IRIS AperTO

UNIVERSITÀ DEGLI STUDI DI TORINO

AperTO - Archivio Istituzionale Open Access dell'Università di Torino

**A Latency, Throughput, and Programmability Perspective of GrPPI for Streaming on Multi-cores**

**This is a pre print version of the following article:**

*Original Citation:*

*Availability:*

This version is available http://hdl.handle.net/2318/1948959 since 2024-01-08T14:22:24Z

*Publisher:*

IEEE

*Published version:*

DOI:10.1109/PDP59025.2023.00033

*Terms of use:*

Open Access

Anyone can freely access the full text of works made available as "Open Access". Works made available under a Creative Commons license can be used according to the terms and conditions of said license. Use of all other works requires consent of the right holder (author or publisher) if not exempted from copyright protection by the applicable law.

(Article begins on next page)

25 July 2024

# A Latency, Throughput, and Programmability Perspective of GrPPI for Streaming on Multi-cores

Adriano Marques Garcia*, Dalvan Griebler*, Claudio Schepke†, André Sacilotto Santos*, José Daniel García‡, Javier Fernández Muñoz‡, Luiz G. L. Fernandes*

*School of Technology, Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre, Brazil.

†Laboratory of Advances Studies in Computation (LEA), Federal University of Pampa (UNIPAMPA), Alegrete, Brazil.

‡Department of Computer Science, University Carlos III of Madrid (UC3M), Madrid, Spain.

Email: {adriano.garcia, andre.santos01, dalvan.griebler, luiz.fernandes}@edu.pucrs.br, claudioschepke@unipampa.edu.br, {jdgarcia,jfmunoz}@inf.uc3m.es

***Abstract*—Several solutions aim to simplify the burdening task of parallel programming. The GrPPI library is one of them. It allows users to implement parallel code for multiple backends through a unified, abstract, and generic layer while promising minimal overhead on performance. An outspread evaluation of GrPPI regarding stream parallelism with representative metrics for this domain, such as throughput and latency, was not yet done. In this work, we evaluate GrPPI focused on stream processing. We evaluate performance, memory usage, and programming effort and compare them against handwritten parallel code. For this, we use the benchmarking framework SPBench to build custom GrPPI benchmarks. The basis of the benchmarks is real applications, such as Lane Detection, Bzip2, Face Recognizer, and Ferret. Experiments show that while performance is competitive with handwritten code in some cases, in other cases, the infeasibility of fine-tuning GrPPI is a crucial drawback. Despite this, programmability experiments estimate that GrPPI has the potential to reduce by about three times the development time of parallel applications.**

***Index Terms*—Stream Parallelism, Multi-core, GrPPI, SPBench, OpenMP, ISO C++, Intel TBB, FastFlow**

## I. Introduction

Implementing parallelism for stream processing is not easy. Some strategies can mitigate this difficulty, such as structured parallel patterns [1]. Pipeline and Farm are examples of parallel patterns for stream processing. Some parallel programming interfaces (PPIs), such as FastFlow [2] and Intel® Threading Building Blocks (TBB) [3], natively implement parallel patterns. However, even using these PPIs, implementing stream parallelism while achieving performance improvement is still a task for experts. In this context, GrPPI is a generic and reusable parallel pattern interface for both stream processing and data-intensive C++ applications [4]. It allows users to compile their programs with FastFlow, TBB, OpenMP, ISO C++ threads, and other backends from a single generic implementation. Even though they also implement parallel patterns for data stream processing applications [5], which usually have low-latency requirements, and support for distributed platforms [6], [7], which increases the communication delay between nodes of the pipeline, we found no performance evaluation of GrPPI regarding latency in the literature. Previous work evaluated it regarding execution time or speedup.

In this work, we evaluate the performance of GrPPI's backends for multi-cores in terms of throughput, latency, memory usage, and programmability. We implemented four benchmarks using GrPPI and compared the performance of each backend against the handwritten benchmark using FastFlow and TBB. We use SPBench [8] to create the handwritten and GrPPI benchmarks. It is a framework that simplifies the development and management of custom benchmarks for stream processing. Its main goal is to enable users to evaluate and compare the performance of PPIs, which is the purpose of this paper.

