PUCRS

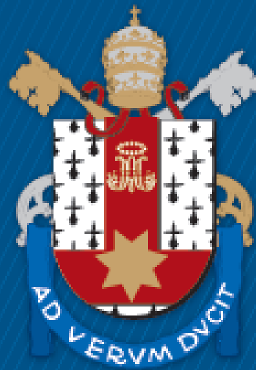
ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
DOUTORADO EM CIÊNCIA DA COMPUTAÇÃO

ADRIANO MARQUES GARCIA

EASING THE BENCHMARKING OF PARALLEL STREAM PROCESSING ON MULTI-CORES

Porto Alegre
2023

PÓS-GRADUAÇÃO - *STRICTO SENSU*



Pontifícia Universidade Católica do Rio Grande do Sul

PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
SCHOOL OF TECHNOLOGY
COMPUTER SCIENCE GRADUATE PROGRAM

EASING THE BENCHMARKING OF PARALLEL STREAM PROCESSING ON MULTI-CORES

ADRIANO MARQUES GARCIA

Doctoral Thesis submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fulfillment of the requirements for the degree of Ph. D. in Computer Science.

Advisor: Prof. Ph.D. Luiz Gustavo Leão Fernandes
Co-Advisor: Prof. Ph.D. Dalvan Griebler
Co-Advisor: Prof. Ph.D. Claudio Schepke

Porto Alegre
2023

Ficha Catalográfica

G216e Garcia, Adriano Marques

Easing the Benchmarking of Parallel Stream Processing on Multi-cores / Adriano Marques Garcia. – 2023.
212 p.
Tese (Doutorado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. Luiz Gustavo Leão Fernandes.
Coorientador: Prof. Dr. Dalvan Griebler.
Coorientador: Prof. Dr. Claudio Schepke.

1. Stream Processing. 2. Framework. 3. Benchmark. 4. Multi-core. I. Fernandes, Luiz Gustavo Leão. II. Griebler, Dalvan. III. Schepke, Claudio. IV. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS
com os dados fornecidos pelo(a) autor(a).
Bibliotecária responsável: Clarissa Jesinska Selbach CRB-10/2051

ADRIANO MARQUES GARCIA

EASING THE BENCHMARKING OF PARALLEL STREAM PROCESSING ON MULTI-CORES

This Doctoral Thesis has been submitted in partial fulfillment of the requirements for the degree of Ph. D. in Computer Science, of the Computer Science Graduate Program, School of Technology of the Pontifical Catholic University of Rio Grande do Sul

Sanctioned on March 31, 2023.

COMMITTEE MEMBERS:

Prof. Ph.D. Marco Aldinucci (University of Turin)

Prof. Ph.D. Claudio Fernando Resin Geyer (UFRGS)

Prof. Ph.D. César Augusto Fonticielha De Rose (PPGCC/PUCRS)

Prof. Ph.D. Dalvan Griebler (PPGCC/PUCRS- Co-Advisor)

Prof. Ph.D. Claudio Schepke (PPGCC/PUCRS- Co-Advisor)

Prof. Ph.D. Luiz Gustavo Leão Fernandes (PPGCC/PUCRS - Advisor)

“We live on an island surrounded by a sea of ignorance. As our island of knowledge grows, so does the shore of our ignorance.”
(John Archibald Wheeler)

ACKNOWLEDGMENTS

My deepest thanks to everyone who helped me in any way during my doctorate. Especially to my family and friends for their help and encouragement and to my advisors for their support and guidance during this project.

I also want to thank my PUCRS/GMAP research group colleagues, who enriched this work through feedback and discussions.

This study was partly financed by the Coordenação de Aperfeiçoamento de Pessoal de Nivel Superior - Brasil (CAPES) - Finance Code 001 and by SAP Labs Latin America.

Acknowledgements to the High-Performance Computing Laboratory of the Pontifical Catholic University of Rio Grande do Sul (LAD-IDEIA/PUCRS, Brazil) for providing computing resources.

FACILITANDO A AVALIAÇÃO DO PROCESSAMENTO PARALELO DE *STREAM* EM ARQUITETURAS MULTI-NÚCLEO

RESUMO

No mundo de hoje impulsionado por dados e crescente expectativa por resultados imediatos, há uma demanda crescente por processamento de dados em tempo real/baixa latência. O processamento de *stream* é uma técnica que processa os dados à medida que eles se tornam disponíveis, permitindo o processamento de dados quase em tempo real. Para lidar com o processamento de grandes volumes de dados, aplicações de processamento de *stream* devem recorrer a técnicas de paralelismo para acelerar o processamento. Embora existam interfaces de programação paralela (IPPs) capazes de adicionar várias camadas de abstração, o paralelismo no processamento de *stream* ainda é uma tarefa difícil e que normalmente exige conhecimento especializado para atingir os níveis de desempenho desejados. Isso gera um grande esforço de pesquisa em direção à aumentar o desempenho do processamento paralelo de *stream* e tornar a programação paralela mais acessível. Tipicamente, *benchmarks* são usados para avaliar as IPPs e novas soluções nesse contexto. No entanto, existem várias limitações nos *benchmarks* existentes, incluindo a falta de *benchmarks* para algumas categorias de aplicações de processamento de *stream*, poucas ou nenhuma opção de parametrização, dificuldade em estender os *benchmarks* para outras IPPs, falta de métricas de desempenho adequadas, falta de preocupação com usabilidade, suporte apenas para linguagens baseadas na Java Virtual Machine (JVM), etc. Este trabalho propõe um *framework* chamado SPBench para criar benchmarks personalizados e avaliar o processamento paralelo de *stream*. Nosso principal objetivo é facilitar o processo de *benchmarking* no processamento de *stream*, incluindo a criação, compilação, execução, ajuste-fino e avaliação dos *benchmarks*. Portanto, esta tese de doutorado fornece as seguintes principais contribuições científicas: (I) Um *framework* que simplifica o *benchmarking* de aplicações de processamento de *stream*, fornecendo uma Application Programming Interface (API) e uma interface de linha de

comando para simplificar, reutilizar código, personalizar, estender e avaliar diferentes aspectos ou propriedades em relação ao processamento paralelo de *stream*. (II) Um conjunto de *benchmarks* paralelos em C++ para processamento de *stream* que inclui aplicações do mundo real e as IPPs mais utilizadas neste contexto. (III) Um estudo comparativo abrangente das IPPs mais populares que usam o paralelismo de *stream* em C++. (IV) Mecanismos para simulação dinâmica de frequência de *stream* de dados em aplicações de processamento de *stream*, com um conjunto de algoritmos para gerar os padrões de frequência de *stream* de dados mais comumente usados na literatura e uma análise do impacto da frequência de dados no desempenho dessas aplicações. (V) Uma análise do impacto do tamanho de *micro-batches* no desempenho de aplicações de processamento de *stream*, incluindo mecanismos para controle dinâmico de *batch* baseado em tamanho específico ou intervalos de tempo. Testamos o *framework* SPBench com cinco aplicações do mundo real de processamento de vídeo/imagem, compressão de dados e detecção de fraudes. Neste trabalho nós tentamos mostrar os benefícios do SPBench usando-o em combinação com IPPs para gerar benchmarks de processamento paralelo de *stream* e realizar diversas análises. No geral, os resultados mostraram que as abstrações de alto nível das IPPs podem causar um grande impacto no desempenho quando abstraem mecanismos de ajuste fino. Nos experimentos de frequência de dados, a IPP FastFlow obteve mais benefícios de cenários de frequência variável do que o TBB em nossos casos de teste. Por fim, os resultados experimentais mostraram que a potencial vantagem de desempenho do uso de *microbatches* em ambientes multi-núcleo tende a aparecer apenas em cenários muito específicos.

Palavras-Chave: Processamento de Fluxo, Framework, Benchmark, Multi-núcleo.

EASING THE BENCHMARKING OF PARALLEL STREAM PROCESSING ON MULTI-CORES

ABSTRACT

In today's fast-changing data-driven world, there is increasing demand for real-time/low-latency data processing. Stream processing is a technique that envisages processing data as it becomes available, enabling near real-time data processing. Stream processing applications must resort to parallelism techniques to speed up processing and to cope with processing large volumes of data. Although there are parallel programming interfaces (PPIs) that add several abstraction layers, parallelism in stream processing is still a difficult task, usually demanding expert knowledge to achieve desired performance levels. This generates a lot of research effort toward boosting parallel stream processing performance and making parallel programming more accessible. Typically, benchmarks are used to evaluate the PPIs and new solutions in this context. However, there are a number of limitations in existing benchmarks, including not addressing some categories of stream processing applications, few or no parameterization options, difficulty extending the benchmarks to other PPIs, lack of appropriate performance metrics, poor usability, only targeting JVM-based languages, and others. This work proposes a framework called SPBench for creating custom benchmarks and evaluating parallel stream processing. Our main goal is to ease the benchmarking process in parallel stream processing, including the creation, building, execution, tuning, and evaluating of the benchmarks. Therefore, this doctoral dissertation provides the following main scientific contributions: (I) A framework that simplifies the benchmarking of stream processing applications, providing an API and a command-line interface to simplify, reuse code, customize, extend, and evaluate different aspects or properties regarding parallel stream processing. (II) A parallel C++ benchmark suite for stream processing that includes real-world applications and the most state-of-the-art Parallel Programming Interfaces (PPIs) in this context. (III) A comprehensive comparative study of the most popular PPIs leveraging C++ stream parallelism. (IV) Mechanisms for

dynamic data stream frequency simulation in stream processing applications with a set of algorithms for generating the literature's most commonly used data stream frequency patterns and an analysis of the data frequency impact on the performance of stream processing applications. (V) An analysis of the performance impact of micro-batch sizing on stream processing applications, including mechanisms for real-time and dynamic batching management, allowing users to adjust batch sizes on the fly either based on specific size targets or time intervals. We test the SPBench framework with five real-world applications of video/image processing, data compression, and fraud detection. We show the benefits of SPBench by using it in combination with PPIs to generate parallel stream processing benchmarks and conduct various analyses. Overall, the results showed that the high-level abstractions of PPIs can cause significant performance penalties when they hide fine-tuning mechanisms. In the data frequency experiments, the FastFlow PPI benefited more from varying frequency scenarios than the TBB in our test cases. Finally, the experimental results showed that the potential performance advantage of using micro-batches on multi-cores tends to show up only in specific scenarios.

