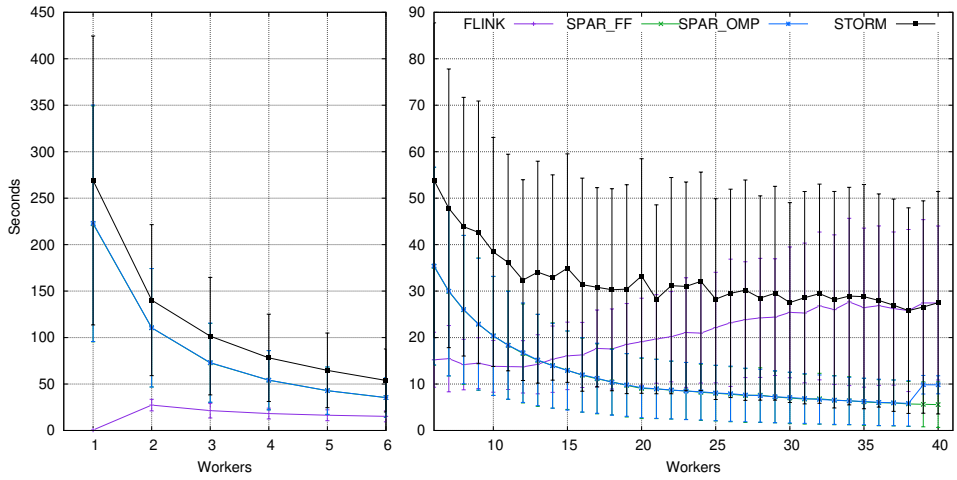


Fig. 7: Imgcmp latency.

TABLE I: Programmability metrics.

	SPar (C++)			Flink (Java)			Storm (Java)		
	SLOC	**Tokens**	**Hours**	**SLOC**	**Tokens**	**Hours**	**SLOC**	**Tokens**	**Hours**
Face Recognizer	10	1226	13	196	3281	45	257	3660	32
Bzip2	96	8879	253	141	10744	360	231	11556	237
Imgcmp	32	1441	16	214	3503	61	196	4406	52

31.12% more SLOC, Bzip2 63.83%, while 8.41% less SLOC on Imgcomp. Overall, Storm requires more verbose code to implement its interface when compared to Flink. However, the Storm interface code is simpler because it has fewer unique operands and operators. The complexity estimation shows that, compared to Flink, Storm takes 28.89% less estimated development hours on Face Recognizer, 34.17% on Bzip2, and 14.75% on Imgcomp.

VI. Conclusion

This paper assessed the challenges and implications of porting stream parallelism applications from C++ to Java. We discussed the qualitative aspects and challenges of porting applications for face recognition, compression, and other image similarity search applications. Furthermore, we explained the parallelization strategy using Flink and Storm. Beyond merely translating the code, Java native interface introduces challenges when handling pointers, dealing with non-existing type checking, and circumventing non-direct C++ and Java equivalents. We also explained that implementing Flink and Storm parallelization is direct but requires serializing data. For the experimental side, we discussed the performance pitfalls of Flink and Storm running in shared-memory multicore machines, which are present in many cluster architectures. We compared Flink and Storm with three C++ parallelism alternatives while explaining latency, throughput, and CPU and memory usage results. We evaluated Halstead software complexity estimation, where C++ versions were estimated to

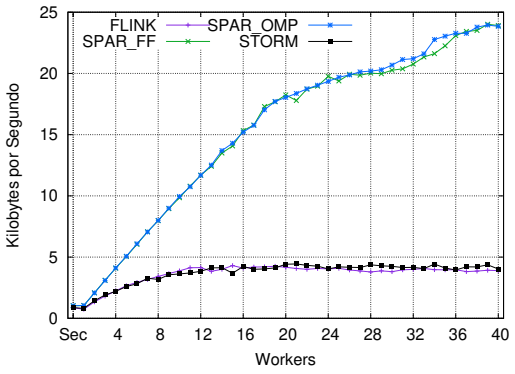


Fig. 8: Imgcmp throughput

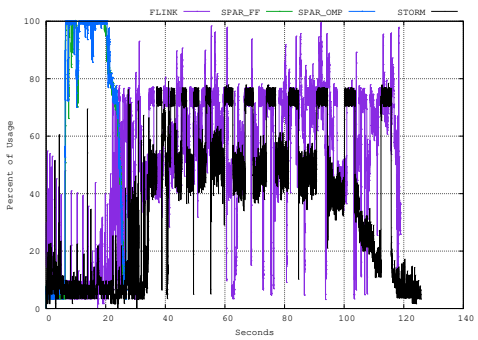


Fig. 9: Imgcmp CPU usage with 40 Workers.

take up to 281% fewer development hours, and Storm takes up to 34% fewer estimated development hours than Flink. We highlight that different applications achieved varied results. The main limitation of our work is that we are limited to the set of characteristics of the four applications we evaluate. Therefore, the main future work involves developing and analyzing applications when porting C++ to Java.

We want to acknowledge the support of LAD-PUCRS, the GMAP research group and PUCRS University. This research is partially funded by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, CNPq Scholarship - Brazil (308456/2020-3), and FAPERGS 10/2020-ARD project SPar4.0 (Nº 21/2551-0000725-7).

References

[1] H. C. M. Andrade, B. Gedik, and D. S. Turaga, *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*. Cambridge: Cambridge University Press, April 2014.