The main contributions of this work can be summarized as follows: (1) An analysis of GrPPI performance from a latency perspective using four real-world stream processing applications; (2) An analysis of the programmability of GrPPI using Halstead's method adapted for parallel applications; (3) An extension of the SPBench benchmarking framework with support for benchmark generation using GrPPI.

## II. Related Work

Our related work is papers that have evaluated the performance of applications implemented with GrPPI. Table I summarizes the related work, and the last row regards this paper. Concerning parallel patterns for stream processing, most works have evaluated pipe-farm compositions. The exception was [5], which evaluated patterns more specific to data stream processing. In our work, we evaluate different compositions of farms with pipelines. Considering the PPIs, only [9] and [10] evaluated all backends currently provided by GrPPI. Most papers also compared GrPPI with handwritten parallel code. [4] also compared it with CUDA and [7] with MPI.

Regarding performance, no work has evaluated the latency with GrPPI. Latency is a more sensitive metric than throughput and requires fine-tuning to keep it low for the different PPIs. Evaluating this metric helps show that GrPPI also allows fine-tuning with a generic interface for different backends. Programmability was also a concern for [5], [6], [10], [11]. These works evaluated it regarding lines of code (LOC) or cyclomatic complexity number (CCN). Several benchmarks were used to evaluate GrPPI in the related work. Half of them are synthetic benchmarks, usually applying combinations of image filters. Some also used specific real-world benchmark

*This is a preprint version of: A. M. Garcia et al., "A Latency, Throughput, and Programmability Perspective of GrPPI for Streaming on Multi-cores," 2023 31st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), Naples, Italy, 2023, pp. 164-168, doi: 10.1109/PDP59025.2023.00033.

TABLE I
RELATED WORK SUMMARY TABLE.

| RW | Parallel patterns | Parallel programming interfaces (PPIs) | Performance metrics | Program. metrics | Benchmark applications |
|---|---|---|---|---|---|
| [4] | Pipeline + Farm, Pipeline + Stencil | GrPPI(TBB, THR, OMP), TBB, CUDA, ISO C++ the., OpenMP | Throughput | – | Gaussian blur + Sobel filter benchmark |
| [5] | Stream-Pool, Window-Farm, Stream-Iterator | GrPPI(TBB, THR, OMP) | Speedup | LOC, CCN | FM-Radio and 3 synthetic bench.: Traveling salesman, "Sensor" and "Image" |
| [6] | Pipeline-Farm | GrPPI(THR, MPI) | Speedup | LOC, CCN | Mandelbrot + Gaussian Blur |
| [7] | Pipeline-farm | GrPPI(THR, MPI), Boost-MPI, Spark | Speedup | – | Gaussian blur + Sobel filter and Mandelbrot + Gaussian Blur |
| [9] | Pipeline-farm | GrPPI(TBB, THR, OMP, FF), OMP | Exec. time, Mem. usage, other hardware metrics | – | pHARDI |
| [10] | Map, reduce, stencil, farm | GrPPI(TBB, THR, OMP, FF), FastFlow | Exec. time | LOC, CNN | Four synthetic bench. with simple math and vector operations |
| [11] | Pipeline-farm, map, reduce | GrPPI(THR, OMP), PThreads, OpenMP | Exec. time | LOC | From PARSEC: Swaptions, Blacksholes, Streamcluster, and Ferret |
| [12] | Pipeline-farm | ISO C++ thr., GrPPI(THR, TBB) | Speedup | – | Mandelbrot, Ant colony optimization, Matrix multi., and Image convolution |
| **This work** | **Farm, Pipe-farm, Farm-pipeline** | **GrPPI(TBB, THR, OMP, FF), FastFlow, TBB** | **Throughput, Latency, Memory usage** | **LOC, CCN, PHalstead** | **Lane Detect., Bzip2, Face Recog., and Ferret (PARSEC)** |

applications, such as pHARDI [10], or well-known benchmark suites, such as PARSEC [11].