Keywords: Stream Processing, Framework, Benchmark, Multi-core.

LIST OF FIGURES

2.1 Overview of parallel patterns. 28
2.2 Stream processing applications. 31
2.3 Stream processing workloads. 32
2.4 Most common terms for stream processing applications and technologies found in 70 surveys published between 2008 and 2020. 33
2.5 A dataflow system representation. 34
2.6 Push and pull data propagation model. 36
2.7 A data stream application to continuously count hashtags. Adapted from [HK19]. 38
2.8 Stream systems representation, from [Gri16a]. 40
2.9 Stateless and stateful stream processing operators. 41
2.10 Pipeline, task, and data parallelism in stream graphs. Adapted from [HSS+14] 42
3.1 Users' perspective when writing stream parallel code: traditional vs. SP-Bench way. 61
3.2 SPBench architecture. 61
3.3 SPBench framework. 62
3.4 SPBench architecture. 63
3.5 Example of a word-counter micro-benchmark generation using NAMB. 78
4.1 Bzip2 flow graph. 86
4.2 Face Recognizer workflow. 86
4.3 Face Recognizer flow graph. 86
4.4 Lane Detection workflow. 87
4.5 Lane Detection flow graph. 87
4.6 Ferret workflow. 88
4.7 Ferret flow graph. 88
4.8 Fraud Detection flow graph. 89
4.9 Farm implementation in FastFlow. 96
4.10 Structure of a Pipeline of Farms implementation with FastFlow in SPBench. 96
4.11 Structure of a farm implementation with Threading Building Blocks (TBB) in SPBench. 98
5.1 Characterization results (specific y scales for each application). 120
5.2 Characterization results (same y scales for all applications). 121
5.3 Example of SPBench latency results from the execution of Ferret (sequential).123

5.4 Throughput results of the TBB, FastFlow, OpenMP, and ISO C++ Threads Farm implementations in different computers. 126
5.5 Latency results of the TBB, FastFlow, OpenMP, and ISO C++ Threads Farm implementations in different computers. 127
5.6 Latency and throughput of the Bzip2 (compress mode) benchmarks with different GrPPI backends and PPIs. 129
5.7 Latency and throughput of the Lane Detection benchmarks with different GrPPI backends and PPIs. 129
5.8 Latency and throughput of the Face Recognizer benchmarks with different GrPPI backends and PPIs. 129
5.9 Latency and throughput of the Ferret (farm) benchmarks with different GrPPI backends and PPIs. 130
5.10 Ferret with compositions of pipelines and farms. 131
5.11 Performance of SPar-FastFlow, GrPPI-FastFlow, and handwritten FastFlow with single farm benchmarks. 133
5.12 Performance of SPar-FastFlow, GrPPI-FastFlow, and handwritten FastFlow with a pipeline of farms benchmark. 134
5.13 Usual on-demand behavior in a pipeline of farms in FastFlow. 135
5.14 Latency and throughput for the custom `pipe-farm` Ferret benchmark. It defined as `pipe(seq(source), farm((seg,ext,vect),` n_1`), farm(rank,` n_2`), seq(sink))`, where n_1 and n_2 are the number of workers in the farms. Each line in the graphs represents a different ratio of the parallelism degree of the two farms. E.g., $n_2 = 3n_1$ means that every time n_2 is increased by 3, n_1 is increased by 1. 138
5.15 Fraud detection performance results. 140
5.16 Total memory consumption of benchmarks with a single farm. 143
5.17 Total memory consumption of Ferret benchmarks using pipeline-farm (PF) and farm-pipeline (FP) compositions. 144
5.18 Number of lines of code, cyclomatic complexity, and estimated development time (PHalstead [AGS+22]) of the benchmarks implemented with FastFlow, TBB, OpenMP, ISO C++ Threads, SPar, GrPPI-static, and GrPPI-dynamic. . . . 146
6.1 Latency of Ferret TBB and FastFlow benchmarks (farm) under different data stream frequency strategies. 161
6.2 Snapshot from Figure 6.1 of a frequency switching cycle from Ferret using the binary pattern. 161
6.3 Wave frequency pattern with Bzip2, Lane Detection, and Ferret. 169
6.4 Binary frequency pattern with Bzip2, Lane Detection, and Ferret. 170

6.5 Increasing frequency pattern with Bzip2, Face Recognizer, and Ferret. 171
6.6 Decreasing frequency pattern with Face Recognizer, Lane Detection, and Ferret. 173
6.7 Spike frequency pattern with Bzip2, Face Recognizer, and Ferret. 175
7.1 Micro-batching. 180
7.2 SPBench's batch sizing flowchart. 184
7.3 Throughput and latency results of Bzip2 benchmark implemented as a farm (40 workers) with TBB and FastFlow, increasing the batch size dynamically from 1 to 10 along the execution. 185
7.4 Throughput and latency results of Ferret implemented as a pipeline of farms (maximum of 40 workers per farm) with TBB and FastFlow, increasing the batch size dynamically from 1 to 10 along the execution. 186
7.5 Latency and throughput results for Lane Detection with multiple parallelism degrees and statically set micro-batch sizes (AMD Ryzen 5 5600X). 187
7.6 Latency and throughput results for Lane Detection with multiple parallelism degrees and statically set micro-batch sizes (Intel Xeon Silver 4210). 188

LIST OF TABLES

3.1 List of available performance metrics in SPBench. 71
3.2 Main characteristics of the work related to the SPBench framework. 80
4.1 Related benchmark suites. 106
5.1 Summary table of related work regarding performance evaluation of PPIs that support stream processing in C++. 117
5.2 Overview of the computer systems used in the experiments. 119
6.1 Data stream frequency patterns found in the literature. 158

LIST OF ALGORITHMS

6.1 SPBench’s algorithm for frequency management 159
6.2 Sine wave frequency pattern 164
6.3 Binary frequency pattern 165
6.4 Increasing frequency pattern 165
6.5 Spike frequency pattern 166

LIST OF ACRONYMS

API Application Programming Interface

CEP Complex Event Processing

CLI Command-Line Interface

CPU Central Processing Unit

DAG Directed Acyclic Graph

DSL Domain-Specific Language

DSPS Distributed Stream Processing System

FF FastFlow

FIFO First In, First Out

GMAP Grupo de Modelagem de Aplicações Paralelas

GPU Graphic Processing Unit

IoT Internet of Things

JSON JavaScript Object Notation

JVM Java Virtual Machine

PPI Parallel Programming Interface

TBB Threading Building Blocks

SP Stream Processing

PF Pipeline of Farms

FP Farm of Pipelines

FLOPS Floating-point Operations Per Second

MPI Message-passing Interface

LOC Lines of Code

CCN Cyclomatic Complexity Number

CONTENTS

1 INTRODUCTION **20**
1.1 RESEARCH PROBLEM 21
1.2 RESEARCH GOALS 23
1.3 CONTRIBUTIONS 23
1.4 PUBLICATIONS 24
1.5 DOCUMENT ORGANIZATION 25

2 BACKGROUND **26**
2.1 PARALLEL PROGRAMMING 27
2.1.1 TYPES OF PARALLELISM 27
2.1.2 PARALLEL PATTERNS 27
2.1.3 PARALLEL PROGRAMMING INTERFACES 28
2.2 STREAM PROCESSING 30
2.2.1 STREAM PROCESSING APPLICATIONS 31
2.2.2 STREAM PROCESSING PARADIGMS 32
2.2.3 STREAM PROCESSING OPERATORS 41
2.2.4 TYPES OF PARALLELISM FOR STREAM PROCESSING 42
2.2.5 STREAM PROCESSING ON PARALLEL ARCHITECTURES 44
2.2.6 PARALLEL PROGRAMMING INTERFACES FOR STREAM PROCESSING 46
2.3 BENCHMARKS 49
2.3.1 TYPES OF BENCHMARKS 49
2.3.2 PROPERTIES OF BENCHMARKS 50
2.3.3 PARALLEL BENCHMARKS 52

3 SPBENCH BENCHMARKING FRAMEWORK **55**
3.1 MOTIVATION 56
3.2 SPBENCH FRAMEWORK 60
3.2.1 FRAMEWORK API 64
3.2.2 SPBENCH SEQUENTIAL BENCHMARKS 67
3.2.3 SPBENCH PARALLEL BENCHMARKS 68
3.2.4 PERFORMANCE METRICS 70
3.2.5 BENCHMARK PARAMETERIZATION 72

3.2.6 COMMAND-LINE INTERFACE 72
3.3 RELATED WORK 77
3.3.1 DISCUSSION 79
3.4 CHAPTER SUMMARY 82

4 SPBENCH APPLICATIONS AND PARALLEL BENCHMARK SUITE 83
4.1 CONTEXT 84
4.2 SPBENCH APPLICATIONS 85
4.2.1 BZIP2 85
4.2.2 FACE RECOGNITION 86
4.2.3 LANE DETECTION 87
4.2.4 FERRET 88
4.2.5 FRAUD DETECTION 89
4.3 WORKLOAD CLASSES 91
4.3.1 INPUT WORKLOADS 91
4.3.2 USING CUSTOM INPUT WORKLOADS 93
4.3.3 CORRECTNESS TESTING 94
4.4 BUILDING THE PARALLEL BENCHMARKS 94
4.4.1 FASTFLOW 95
4.4.2 THREADING BUILDING BLOCKS 96
4.4.3 SPAR 99
4.4.4 OPENMP AND ISO C++ THREADS 99
4.4.5 GRPPI 101
4.4.6 WINDFLOW 103
4.5 RELATED BENCHMARK SUITES 104
4.5.1 DISCUSSION 105
4.6 CHAPTER SUMMARY 107

5 PARALLELISM AND PERFORMANCE EVALUATION 109
5.1 CONTEXT 111
5.2 RELATED WORK 112
5.2.1 DISCUSSION OF RELATED WORK 116
5.3 EXPERIMENTAL SETUP 119
5.4 WORKLOAD CHARACTERIZATION 120
5.5 LATENCY AND THROUGHPUT PERFORMANCE 122