[2] E. Friedman and K. Tzoumas, *Introduction to Apache Flink: stream processing for real time and beyond*. " O'Reilly Media, Inc.", 2016.

[3] M. Jankowski, P. Pathirana, and S. Allen, *Storm Applied: Strategies for real-time event processing*. Simon and Schuster, 2015.

[4] M. Nowicki, L. Górski, and P. Bala, "Pcj java library as a solution to integrate hpc, big data and artificial intelligence workloads," *Journal of Big Data*, vol. 8, pp. 1–21, 2021.

[5] S. Oaks, *Java Performance: In-Depth Advice for Tuning and Programming Java 8, 11, and Beyond*. O'Reilly Media, Inc., March 2020.

[6] A. J. Maas, H. Nazaré, and B. Liblit, "Array length inference for c library bindings," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 461–471. [Online]. Available: https://doi.org/10.1145/2970276.2970310

[7] A. Cheptsov, "An approach for distributed parallelization of large-scale semantic web reasoners based on mpi," in *Web Information Systems Engineering – WISE 2011 and 2012 Workshops*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 4–12.

[8] D. Comer, *The Cloud Computing Book: The Future of Computing Explained*. Chapman and Hall, July 2021.

[9] G. Farina, G. Gala, M. Cinque, and G. Fohler, "Enabling memory access isolation in real-time cloud systems using intel's detection/regulation capabilities," *Journal of Systems Architecture*, vol. 137, p. 102848, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762123000279

[10] S. Yang, Y. Jeong, C. Hong, H. Jun, and B. Burgstaller, "Scalability and state: A critical assessment of throughput obtainable on big data streaming frameworks for applications with and without state information," in *Euro-Par 2017: Parallel Processing Workshops*. Cham: Springer International Publishing, 2018, pp. 141–152.

[11] S. Lee, "Jni program analysis with automatically extracted c semantic summary," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 448–451. [Online]. Available: https://doi.org/10.1145/3293882.3338990

[12] S. Afonso and F. Almeida, "Fancier: A unified framework for java, c, and opencl integration," *IEEE Access*, vol. 9, pp. 164 570–164 588, 2021.

[13] T. A. Voicu and Z. Al-Ars, "Sparkjni: A toolchain for hardware accelerated big data apache spark," in *2019 IEEE 4th International Conference on Big Data Analytics (ICBDA)*, 2019, pp. 152–157.

[14] V. E. Venugopal and e. a. Theobald, "Air: A light-weight yet high-performance dataflow engine based on asynchronous iterative routing," in *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2020, pp. 51–58.

[15] S. Ekanayake, S. Kamburugamuve, P. Wickramasinghe, and G. C. Fox, "Java thread and process performance for parallel machine learning on multicore hpc clusters," in *2016 IEEE International Conference on Big Data (Big Data)*, 2016, pp. 347–354.

[16] G. Mencagli, M. Torquati, A. Cardaci, A. Fais, L. Rinaldi, and M. Danelutto, "Windflow: High-speed continuous stream processing with parallel building blocks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 11, pp. 2748–2763, 2021.

[17] D. Griebler, M. Danelutto, M. Torquati, and L. G. Fernandes, "SPar: A DSL for High-Level and Productive Stream Parallelism," *Parallel Processing Letters*, vol. 27, no. 01, p. 1740005, March 2017. [Online]. Available: http://dx.doi.org/10.1142/S0129626417400059

[18] J. Löff, R. B. Hoffmann, D. Griebler, and L. G. Fernandes, "High-Level Stream and Data Parallelism in C++ for Multi-Cores," in *XXV Brazilian Symposium on Programming Languages (SBLP)*, ser. SBLP'21. Joinville, Brazil: ACM, October 2021, pp. 41–48. [Online]. Available: https://doi.org/10.1145/3475061.3475078

[19] R. B. Hoffmann, D. Griebler, M. Danelutto, and L. G. Fernandes, "Stream Parallelism Annotations for Multi-Core Frameworks," in *XXIV Brazilian Symposium on Programming Languages (SBLP)*, ser. SBLP'20. Natal, Brazil: ACM, October 2020, pp. 48–55. [Online]. Available: https://doi.org/10.1145/3427081.3427088

[20] R. B. Hoffmann, J. Löff, D. Griebler, and L. G. Fernandes, "Openmp as runtime for providing high-level stream parallelism on multi-cores," *The Journal of Supercomputing*, vol. 1, p. 7655–7676, 2022. [Online]. Available: https://doi.org/10.1007/s11227-021-04182-9

[21] R. L. Pieper, "High-level Programming Abstractions for Distributed Stream Processing," Master's Thesis, School of Technology - PPGCC - PUCRS, Porto Alegre, Brazil, October 2020.

[22] D. A. Rockenbach, "High-Level Programming Abstractions for Stream Parallelism on GPUs," Master's Thesis, School of Technology - PPGCC - PUCRS, Porto Alegre, Brazil, November 2020.

[23] D. Griebler, R. B. Hoffmann, M. Danelutto, and L. G. Fernandes, "Stream Parallelism with Ordered Data Constraints on Multi-Core Systems," *Journal of Supercomputing*, vol. 75, no. 8, pp. 4042–4061, July 2018.

[24] G. Andrade, D. Griebler, R. Santos, C. Kessler, A. Ernstsson, and L. G. Fernandes, "Analyzing Programming Effort Model Accuracy of High-Level Parallel Programs for Stream Processing," in *48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2022)*. Gran Canaria, Spain: e.g., IEEE, September 2022, pp. 1–4.