The general conclusion of the related work is that GrPPI is able to add generic parallelism abstractions to several PPIs with a minimal performance penalty. However, as shown in Table I, most related works evaluated performance regarding execution time/speedup only. None of them measured latency, an increasingly important metric for real-time processing, which is one of the goals of stream processing. The evaluations considering throughput and memory were also quite limited, not considering different strategies and degrees of parallelism or different stream processing applications. In this work, we evaluate GrPPI considering latency, throughput, and memory usage. Regarding programmability, in addition to lines of code and cyclomatic complexity number, we use Halstead's method for parallel applications (PHalstead) from [13].

## III. GrPPI

Some solutions, such as pattern-based programming models, aim to alleviate the burden of parallel programming. Parallel patterns allow the implementation of robust, readable, and portable parallel code while abstracting away the complexity of concurrency control mechanisms. Intel TBB and FastFlow are two examples of PPIs that support parallel patterns. However, such PPIs do not share the same programming interface and require code rewriting to port a parallel application to other platforms. The GrPPI library [4] was developed to overcome these drawbacks and be a unified, generic abstraction layer between PPIs. It proposes to act as a switch between different parallel programming interfaces. It provides a compact and generic parallel interface that seeks to hide the complexity of concurrency mechanisms. It is also highly modular, allowing easy composition of parallel patterns. Its goal is to make applications independent of the parallel programming framework used underneath, thus providing portable and readable codes [10].

In its latest release, GrPPI allows running applications with four backends: ISO C++ threads, FastFlow, OpenMP, and Intel TBB. However, in stream processing, PPIs often offer specific mechanisms for fine-tuning performance, such as the number of tokens in TBB. Latency, for instance, is a sensitive metric that can be excessively high if the application is not properly tuned. Although GrPPI includes directives for managing many of these mechanisms, their extent and functionality have not yet been further evaluated. We argue that evaluating GrPPI with a latency-oriented perspective can show how much it can express parallelism for different stream processing scenarios while maintaining a simple and generic interface.

## IV. Experimental Methodology

In this section, we discuss the methodology used for the experiments. We evaluate performance regarding throughput, latency, and memory consumption with varying degrees of parallelism. All these metrics were provided by SPBench [8]. For the performance evaluation, we ran each benchmark three times and the standard deviations are in the charts.

### A. Execution Environment

All experiments were performed in a computer that has 144 GB of RAM and two Intel(R) Xeon(R) Silver 4210 CPU @ 2.20GHz processors. We used Ubuntu 20.04.4 LTS x86-64 OS, with Linux kernel 5.4.0-105-generic, and GCC 9.4.0 using `-O3` flag. The GrPPI benchmarks were implemented using GrPPI v0.4.0. Intel TBB 2020 Update 2 was used for `GrPPI-TBB` and handwritten TBB benchmarks. Handwrittten code with FastFlow used version 3, while GrPPI used FastFlow 2.2.0.

### B. SPBench Benchmarks

We created GrPPI benchmarks using the SPBench[1] framework [8] and compared their performance against handwritten

[1]https://github.com/GMAP/SPBench

implementations of FastFlow and Intel TBB. SPBench is a framework that allows easy evaluation of different parallel programming interfaces for stream processing in C++. Initially, the framework already provided benchmarks with Intel TBB and FastFLow. We have extended this framework in this work, and now it also provides GrPPI benchmarks.

Listing 1 shows how a SPBench benchmark with GrPPI looks like. It represents a Bzip2 benchmark implementation using a GrPPI farm with OpenMP backend. The Bzip2 application has three stages: Source, Compress/Decompress, and Sink. Therefore we implemented a farm where the Source operator acts as an Emitter, Compress are the workers (which are replicated), and Sink is the Collector.

```
void grppi_func() {
    grppi::parallel_execution_omp ex;
    grppi::pipeline(ex,
        []() std::mutable -> optional<spb::Item> {
        spb::Item item;
        if(!spb::Source::op(item)) { return {}; }
            else { return item; }},
        grppi::farm(spb::nthreads,
            [](spb::Item item) {
            spb::Compress::op(item);
            return item; }),
        [](spb::Item item){ spb::Sink::op(item); }
    );
}
int main (int argc, char* argv[]){
    spb::init_bench(argc, argv);
    spb::Metrics::init();
    grppi_func();
    spb::Metrics::stop();
    spb::end_bench();
}
```

Listing 1. Example of a Bzip2 benchmark in SPBench using GrPPI with OpenMP backend and a single farm.