5.5.1 EXPERIMENTAL METHODOLOGY 124
5.5.2 TBB, FASTFLOW, OPENMP, AND ISO C++ THREADS RESULTS 125
5.5.3 GRPPI RESULTS 128
5.5.4 COMPARING HANDWRITTEN FASTFLOW, SPAR-FASTFLOW, AND GRPPI-FASTFLOW 132
5.5.5 CUSTOM PARALLEL COMPOSITIONS RESULTS 136
5.5.6 DATA STREAM PERFORMANCE 139
5.6 MEMORY USAGE 142
5.7 PROGRAMMABILITY EVALUATION 145
5.8 OVERVIEW OF THE RESULTS 148
5.9 CHAPTER SUMMARY 150

6 DATA STREAM FREQUENCY 153
6.1 MOTIVATION 155
6.2 RELATED WORK 156
6.3 DATA STREAM FREQUENCY MANAGER 159
6.4 FREQUENCY PATTERNS 160
6.4.1 FIRST PROPOSED SOLUTION 160
6.4.2 CURRENT SOLUTION 163
6.5 EXPERIMENTAL EVALUATION 167
6.5.1 EXPERIMENTAL METHODOLOGY 167
6.5.2 EXPERIMENTAL RESULTS 168
6.5.3 DISCUSSION OF THE RESULTS 176
6.6 CHAPTER SUMMARY 177

7 MICRO-BATCHING 179
7.1 MOTIVATION 180
7.2 RELATED WORK 181
7.3 PROPOSED SOLUTION 183
7.4 EXPERIMENTAL EVALUATION 183
7.4.1 EXPERIMENTAL METHODOLOGY 183
7.4.2 EXPERIMENTAL RESULTS 184
7.5 CHAPTER SUMMARY 188

8 CONCLUSION 190
8.1 LIMITATIONS AND FUTURE WORK 193

REFERENCES **195**

1. INTRODUCTION

Contents

1.1 RESEARCH PROBLEM 21
1.2 RESEARCH GOALS 23
1.3 CONTRIBUTIONS 23
1.4 PUBLICATIONS 24
1.5 DOCUMENT ORGANIZATION 25

The amount of data/information created, captured, copied, and consumed daily by us is growing like never before. More than 220 zettabytes (220 trillion gigabytes) of data will need to be processed and analyzed by 2026, as estimated by the International Data Corporation (IDC) [Ryd22]. Every day Google processes over 3.5 billion search records, NASA satellites produce about 4 Terabyte image data, and Walmart supermarkets generate over 20 million transaction records [ZLCH20]. However, only about 2% of this data remains saved or retained from one year to the next [Ryd22]. Most of this data only exists for time enough to extract relevant information and be consumed, as in many multimedia streaming applications. The consequence is that the demand for real-time data processing has grown in recent years. Traditional batch-oriented data processing is known to be insufficient to keep up with this demand [AGT14]. This way, organizations increasingly adopt stream processing systems, which can process data in nearly real-time. Many sectors use stream processing applications, such as surveillance, transaction processing, radio astronomy, signal processing, stock market, healthcare, traffic control, multimedia, and others [BGM+20, SRG+20].

Stream processing applications require parallelism exploitation to accelerate the computation and promptly process large volumes of data. The parallelism can be applied through different Parallel Programming Interfaces (PPIs) such as libraries, frameworks, and Domain-Specific Languages (DSLs). As the stream processing domain grows, so is the development of new PPIs. Besides, there is much research towards adding new features to existing PPIs and developing new technologies for stream processing through these PPIs.

Despite the growth of this area, there is a lack of stream processing benchmarks for developers and researchers to test and evaluate PPIs, techniques, and parallelism strategies [GAA+20, TSR20, Gri16a]. Even with the few existing solutions, evaluating these new technologies with different stream processing applications or benchmarks is a time-consuming task that shifts the programmer's focus away from the technology itself. This doctoral thesis proposes a framework for developing customizable benchmarks targeting

stream parallelism in C++. We expect to allow programmers and researchers to easily evaluate PPIs and new solutions for stream processing in real-world applications.

1.1 Research Problem

The research problem that this thesis addresses can be broken down into five parts:

1. It is not easy to explore parallelism in stream processing applications.

   Many of the PPIs for stream processing applications provide abstractions to make the parallelization process easier. Some are based on structured parallel programming to provide parallel structures ready-to-use for programmers [MRR12, AGT14, HSS+14]. Others go beyond and try to avoid code rewriting through compilation directives and code annotations [Gri16a, DM98]. However, stream processing applications have unique characteristics that make them difficult to parallelize by those who are not experts in this area. It is necessary to identify the application operators, which are stateless or stateful, which data is consumed or produced by each of them, etc. For instance, this problem significantly scales when a researcher needs to test a new solution in several applications. Herefore, it is helpful to have benchmarks that prevent programmers from spending time with these particularities of the applications and allow them to go direct to the point.

2. There are no representative benchmark suites for traditional stream processing.

   Representative benchmarks should be able to capture the key features and workloads of its domain's applications. There are many sub-domains of stream processing, such as data stream processing, reactive programming, and complex event processing (CEP). Also, there are some recent initiatives to build benchmark suites for these sub-domains [BGM+20, PHUK20, SCS17, LWXH14, Wan16, ABD+16]. However, for a particular type of stream processing applications such as data compression, DNA sequencing, and multimedia processing, representative benchmarks are still missing [GAA+20, TSR20, Gri16a]. The consequence is that researchers have to develop their specific-purpose solutions, often synthetic, or use outdated benchmarks [MVGF19, MTG+19, ADKT17a, ZHD+17, Gri16a]. Hence, according to our knowledge, there is a demand for initiatives that fill this gap.

3. Benchmarks for C++ stream processing are limited.

   Traditional stream processing generally relies on centralized architectures. In contrast, most data stream applications run under Distributed Stream Processing Systems (DSPSs), such as Apache Flink, Apache Storm, and Apache Spark [BGM+20,

NQA+20, GPRD19, ASAP17]. These Distributed Stream Processing Systems (DSPSs) are designed to run in the cloud or large clusters, so they use the Java Virtual Machine (JVM) to provide hardware abstractions. Therefore, applications in this subdomain are generally implemented in JVM languages, such as Java and Scala [ZHD+17, ZGQB17, GPRD19]. For this scenario, recent studies are already proposing benchmark suites [BGM+20, PHUK20, SCS17, LWXH14, Wan16, ABD+16]. However, studies have already shown the performance efficiency limitations of these Java-based systems due to the sub-optimal data serialization, memory accesses, and garbage collection. The consequence is that there is research effort towards exploring these applications in centralized single-node architectures [KWCF+16, MPJ+17, ZHD+17, ZHZH19, ZWZH20] and many of them using C++ [TKPP20, CBP+17, DMM17, MJP+19, MTG+19, RTMD20, TKPP22].

4. There is a lack of solutions targeting to ease the evaluation process of stream processing.

   Many PPIs provide high-level abstractions to improve the productivity of stream parallelism exploration by application developers. However, virtually no similar initiatives allow programmers or researchers developing these PPIs and new technologies to evaluate their solutions more easily. There are many recent studies focused on evaluating PPIs [MTG+19, GHDF18a, GHDF17, RSG+19] or developing techniques to improve different aspects of them, such as self-adaptive parallelism [VGF21, VGDF19, VGDF22], add new features [GHDF18b], and support for new parallel abstractions [HGDF20, GHDF17] and architectures [SRG+20, RTMD20, RGDF19, RSG+19, RLA+22].

5. Performance analysis of PPIs for C++ stream processing is usually incomplete

   Properly evaluating the performance of stream processing applications and PPIs can be challenging [KRK+18]. Many works in the literature consider only speedup or throughput as a performance metric when evaluating PPIs in the traditional stream processing domain. However, SP applications can be tuned to achieve different goals, such as reduced latency, increased throughput, or efficient resource usage [KRK+18, ZMK+19, SRG+20]. Latency can be a critical factor for applications such as lane detection, object tracking, high-frequency trading, augmented reality, anomaly detection, online games servers, etc. [DDMMT15, NXC19, LLG19, YLL+22]. Also, this scenario of possible endless data streams can significantly impact the memory usage of different applications and PPIs [TKPP20, MJP+19]. When comparing PPIs, analyzing their usability/programmability is also an important factor often overlooked [AGSF23]. Nevertheless, all these aspects are often neglected in the literature concerning evaluating C++ stream processing PPIs.

1.2 Research Goals

In this doctoral thesis, we go into the performance analysis of stream processing systems. We found unfilled gaps in this area regarding representative benchmarks, especially for lower-level abstraction programming languages such as C++. This way, we aim to provide a means for developers and researchers to more easily test, validate, and address the performance of systems, techniques, and technologies for stream processing. Therefore, we conduct our research in this direction, and the main research goals can be summarized as follows:

- Ease the benchmarking of C++ parallel stream processing.
- Speedup and simplify the research for parallel stream processing by providing highly parameterizable benchmarks with self-built representative mechanisms, such as batching, data stream frequency management, and real-time performance metrics.
- Conduct a comprehensive evaluation and comparison of the state-of-art PPIs that support parallel stream processing in C++.

1.3 Contributions

Based on the research challenges and goals discussed above, we provide the following main scientific contributions in this doctoral thesis:

- A user-friendly framework designed to facilitate the benchmarking of stream processing applications. With its intuitive API and command-line interface, this framework streamlines the process of customizing, extending, and evaluating various aspects of parallel programming and architectures relevant to stream processing. By reusing code, users can easily compare and optimize different parallelism strategies, enhancing their ability to achieve optimal performance (Chapter 3).
- A parallel C++ benchmark suite for stream processing. This suite encompasses real-world applications and incorporates most of the state-of-art PPIs available in this context, making it a valuable resource for the industry and scientific community (Chapter 4).
- A comprehensive comparison and analysis of parallel programming interfaces that harness stream parallelism, which includes Intel TBB, FastFlow, GrPPI, SPar, OpenMP, ISO C++ threads, and WindFlow. Through our thorough analysis, we aim to provide researchers and developers with insights into the relative strengths and weaknesses of each interface in the context of stream processing (Chapter 5).

- Mechanisms for data stream frequency simulation in stream processing applications. We provide a set of algorithms for generating the literature's most commonly used data stream frequency patterns. These mechanisms are fully integrated into our framework and can be dynamically reconfigured at runtime, allowing for flexible evaluation of the impact of data frequency on stream processing applications. Using these algorithms, we conduct a comprehensive study of the effects of data frequency on stream processing applications (Chapter 6).

- An analysis of the performance impact of micro-batch sizing on stream processing applications. Additionally, we introduce mechanisms for real-time and dynamic batching management, allowing users to adjust batch sizes on-the-fly either based on specific size targets or time intervals. By providing this flexibility in micro-batch sizing, our framework enables users to optimize their stream processing applications for varying workloads and use cases (Chapter 7).

1.4 Publications

Most of these contributions have already been presented and published in international conferences and journals. The followings are the research papers and articles accepted during the doctorate period that are directly related to this study:

Journal articles:

- Garcia, A. M.; Griebler, D.; Schepke, C.; Fernandes, L. G. "**SPBench: A Framework for Creating Benchmarks of Stream Processing Applications**", Computing [GGSF22b].

- Garcia, A. M.; Griebler, D.; Schepke, C.; Fernandes, L. G. "**Micro-batch and Data Frequency for Stream Processing on Multi-cores**". The Journal of Supercomputing [GGSF23].

Conference papers:

- Garcia, A. M.; Griebler, D.; Schepke, C.; Fernandes, L. G. "**Introducing a Stream Processing Framework for Assessing Parallel Programming Interfaces**". In: 29th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP) [GGSF21].

- Garcia, A. M.; Griebler, D.; Schepke, C.; Fernandes, L. G. "**Evaluating Micro-batch and Data Frequency for Stream Processing Applications on Multi-cores**". In:

30th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP) [GGSF22a].

- Garcia, A. M.; Griebler, D.; Schepke, C.; Santos, A. S.; García, J. D.; Muñoz, J. F.; Fernandes, L. G. "**A Latency, Throughput, and Programmability Perspective of GrPPI for Streaming on Multi-cores**". In: 31st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP).

Moreover, here is the work published during the Ph.D. course that is not directly connected to this thesis:

- Title: **The Impact of CPU Frequency Scaling on Power Consumption of Computing Infrastructures**. Reference [GSG+20b].

1.5 Document Organization

The remainder of this proposal is organized as follows: Chapter 2 describes the background of this study. Chapter 3 introduces the motivation, presents SPBench benchmarking framework, and discusses our related work. Then, Chapter 4 shows the set of applications supported by SPBench and how to use the SPBench framework with different parallel programming interfaces. Chapter 5, Chapter 6, and Chapter 7 present performance and parallelism evaluation, tests with distinct data stream frequency patterns, and the impact of micro-batching size in throughput and latency performance. The conclusion is in Chapter 8.

2. BACKGROUND

In the last decades, parallel architectures have evolved significantly, leveraging the increased popularity of parallel programming. The focus of parallelism is to enhance the performance of applications. On the other hand, stream processing offers a solution for processing data streams continuously without requiring knowledge of the stream's size or end. The high demand for real-time processing and the increased volume of data motivates the need to improve the parallelism capabilities of stream processing systems. This incurs a lot of research and development of new solutions and technologies in this direction. Whether in stream processing or any other domain, benchmarks are crucial for the academic field and industry to evaluate and validate such technologies. Section 2.1 discusses the most common ways to explore parallelism through parallel programming techniques. In Section 2.2, we give a detailed background on stream processing. Finally, Section 2.3 presents some concepts and challenges regarding benchmarks.

Contents

2.1 PARALLEL PROGRAMMING 27
2.1.1 TYPES OF PARALLELISM 27
2.1.2 PARALLEL PATTERNS 27
2.1.3 PARALLEL PROGRAMMING INTERFACES 28
2.2 STREAM PROCESSING 30
2.2.1 STREAM PROCESSING APPLICATIONS 31
2.2.2 STREAM PROCESSING PARADIGMS 32
2.2.3 STREAM PROCESSING OPERATORS 41
2.2.4 TYPES OF PARALLELISM FOR STREAM PROCESSING 42
2.2.5 STREAM PROCESSING ON PARALLEL ARCHITECTURES 44
2.2.6 PARALLEL PROGRAMMING INTERFACES FOR STREAM PROCESSING 46
2.3 BENCHMARKS 49
2.3.1 TYPES OF BENCHMARKS 49
2.3.2 PROPERTIES OF BENCHMARKS 50
2.3.3 PARALLEL BENCHMARKS 52

2.1 Parallel Programming

Parallel architectures have been around for a long time, but they alone cannot meet performance needs. Traditionally, computer programs are written to run serially. Algorithms are built as a sequence of instructions. Then, a computer's processor executes these instructions one at a time. Despite the ubiquity of parallel architectures, most computer applications are still sequential [Gri16a]. With these architectures, it is possible to use parallel programming to make sets of instructions run concurrently. This is done by separating a problem into smaller, independent parts that can execute simultaneously.

2.1.1 Types of Parallelism

Several types of parallelism are generic and universal in parallel programming. The two main ones are data parallelism and task parallelism [Tor19]. The former consists of the replication of code blocks so that the workload is partitioned between these blocks, and these blocks are processed concurrently. Task parallelism consists of executing different concurrently independent blocks of code in parallel and is usually defined through task-dependency graphs. However, the exact definition of task parallelism is nebulous, and its meaning may vary. According to [MRR12], while some programmers use it to mean (unscalable) functional decomposition, others use it to mean (scalable) recursive fork-join, and some even mean any parallelism with distinct tasks in control flow. In addition, if we consider that the input of an application can be a flow of data items, and depending on how the graph is designed, this model can be considered stream parallelism [ATM09]. We present this type of parallelism in more detail in Section 2.2.4.

2.1.2 Parallel Patterns

The idea of structured parallel programming comes from design patterns. It transforms recurrent ways of solving computational problems into well-established patterns to simplify programming. However, defining patterns to facilitate programming is not a recent idea. This has already been considered with other names, such as algorithmic skeletons [Col89, Col04] and parallel patterns [MRR12]. This kind of design patterns approach is widely discussed and used in software engineering. It has been imported from other domains, such as architecture and city planning [MSM04]. In computing, it has become prominent mainly in object-oriented programming [Gam95].

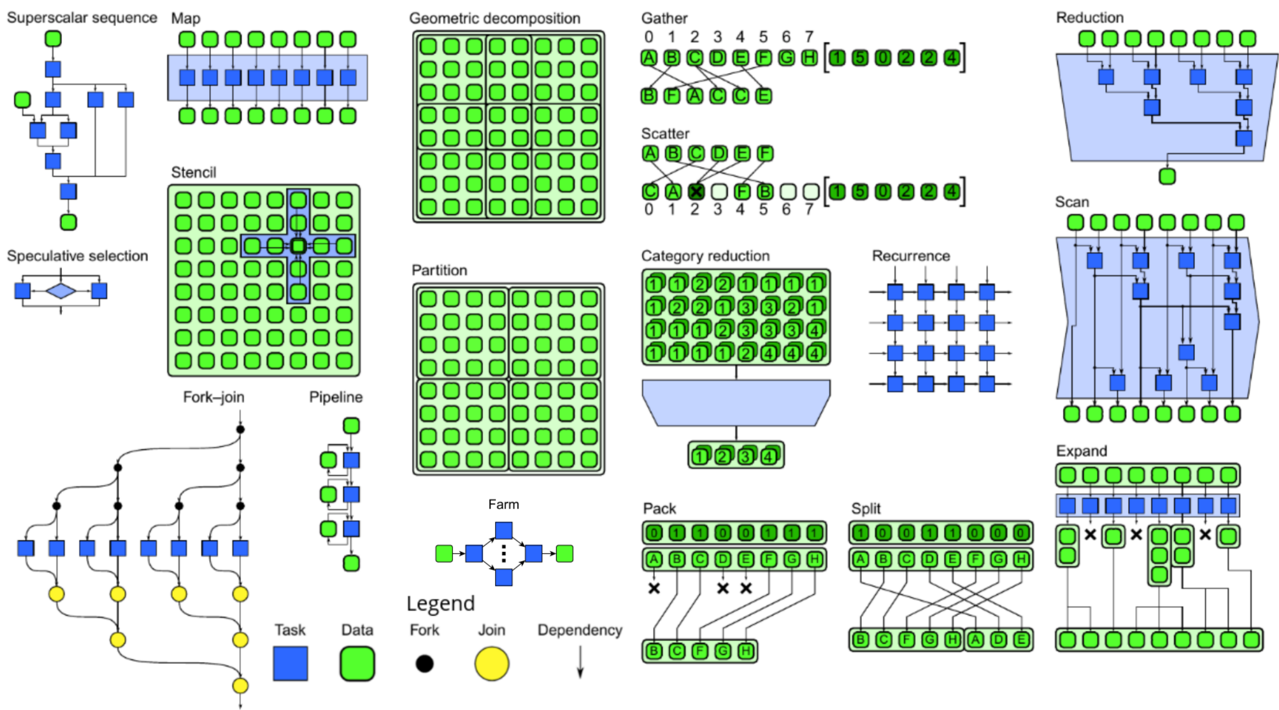


Figure 2.1: Overview of parallel patterns.

Source: [MRR12]

This approach that follows the design patterns path is handy for programmers who aim for parallel systems. Among the benefits we can list [Col89]: (a) simplified programming; (b) increased portability and reuse; (c) improved performance; and (d) allowing more automatic optimizations. Figure 2.1 shows an overview of the main parallel patterns, like map, farm, pipeline, and fork-join. We discuss the relevant patterns for this thesis in Section 2.2.4.

2.1.3 Parallel Programming Interfaces

With all the performance potential of parallel architectures, parallel programming was expected to be more popular among application developers. However, using parallelism to extract all this potential requires a deep understanding of micro-structural aspects of the architecture [MRR12]. Although there are mechanisms of automatic parallelism, such as vectorization (an extension of data width parallelism), it has not been universally successful [MRR12]. This automatic hardware parallelism has low portability because the performance relies on a specific architecture. Thus, it is still necessary that programmers use explicit parallelism to explore the hardware. However, the cost of higher portability and performance is a higher programming complexity, which implies that it is necessary to have a balance to make it worthwhile.