Although omitted in this example for space reasons, we enabled item sorting to ensure the correctness of the output file, use queues of size 1 to reduce latency, and enable blocking mode to improve resource utilization. We implemented a version of the benchmarks using a mechanism for dynamically changing the `grppi::parallel_execution` mode. For the pipeline of farms implementation, we added more farm stages to the pipeline. For the farm of pipelines, we created a single farm and then added a pipeline to it.

## V. Experimental Results

This section presents the experimental performance, memory usage, and programmability results.

### A. Performance

We measure latency and throughput by varying the degree of parallelism in each farm stage from 1 to 40, the number of threads in the architecture we use. We enabled blocking mode on the PPIs as this allows more efficient use of resources and can improve performance when using hyperthreading, especially in applications that implement a pipeline farm [8]. Figure 1 presents each application's latency (left) and throughput/items per second (right) results. The minimum parallelism degree that GrPPI-OMP accepts is three since it requires two

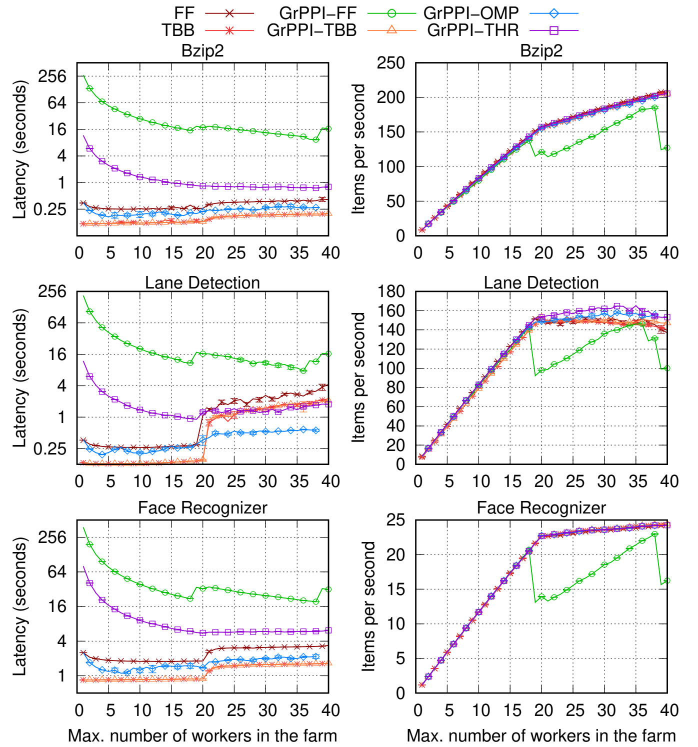


Fig. 1. Latency and throughput of the farm benchmarks with different backends.

dedicated threads for running the source and the sink.So we shifted the results of GrPPI-OMP by -2 in the x-axis.

Figure 1 shows that the throughputs of GrPPI backends (except FF) are equivalent to the handwritten code with TBB and FastFlow (FF). In the Lane Detection application, GrPPI-THR and GrPPI-OMP achieve better performance using hyperthreading. In the case of GrPPI-FF, it achieves throughput comparable to the other PPIs with lower parallelism degrees. Still, the inability to apply blocking mode in GrPPI-FF knocks down performance when using hyperthreading.

Regarding latency, the results of PPIs vary widely. In addition to the inability to enable blocking mode in `GrPPI-FF`, it is also not possible to enable on-demand mode or set the queues to size 1, which would have a similar effect. Therefore, `GrPPI-FF` has an unlimited buffer between stages that stores many items simultaneously. These items wait a long in the buffers/queues until the other stages can process them. It incurs a significant increase in latency. Although `GrPPI-THR` has the best throughput performance, it has the second-worst latency performance overall. However, it manages to have lower latency than the TBB in Lane Detection when using more workers. `GrPPI-TBB` has comparable latency to the handwritten TBB. `GrPPI-OMP` presented a better latency than TBB in Lane Detection over 20 workers.