None of the most widely used programming languages are designated for parallel programming [MRR12]. In addition, a lot of legacy code is serially implemented in these languages. Therefore parallel programming interfaces (also known as parallel programming models) must provide support and tools to explore parallelism more independently. Their goal is to provide abstractions for hardware, system, and languages. These PPIs can take many forms, such as domain-specific languages, libraries, and language extensions. Some examples are OpenMP, Message-passing Interface (MPI), TBB, CUDA, and FastFlow. Each PPI has a target, which can be a specific architecture, a language, application classes, and types of problems. However, despite assuming so many different forms, [MRR12] and [Tor19] say that these PPIs should balance 3 parallelism properties:

Programmability/Productivity: this property concerns the programming complexity of reaching a solution. Two main aspects are considered: the time spent by programmers and the effort required to write the code. In the ideal scenario, parallelism adds no programming cost, and programmers should face the same complexity of writing sequential code. However, it is challenging to achieve this because programming overheads are added by the management of synchronization, communication, and task scheduling [Tor19]. The extra code these mechanisms add for concurrency and parallelism control keeps PPIs away from the ideal scenario. In addition, programmability also refers to the ability to reuse code. Many sequential codes result from years of improvements and tuning, where rewriting code would be out of the question. Therefore PPIs need to have the ability to apply parallelism using this same highly optimized code with as little rewriting as possible. Additionally, [MRR12] says that a PPI should be: “Expressive, composable, debuggable, and maintainable”. Addressing these issues regarding programmability/productivity can bring parallelism to application developers.

Portability: This refers fundamentally to the ability of the code to run on different platforms. This can be done using standard libraries that will ensure this higher level of compatibility. However, even if a parallel program can run on another architecture without rewriting code, this does not guarantee so-called “performance portability”. In reality, it means the ability of a parallel application to maintain a performance level considering the relative peak performance on different architectures. An application tuned to run at 80% performance on one machine should not run at only 30% on another [MRR12]. However, this has many challenges, such as other parallel applications running on the new system. While [Tor19] argues that code rewriting is still a necessity for performance portability, [VGF21] and [DSTD16] argue that this portability can be improved through the use of self-adaptivity to autonomously managing parallelism aspects. In addition, [MRR12] says performance portability is usually only achievable with high abstraction parallel interfaces, as there are fewer manual parallelism inserts. These PPIs could identify hardware and adapt better.

Performance: according to [MRR12] the performance of a PPI should be “achievable, scalable, predictable, and tunable”. It means it should allow performance to be achieved in a predictive and scalable way in large systems. It is usually the primary target of any PPI, where performance improvement Uis expected to be proportional to the system’s processing power. However, this improvement is challenging to scale optimally for large systems, as there is a limit to how much an application can be parallelized. Amdahl’s law [Amd67] describes this limit, which refers to the non-parallelizable parts of a program. Even in highly parallelizable applications, intrinsically serial parts will be impossible to avoid. For example, programs with high data dependency usually have several sequential snippets. On the other hand, Gustafson’s law [Gus88] says that parallel regions grow faster than sequential ones when you scale up the problem. It means that the limit defined by Amdahl’s law may vary according to the scale of the problem. We add to these aspects that it is also important for the parallel code to be written dynamically to scale and exploit the resources of different systems. Furthermore, self-adaptive techniques can be employed to help in this regard [VGF21].

2.2 Stream Processing

We are living in a data-oriented world. The Big data within the expansion of Internet of Things (IoT) technologies and the emergence of Edge Computing have made stream processing applications widely used in many areas and industries. This type of application needs to handle large volumes of user and sensor data. Stream processing contrasts with traditional batch processing because there is no need for data storage here, and processing is done in near real time [AGT14]. In batch processing, data are collected over some time. This data is eventually processed and sent to an analysis system that runs periodically. In stream processing, data is processed continuously as new data becomes available for analysis, applying a series of small computations as stages in a pipeline, doing this processing incrementally [AGT14, SHGW15].

According to [HSS+14], stream processing systems allow the execution of the stages of an application in parallel, making the communication of data items between them through FIFO queues. Each item in the stream is an atomic piece of data, which can be consumed or produced by the stages [SHGW15, AGT14]. Therefore, stream processing also has three advantages [AGT14]: 1) for programmers to develop parallel programs, they do not need to think parallel because they can implement individual blocks (stages/operators), keeping the sequential programming logic. 2) a streaming run-time can reason about each stage individually while potentially optimizing globally. 3) it enables an application to process a constant stream of data over time, optimizing the use of hardware resources.

2.2.1 Stream Processing Applications

Different sectors present stream processing applications. Figure 2.2 presents some of them, such as surveillance systems, financial transactions, radio astronomy, signal processing, fraud detection, stock market, healthcare monitoring systems, and traffic control [AGT14]. Although most recent literature uses the term “stream processing” to refer to applications that process infinite streams of real-time data, this is a more specific feature of “data stream” applications. Conversely, some applications do not necessarily process unbounded real-time data streams within the stream processing domain, and latency is often not a critical factor. This class of application, such as data compression, industrial simulation, DNA sequencing, data deduplication, general video, audio and image processing, etc [TA10, TZ14, MVGF19] most target a high-throughput rate instead.

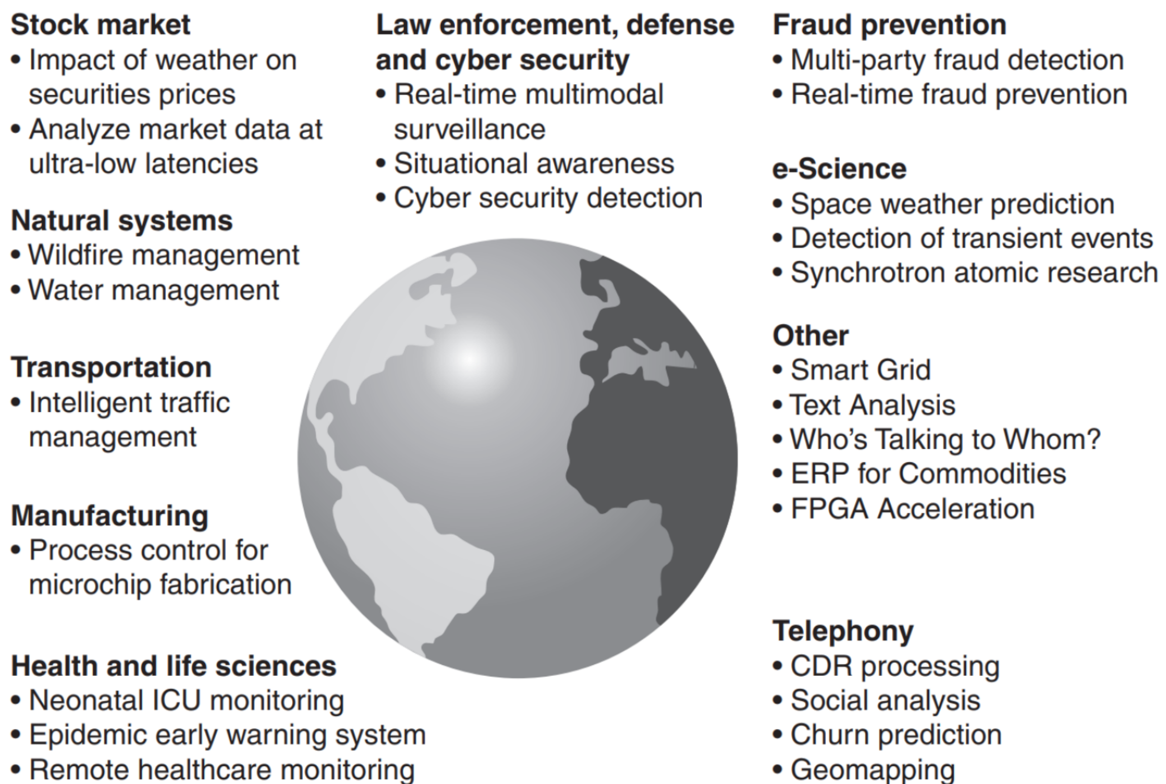


Figure 2.2: Stream processing applications.

Source: [AGT14]

Stream processing applications also need to work with different types of workloads, which impacts the applications’ characteristics. Data streams can contain data of different types from different sources. This data can constantly arrive, irregularly, or sporadically. Data items can be small as a single numeric attribute or be sets of information with thousands of bytes. Although stream processing targets real-time processing, this data does not necessarily arrive in real-time, as with a file compression application. Figure 2.3 shows examples of data types processed by applications in this domain. As we will see in

the following sections, some applications need to process combined workloads to find more complex solutions.