Figure 2 presents the performance results for the pipeline with multiple farms (PF) and farm with pipelines (FP) compositions. We could not run these versions with the OpenMP backend in GrPPI, so it is not presented. We present these results only for the Ferret application since it originally

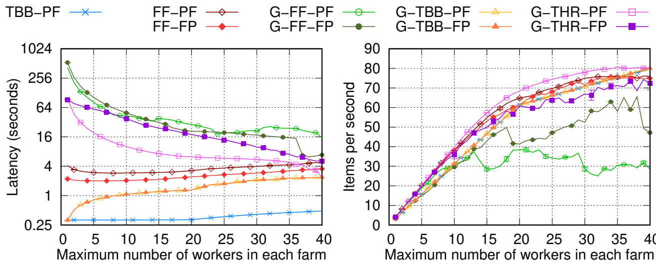


Fig. 2. Ferret with compositions of pipelines and farms.

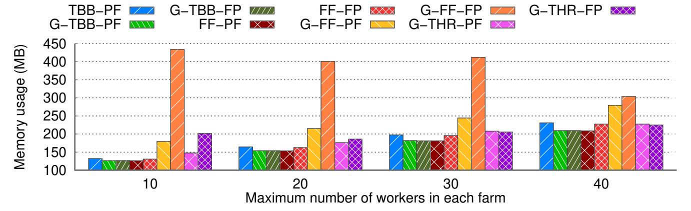


Fig. 3. Total memory consumption of Ferret benchmarks using pipeline-farm (PF) and farm-pipeline (FP) compositions.

implements a pipeline-farm in PARSEC [14]. The highest throughputs were achieved by `GrPPI-THR-PF`, followed closely by the handwritten version of FastFlow. A pipeline of farms in TBB in an application with no ordering requirement (this case) can avoid buffering and process an item from the beginning to the end of the pipeline if resources are available.

In a pipeline of farms, the inability to optimize FastFlow code in GrPPI is a critical factor for both throughput and latency. If on-demand mode is not enabled, this adds multiple unlimited queues/buffers in the pipeline, further increasing latency. Without enabling blocking mode, idle workers are in a busy wait state and do not free up resources. This combination causes a large load unbalance in this application. Since Ferret-PF has a pipeline with four farms and the architecture has 40 threads, it is expected that the pipeline of farms in non-blocking mode will have a significant drop in performance above ten workers per farm. On the other hand, the farm-pipeline pattern avoids the additional collectors/emitters between stages that there would be in a pipeline-farm. So it requires fewer threads and has fewer shared queues, leading to better load balancing and mitigating the performance impact.

Except for the TBB, all the PPIs considerably increased latency with pipe-farm implementations. The difference between TBB and FastFlow in these situations has been extensively discussed in previous work [8], [15]. Despite the difference in throughput between `GrPPI-FF-FP` and `GrPPI-FF-PF`, in terms of latency, the two strategies behaved somewhat similarly, showing the highest latencies. `GrPPI-THR-PF`, however, presented far better results than the FP composition. After all, it achieved the best throughputs and reduced latency to the same level as `GrPPI-TBB` with 40 workers.

### *B. Memory Usage*

We get the memory usage from the SPBench `memory-usage` metric. This metric returns the total memory used by a benchmark during its execution. We ran the benchmarks with a parallelism degree of 10, 20, 30, and 40. The PPI behavior of the benchmarks with a single farm were similar to those from the pipeline-farm compositions. Thus, due to space constraints, we are presenting only the results of the Ferret benchmark in Figure 3.

In most cases, PPIs demanded similar amounts of memory. The apparent exception was `GrPPI-FF-FP`, where its unlimited queues/buffers loaded all data into memory at once. It occurs due to the inability of GrPPI to adjust the size of FastFLow queues and the FastFlow developers' decision not to set a lower default boundary for it. Another contribution to this effect is derived from the strategy of GrPPI, which uses value-oriented bounded queues instead of pointer-oriented ones. This avoids excessive allocation/deallocation at the price of preallocating more memory. However, this high memory usage is present only in the farm-pipe (FP) composition and not so much in the pipe-farm (PF). This is because a lower number of queues is required for a pipeline of farms, where all workers in a farm share a single queue, and consequently, the number of queues is independent of the farm multiplicity.

The memory usage of `GrPPI-FF-PF`, however, is reduced with 40 workers. Probably because resources are tightly contested with blocking mode disabled, and more intensive stages get more processing priority. Thus, such a lack of resources can lead to the inability of the first stage (emitter) to send enough items to fill the queues. This is because the bottleneck produced by the more costly task is reduced, and therefore the bottleneck at the emitter is increased. `GrPPI-THR-FP` also uses more memory with ten workers, but we can see that this is directly linked to the latencies shown in Fig. 2. In general, the PPI that used the least memory in the big picture was OpenMP, followed by the handwritten FastFlow.

### *C. Programmability Evaluation*

[1] says that a PPI should balance three properties: performance, portability, and programmability. However, programmability is not usually addressed in parallel programming and it may be directly linked to the lack of methods and tools that support parallel programming and the difficulty of performing experiments on humans. Thus, most productivity evaluations are tied to code metrics such as lines of code (LOC) or cyclomatic complexity. [13] adapted Halstead's method to address parallel programming and created the PHalstead[2] tool. This method is based on tokens of code (operators and operands) and results in a series of measures, including estimated development time. Figure 4 shows the results of different code metrics, including PHalstead. We measured LOCs and cyclomatic complexity using the Lizard 1.17.10 tool.

The graph in Figure 4 includes the benchmarks implemented with Intel TBB, FastFlow, GrPPI, and sequential applications. With GrPPI, we consider two versions: "GrPPI-static" is a more straightforward implementation that invokes the executor

[2]https://github.com/GMAP/phalstead

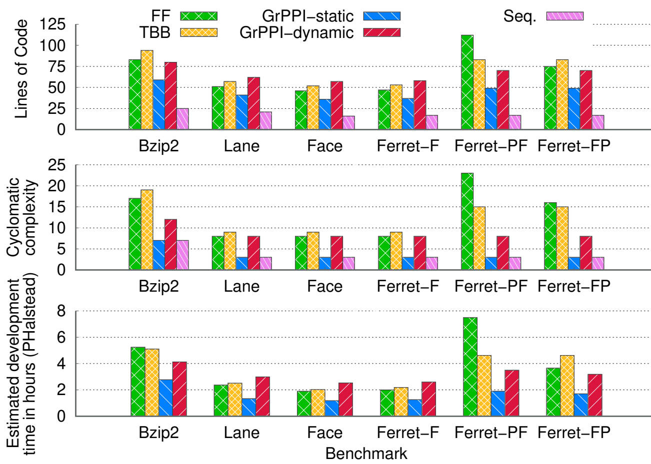


Fig. 4. Number of lines of code, cyclomatic complexity, and estimated development time (PHalstead [13]) of the benchmarks implemented with FastFlow, TBB, GrPPI-static, and GrPPI-dynamic.

of a specific backend statically within the code; "GrPPI-dynamic" is an implementation with a mechanism that allows switching between the four backends dynamically at runtime. We consider these two versions because, although dynamic backend selection is a valuable feature of GrPPI, its use is a user option and not a requirement.

As exposed in Figure 4, GrPPI-static achieved the best results in all cases. In parallel implementations with a single farm, GrPPI-dynamic is similar to TBB and FastFlow. However, we can see that GrPPI shows better results in pipe-farm (PF) and farm-pipe (FP) implementations, where the complexity of programming with FastFlow increases considerably. Concerning cyclomatic complexity, GrPPI-static has equivalent results to the sequential application. On the other hand, PHalstead estimated a longer development time for GrPPI-dynamic with Lane Detection, Face Recognizer, and Ferret-Farm. This extra cost is given by the addition of the backend switching mechanism. In Bzip2, which has two execution modes (compress and decompress), this cost is diluted, and GrPPI-dynamic can maintain a lower development time than TBB and FastFlow. In Ferret PF and FP, this is due to the significant increase in implementation complexity with TBB and FastFlow.

## VI. Conclusion

In this work, we evaluated the GrPPI library targeting stream processing scenarios. Unlike related work, we use more representative metrics to measure stream processing performance, such as latency and throughput. We also evaluate memory usage and programmability/productivity.