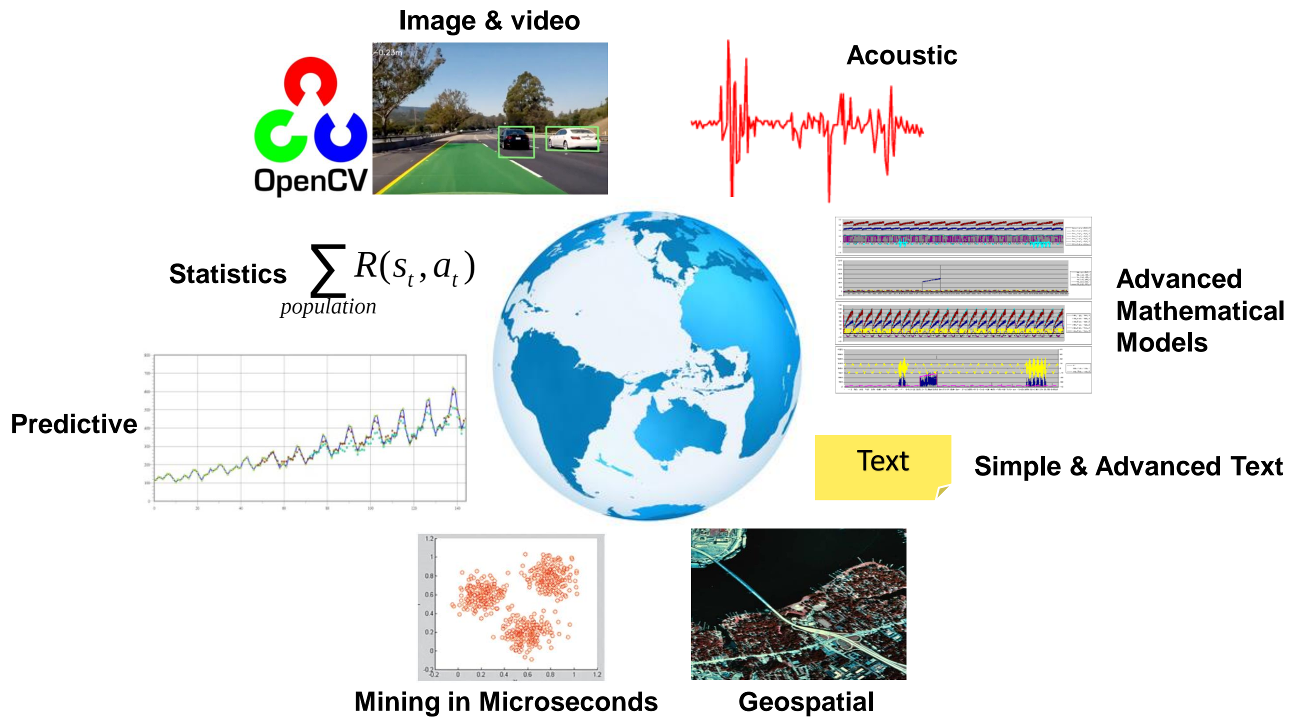


Figure 2.3: Stream processing workloads.

Source: Adapted from [BBD+14]

2.2.2 Stream Processing Paradigms

There are, therefore, many definitions of “stream processing”. Back in 1997 [Ste97] said that stream processing could refer to dataflow systems, reactive systems, synchronous concurrent algorithms, signal processing systems, and certain classes of real-time systems. We have analyzed 70 surveys of this domain published in the last decade and extracted the terms used to designate applications, tools, and technologies for stream processing. Figure 2.4 presents a word cloud based on the frequency that these terms appeared in these works. Most terms refer to data stream processing. Although “stream processing” has a high frequency, we have found that in most cases, it also designates data stream processing.

Some scientists and experts say that many of these variations are just attempts by the industry to sell a new product and that they are equivalent in many ways [AGT14, MDAT17, Car14]. It is true partly because this area has grown very recently, which can be somewhat confusing even for scientists and experts. Thus, a broad taxonomic study to unify knowledge and set some boundaries in this area would be interesting. However, although

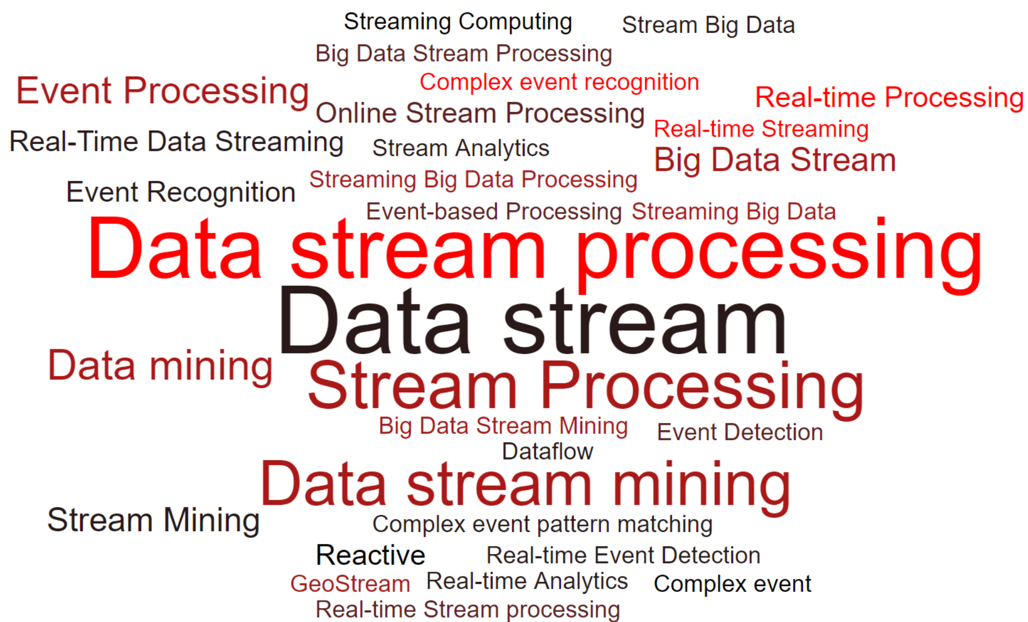


Figure 2.4: Most common terms for stream processing applications and technologies found in 70 surveys published between 2008 and 2020.

paradigms share some aspects, some specific characteristics make some definitions differ from each other, such as the structure of the workload and individual data items, the data propagation strategy, the goals, the architecture, and others. Since, in the scope of this work, we address certain well-defined types of applications, we believe it is valuable to discuss the main paradigm definitions and how they differ. In the following sections, we categorize the paradigms according to our understanding.

Dataflow Paradigm

Dataflow is a broad concept in Computer Science that can have different meanings. A broad description could be “information in motion”. Some scientists say that any system where the data moves between operators and triggers their execution could be called dataflow [Car14]. Here we will approach the concept of dataflow programming, a software architecture, in contrast to dataflow hardware architecture. This programming model can be easily represented by a high-level abstraction acting on different layers, ranging from the definition of the basic structure of an application to the detailed execution model [MDAT17]. It can describe how data flows through a program’s operations, represented by a Directed Acyclic Graph (DAG). In these graphs, the nodes are the computing part and are called operators, while the edges represent the data dependencies. Therefore, the functional part of this representation is the operators, which consume input data, process it, and generate output data. Operators without input or output ports are called “source” and “sink”, respectively. This way, dataflow can be seen as a way to program, and it is not tied to a specific language or technology. It does not guarantee parallelism but makes it much more straightforward.

The dataflow models independent (therefore parallelizable) operators starting from a graph of true data dependencies, and each operator has its execution triggered by data availability [MDAT17]. Figure 2.5 shows a representative illustration of this paradigm, where the spheres are operators and arrows are dependencies [Gri16a]. This means an operator will only be executed when the input data is available. Similarly, an output dependency defines where the following operators will get the data from. Therefore, these input and output dependencies represent and ensure the connections and synchronizations between operators [Gri16a].

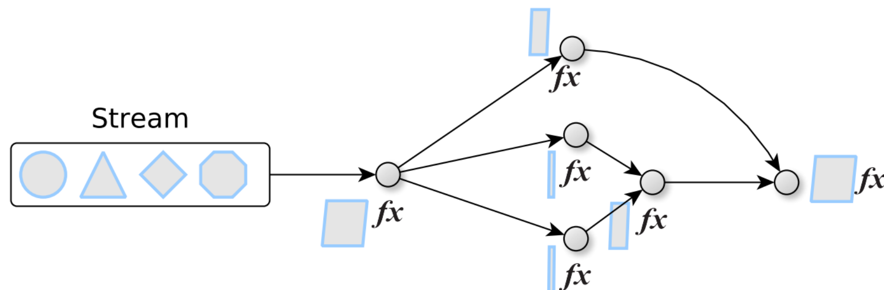


Figure 2.5: A dataflow system representation.

Source: [Gri16a]

The operators of a dataflow program can be executed in two different ways [MDAT17]. The first way is called process-based and is a straightforward model where each operator is assigned to a thread/process, and communication occurs through FIFO queues (First In, First Out). The second way is the scheduling-based model, where a scheduler tracks when a data item satisfies an operator's dependency and then schedules that operator to be executed. Besides, data items can also be propagated in different ways through operators using the synchronous or asynchronous model [Tec01]. The asynchronous model is most commonly used in stream processing and was the model we have addressed so far, where data consumption/production by the operators occurs dynamically. On the other hand, in the synchronous model, the number of items that will be consumed and produced in each operator, and the instant of time they may consume/produce items, is done synchronously [LM87]. It means that it is implemented statically and is known at compiling time. This model is suitable for synchronous signal processing systems and does not allow flow control based on data dependency.

Although the dataflow concept first appeared in the 60's [KLV61, Sut66, Den74], it has evolved in a way that can still represent current programming models. [MDAT17] analyzed modern data stream processing PPIs within a dataflow perspective. They built a layered model to show how the dataflow paradigm can present itself at different levels of abstraction in these PPIs. Three layers of dataflow abstraction were identified: 1) Semantics of the application in terms of dataflow graphs; 2) Instantiation of semantic dataflow that explicitly expresses parallelism; 3) How the program is effectively deployed and executed

onto the platform (e.g., using a Master-Workers pattern). Thus, the dataflow paradigm can still represent the streaming models used for stream processing [MDAT17]. However, as we discussed next, so many other specific variations of this model have emerged that we believe it should now be seen more like a feature or optimization rather than a programming paradigm.

Reactive Programming

The high interactivity of modern applications, including with the surrounding environment, opens space for solutions that react to these stimuli quickly and efficiently. These interactions are usually linked to user interfaces and can be mouse clicks, keyboard button presses, multitouch gestures, etc. [BCC+13, Ell09, KBDM13]. An interaction of this type can generate an event that triggers a series of operations, such as updating the state of an application with new information or generating some alert. Ideally, this processing should be done in real-time, or at least appear real-time according to the context of the application environment [BCC+13]. For this type of application, reactive programming may be the most appropriate paradigm.

Although reactive programming is a relatively new term, it is a new name for old concepts. Its origin stretches back to at least 1985 [HP85]. It has gained popularity in recent decades as a way to simplify the creation of interactive user interface animations in real-time systems. This paradigm emerged around the asynchronous dataflow model and has the principle of “propagation of change” [BCC+13]. That is, it is an event-driven paradigm, unlike dataflow, which is data-driven. Therefore, reactive programming allows the user to express what the program should do, using static or dynamic flows. Thus, instead of waiting for data to arrive, programmers can implement callbacks (event handlers) that only activate execution when a specific event occurs.