As an overall conclusion, we can say that GrPPI does what it promises. It delivers competitive performance while foregoing fine-tuning. However, such fine-tuning can be a crucial requirement for current stream processing applications, which demand real-time processing. Such factors may limit the applicability of GrPPI in more realistic scenarios. As future work, the latency of the GrPPI parallel patterns for data streams could be evaluated using appropriate benchmarks. In addition, a comparison of GrPPI could be made with similar solutions such as SPar [16], which generates code for the same backends as GrPPI but has an annotation-based abstraction approach.

## Acknowledgements

This work was partially supported by the European Union's Horizon 2020 JTI-EuroHPC research and innovation program under grant agreement Nº 956748, project "Adaptive multi-tier intelligent data manager for Exascale" (ADMIRE), by the Spanish Ministry of Science and Innovation, and FAPERGS 10/2020-ARD project SPar4.0 (Nº 21/2551-0000725-7).

## References

[1] M. McCool, J. Reinders, and A. Robison, *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.

[2] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, *Fastflow: High-Level and Efficient Streaming on Multicore*. John Wiley & Sons, Ltd, 2017, ch. 13, pp. 261–280.

[3] C. Pheatt, "Intel® threading building blocks," *Journal of Computing Sciences in Colleges*, vol. 23, no. 4, pp. 298–298, 2008.

[4] D. del Rio Astorga, M. F. Dolz, J. Fernández, and J. D. García, "A generic parallel pattern interface for stream and data processing," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 24, 2017.

[5] D. del Rio Astorga, M. F. Dolz, J. Fernández, and J. D. García, "Paving the way towards high-level parallel pattern interfaces for data stream processing," *Future Gen. Computer Systems*, vol. 87, pp. 228–241, 2018.

[6] J. F. Muñoz, M. F. Dolz, D. del Rio Astorga, J. P. Cepeda, and J. D. García, "Supporting mpi-distributed stream parallel patterns in grppi," in *Proceedings of the 25th European MPI Users' Group Meeting*, ser. EuroMPI'18. New York, NY, USA: ACM, 2018.

[7] J. López-Gómez, J. Fernández Muñoz, D. del Rio Astorga, M. F. Dolz, and J. D. Garcia, "Exploring stream parallel patterns in distributed mpi environments," *Parallel Computing*, vol. 84, pp. 24–36, 2019.

[8] A. M. Garcia, D. Griebler, C. Schepke, and L. G. Fernandes, "SPBench: a framework for creating benchmarks of stream processing applications," *Computing*, vol. In press, no. In press, pp. 1–23, January 2022. [Online]. Available: https://doi.org/10.1007/s00607-021-01025-6

[9] J. Garcia-Blas, D. del Rio Astorga, J. D. García, and J. Carretero, "Exploiting stream parallelism of mri reconstruction using grppi over multiple back-ends," in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2019, pp. 631–637.

[10] J. D. Garcia, D. del Rio, M. Aldinucci, F. Tordini, M. Danelutto, G. Mencagli, and M. Torquati, "Challenging the abstraction penalty in parallel patterns libraries," *The Journal of Supercomputing*, vol. 76, no. 7, pp. 5139–5159, 2020.

[11] C. Vílchez Moya, "Application parallelization and debugging using pattern-based programming," Undergraduate Thesis of Double Degree in Computer Engineering and Mathematics, Faculty of Informatics UCM, Department of Computer Architecture and Automation, Tech. Rep., 2020. [Online]. Available: https://eprints.ucm.es/id/eprint/62014/

[12] C. Brown, V. Janjic, A. D. Barwell, J. D. Garcia, and K. MacKenzie, "Refactoring grppi: generic refactoring for generic parallelism in c++," *Inter. Journal of Parallel Prog.*, vol. 48, no. 4, pp. 603–625, 2020.

[13] G. Andrade, D. Griebler, R. Santos, M. Danelutto, and L. G. Fernandes, "Assessing Coding Metrics for Parallel Programming of Stream Processing Programs on Multi-cores," in *47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, ser. SEAA'21. Pavia, Italy: IEEE, September 2021, pp. 291–295.

[14] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 72–81.

[15] A. M. Garcia, D. Griebler, C. Schepke, and L. G. Fernandes, "Micro-batch and data frequency for stream processing on multi-cores," *The Journal of Supercomputing*, pp. 1–39, 2023. [Online]. Available: https://doi.org/10.1007/s11227-022-05024-y

[16] 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.