The reactive programming model, therefore, can be represented in the same way as a dataflow, through a graph with its operators and dependencies. A reaction to an event causes an automatic propagation of change through all the dependent nodes in the graph [Ell09]. Consequently, any computing affected by this change becomes outdated and needs to be flagged for re-execution. This change can spread through the entire graph to a sink operator.

Different strategies can be used to implement a propagation system with reactive programming, such as “pull”, “push”, or “push-pull”, as illustrated in Figure 2.6. In the pull strategy, the operators must be proactive and constantly check status changes in the data stream. As soon as an event is identified, a reaction occurs. This strategy does not fit very well with most reactive systems, as it can increase latency, a critical factor in this paradigm [Ell09, BCC+13]. In the other strategy, named pull, the operators are passive and wait for the data to arrive. It automatically triggers its execution. This model is more

suited to most reactive programs because these programs receive events only at discrete points of time instead of a continuous flow [NCP02, Ell09]. So it is a waste of resources to recompute values even when they do not change. In addition, with the push strategy, the reaction is practically instantaneous.

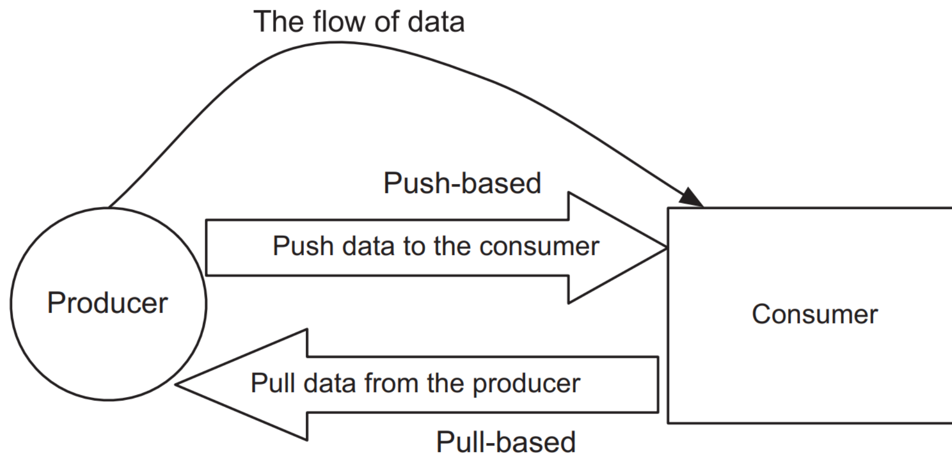


Figure 2.6: Push and pull data propagation model.

Source: [BCC+13].

A third propagation strategy called push-pull is a combination of the last two. Here, the operators receive only a small change of state notification along with a short description of the event (this is the push part). If the operator recognizes a data dependency, he needs to go to the source of the notification to get all the data (this is the pull part). This last strategy is suitable for reactive systems that need to deal with large volumes of data because notifications are light elements, and only interested consumer operators will make a data request [BCC+13].

Applications implemented with the reactive programming paradigm exist in several sectors. However, they are more common among applications that need to respond to requests, such as network request events for exchanging data with background services, or interactions, such as a user interface, which needs to be highly responsive to clicks and other interactions, coordinating such events originated from a user with internal application state changes. The classic example of reactive programming in our daily lives is spreadsheets. [BCC+13] states that reactive programming is essentially about embedding the spreadsheet-like model in programming languages. According to [GJ21], spreadsheet languages like Excel are the most used programming languages in the world. In a spreadsheet, cells can contain literal values or formulas calculated using values from other cells. A formula can be =B1+C3, for example, in which case the cell containing that formula depends on the values of cells B1 and C3. Therefore, when any of these cells has its value updated, it triggers the execution, and the program recalculates the formulas and automatically updates all the values dependent on these cells.

The need for processing can significantly escalate when using large data sets and more complex spreadsheet formulas. In addition, this complexity tends to increase as new technologies emerge. Until then, spreadsheets only allowed the use of pre-programmed functions. However, spreadsheet languages are now becoming full-fledged programming languages, allowing users to create custom functions, such as LAMBDA for Excel [GJ21]. Therefore, reactive programming is essential for evolving this type of technology. Other day-to-day applications that use this paradigm are search engines, text translation tools, social media subscription systems, etc. [KBDM13, BCC+13].

Data Stream Processing (Event Processing)

In the previous section, we discussed the concept of reactive programming, which processes events that occur at discrete points in time. However, to process continuous flow events, other paradigms, such as event processing or data stream processing (they can be seen as equivalent) are necessary. While in reactive programming, stream processing is only enabled when a specific event is identified, Data Stream Processing Systems (DSPSs) need to handle events that arrive continuously, usually carrying meaningful information to be processed. The data items can be called “events” because they arrive as tuples of data, which often carry temporal and spatial data together with the main information [ZGQB17]. Although this is not true for every kind of data tuple, the structure of these data items is quite similar. Applications of this paradigm are implemented using an asynchronous dataflow model and can also include reactive programming.

The data stream has emerged as a solution to process tremendous volumes of data arriving at high velocity and avoid the need for higher storage capacity [AGT14]. The goal is to extract relevant information from these data uninterruptedly as soon as they arrive. Data stream applications, therefore, need to operate mostly infinitely, because the end of a stream is usually not predictable [YT13]. A data stream is basically defined by the structure of its tuples. A tuple is a list of attributes of different types, which usually encapsulates the main information along with the time it was generated, where it was generated, and any additional information. Therefore, we can say that data streams are temporally ordered, fast-changing, massive, and potentially infinite sequences of data [YT13].

A typical data stream system consists of a set of data/events producers and a set of consumers [GPRD19, HK19]. The data produced can be clicks on social networks, financial transactions, stock market feeds, medical systems, traffic reports, sensor data, etc. [AGT14, NNG18, GPRD19]. Consumers in a data stream application can handle this data and take some action. Such actions can include aggregations (e.g., calculations such as sum, mean, standard deviation), transformations (e.g., changing a number into a date format), enrichment (e.g., combining the data point with other data sources to create more context and meaning), data analytics (e.g., predicting a future event based on patterns in

the data), or simple ingestion (e.g., inserting the data into a database). In Figure 2.7 we present a classic example of a data stream application, which receives a tweets stream, processes the text to extract hashtags, counts the most relevant items, and uses the result to update the trending topics list [HK19].

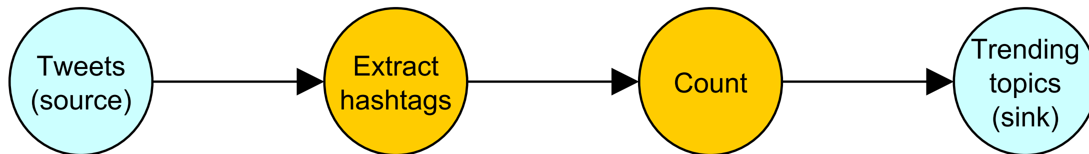


Figure 2.7: A data stream application to continuously count hashtags. Adapted from [HK19].

The domain has grown a lot toward Data Stream, especially with the world becoming data-driven, with IoT and new advances in distributed computing, including broader access to cloud computing [NQA+20]. Data stream applications are becoming more and more present in everyday life as they are used to process events generated by all kinds of sources, such as sensors, surveillance systems, and social media [DBL+19]. There are applications to classify audio information and identify specific events, such as gunshots, cheering (to create indexed records in sports), music, speech recognition, etc. [NQA+20, DBL+19, AGT14]. In monitoring systems, they can act as anomaly detectors, such as anomalies in a security area, congestion in highways, anomalies in an industrial production line, etc. [DBL+19, AGT14]. In social media, it can recognize and classify events, people, and objects in images, run recommendation systems, check for rule violations in posts, etc. [NQA+20, HK19]. Therefore, the list of applications that are implemented with this paradigm is endless, here we just scratch the surface.

Complex Event Processing

Complex Event Processing (CEP) can be seen as a composition or extension of simple event processing that we discussed in the previous subsection. CEP applications, besides processing and identifying events, interpret and combine information or different events to derive conclusions from them. Usually, these applications receive events from multiple data sources, often containing data streams of different types, and search for patterns and data dependencies [HK19]. The aim of this paradigm is to identify a meaningful event (such as an opportunity or threat) from several basic events [TMM+16, THR+18].

CEP applications interpret information and events and see these as events in the physical world [TMM+16]. Here events are filtered, combined, and transformed into higher-level events, trying to make information understandable for computers but also for humans [THR+18]. CEP can interpret and draw relationships between different data, such as audio, text, image, video, weather, location, and sensor data [HZEF16]. In contrast to event processing, CEP analyzes an entire context to identify an event.

As well as event processing, CEP applications have become quite popular with the advent of IoT [NQA+20]. Applications that produce events, such as sensors in a smart home or smart city, can generate a large volume of events that need to be processed with low latency [NNG18]. A CEP application, for example, can monitor temperature sensors and smoke detectors to identify a complex event, such as “fire”, and automatically trigger a fire alarm, call the fire department, and notify residents or the homeowner [NNG18]. In larger environments, such as a factory or even a forest, such an application could also add geospatial information to facilitate the identification of the fire outbreak [HZEF16, NNG18].

Complex event processing has applicability in many domains [TMM+16, ASAP17, THR+18, NQA+20, ICDV15]. An application can monitor audio and video to identify suspicious movements, objects (such as a gun), and sounds (such as shouting, shooting, or breaking into sound) to identify a crime [TMM+16]. In healthcare, it can analyze audio, video, and wearable devices to monitor elderly activities, identify anomalies, such as a fall or heart attack, and automatically trigger notifications and call the healthcare system [ASAP17]. In social media, a CEP application can use geospatial text and data to identify crises, environmental catastrophes, epidemics, terrorist attacks, etc. [ICDV15]. It could also identify patterns in user images to automatically label photo albums, such as “wedding” or “birthday party” [ASAP17]. In online video platforms, this type of application can recognize events and place tags and labels or recognize speech for automatic subtitles, or even check audio and video patterns to identify copyright issues [THR+18]. In agriculture, it is possible to analyze sensors in the field that check soil moisture and compare it with weather forecast events to make an efficient irrigation control. Therefore, this type of application has great potential in this data-driven world.

Traditional Stream Processing

Although the term “stream processing” is also valid to refer to the paradigms described in the previous subsections, it has a broader meaning. However, there is a type of application that is strictly related to the stream processing paradigm, which inherits some characteristics of the previous paradigms but does not fully fit into any of them. These applications are not data-driven, so the dependency between operators is not defined by input or output data specifications [Gri16a], as opposed to Dataflow. They are also neither event-driven nor reactive, as they process simpler items of data that arrive continuously and in large volumes, usually from a single data stream source. These applications also tend to process bounded or unbounded data streams. Here the data items are not necessarily generated in real-time, like the events generated by sensors that we discussed in the previous paradigms. However, we have not found in the literature an adequate nomenclature to differentiate this type of application from the other paradigms. Therefore, in this work, we use the term “*traditional stream processing*” to designate it.

In traditional stream processing, the sequence of operators is structured as a pipeline model so that the dependencies are explicit. Therefore, the data flow in the execution graph is static, and each operator's input and output data only describe what will be produced and consumed. Figure 2.8 illustrates the model of this pipeline graph, where a kernel contains one or more operators that will compute each data item of the stream. An operator comprises an operation or set of operations, which must be performed sequentially and locally within each kernel during program execution. These local operations avoid global data manipulation and can increase memory performance through data locality [Gri16a].

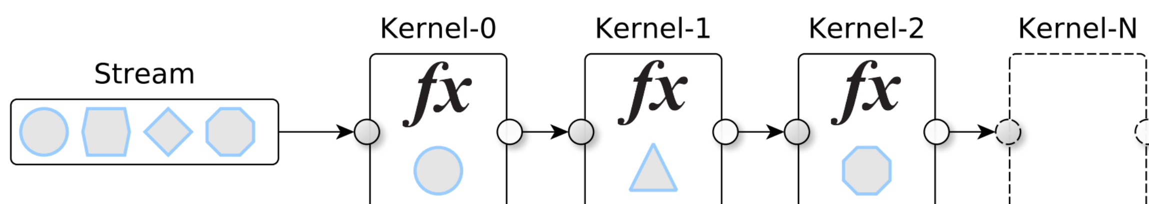


Figure 2.8: Stream systems representation, from [Gri16a].

Communication between operators is often made through FIFO queues. Therefore, output data items are pushed into this queue and operators need to pull them when they are ready to do the computation. The characteristic of these queues is very important for the application's performance. Queues larger or smaller than necessary impact memory consumption and can increase application latency. Also, the policy for accessing these queues can vary, so it needs to be well-balanced. Because this type of application has a deterministic behavior defined at compile-time, it facilitates the implementation of a task scheduler to execute the kernels. In Dataflow, for instance, the sequence of operators is defined by data dependencies, so the flow graph is non-deterministic and makes it more complex to develop a task scheduler [Tec01, Gri16a].

The characteristics of the data stream also differentiate traditional processing from other paradigms. Streams with an undefined end are difficult for DataFlow systems [Gri16a], for instance. On the other hand, although reactive systems can handle these unbounded streams, events arrive elastically at discrete points of time [NCP02]. This occurs similarly in data stream processing and CEP, where the sources of the data and event streams often produce events at irregular frequencies [GPRD19, ZGQB17]. However, in traditional stream processing, data arrives in large volumes at high speed, and the stream is potentially infinite. Instead of event-driven data tuples, here applications usually process raw data, and a data item can be an image, a video frame, a string, a block of bytes, etc. Moreover, often this data does not arrive in real time. In astronomy, for instance, an application can use stream processing to increase the performance of pattern search on telescope images collected over decades.

Therefore, the basic structure of a traditional stream processing application comprises a source operator, n intermediate operators, and a sink operator. Many real-world applications have these characteristics and belong to this paradigm. Among them, we

can mention network packet routing, face tracking, image processing, video streaming, person recognition, lane detection, data compression, signal processing, etc. [TA10, TZ14, MVGF19, Gri16a]. Although these applications represent a large part of the stream processing domain, they are not much discussed in the literature, as evidenced in Figure 2.4. Therefore, in our work, while we address other paradigms, such as data stream/event processing, we are mainly contributing to the traditional stream processing domain.

2.2.3 Stream Processing Operators

Operators are the most basic functional units of a stream application. They may have other names in literature, such as filters, threads, kernels, bolts, stages, etc. [AGT14]. An operator receives a data item, applies an arbitrary function to it, and outputs the item to outgoing streams. The exceptions are the source and sink operators, which only produce or ingest streams, respectively. A source operator does not have an input port like the others because it usually receives raw data from external sources and turns it into data items downstream operators can consume. The reverse logic can be applied to the sink operator. [AGT14] says that a sequence of operators organized in graphs sharing data items via streams is the backbone of any stream processing application.

In stream processing, there are two main types of operators: stateless and stateful. Their basic behavior is represented in Figure 2.9. Stateless operators do not need to keep information (history) from previous items. They usually apply some filter or transformation and pass the item forward. In the trending topics application's graph in Figure 2.7, the operator that extracts the hashtags is an example of a stateless operator because this type of operation does not depend on previous information. It only extracts the hashtags from the tweet, discards the unnecessary text, and sends them individually in the output stream.

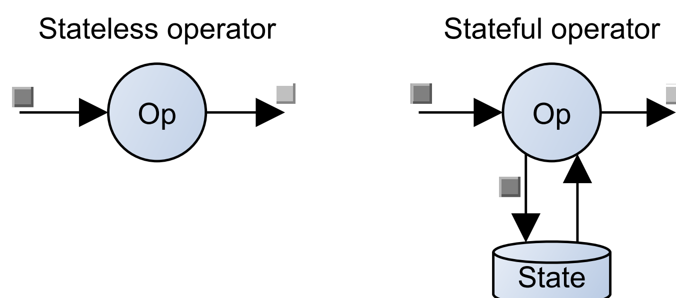


Figure 2.9: Stateless and stateful stream processing operators.

Stateful operators maintain a state based on historical information of items seen so far, which is more complicated to handle. Virtually all non-trivial event stream processing applications require stateful stream processing. In the example of the trending topics

application in Figure 2.7, the "Count" operator is stateful because it needs to maintain the partial sum of the different hashtags and update these values for each new item. The output stream of this operator is a tuple containing the hashtag and partial sum, which the sink will use to update the trending topics list. [AGT14] also discusses *partitioned stateful* operators, which is a particular case of stateful operators.

Operators can be defined from any stretch of the application's sequential code. No rule defines the complexity or size of an operator, it can execute a single line of code up to thousands. Ideally, an operator should perform a single, well-defined task, as this allows a stream processing runtime to better exploit a multi-tasking environment and combine or merge small operators if necessary [AGT14]. However, many applications do not allow you to split main computing into smaller, individual operators, plus source and sink. In these cases, it is important to combine other parallelism strategies, as we discuss in the next subsection.

2.2.4 Types of Parallelism for Stream Processing

In this section, we will discuss aspects of parallelism from the perspective of traditional stream processing, the domain we most focused on in this work. The parallelism in stream processing can be exploited simply by arranging operators in the application's flow graph, disregarding data dependency [AGT14]. These arrangements can define three main types of parallelism that can be applied to stream processing applications, as shown in Figure 2.10: a) pipeline parallelism; b) data parallelism; and c) task parallelism.

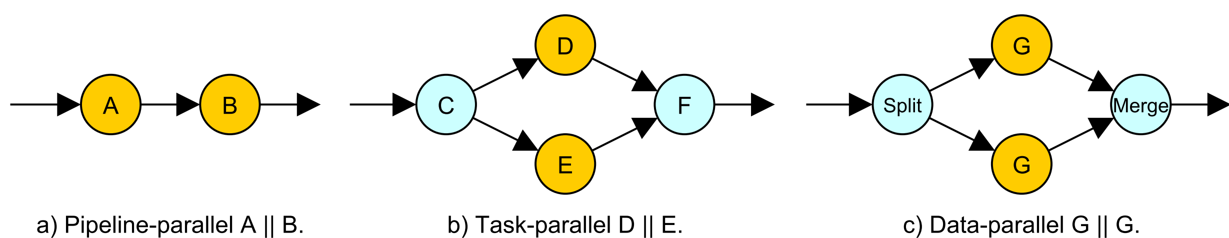


Figure 2.10: Pipeline, task, and data parallelism in stream graphs. Adapted from [HSS+14]

Pipeline Parallelism

For an application to explore pipeline parallelism (Figure 2.10.a), it must be able to express computing as a sequence of individual stages, so this parallelism occurs naturally in stream processing applications [AGT14]. Here operators can be scheduled to run on different processing units of a parallel architecture while maintaining the logical sequence

of operations of the flow graph [HSS+14]. This way, when an operator is computing an item, an earlier operator can process a subsequent upstream item in parallel.

However, just running such operators in parallel does not guarantee performance. For pipeline computing to be efficient it is essential to have a load balance so that different operators are not overloaded or idle [AGT14]. This balancing is achieved when all operators have a similar throughput. To perform this balancing, it may be necessary to merge idle operators in sequence or replicate overloaded operators to increase throughput. One sign of bottleneck and imbalance is the excess data queued at some operator's input port.

Data Parallelism

In data parallelism, the same computation is performed on different items concurrently. From a stream processing perspective, this is achieved through the replication of non-data dependent (stateless) operators (Figure 2.10.c). These operators can run in parallel and consume different items in a queue. Thus, multiple items are processed simultaneously by the replicas, and an operator in the sequence merges these items into the output stream. Therefore, to apply this kind of parallelism it is necessary to have a better understanding of the application because, besides problems with data dependency, it may be necessary to ensure the order of the items in the stream [AGT14].

The data parallelism, therefore, when used within the pipeline, allows implementing the Task-Farm, or simply Farm, which is the most common parallel pattern in stream processing applications [HSS+14]. This pattern comprises at least one operator, usually called "Emitter", which distributes the data items to subsequent operators (workers), which are replicated *n* times. In shared-memory systems, it is common for operators to have item queues to process. Therefore, Emitter can distribute items in different ways depending on the characteristic of the queue. If workers share the same queue, Emitter pushes items into that queue, and workers pull those items on demand [DMM17]. However, each worker can have their own queue, in which case the Emitter needs to use some scheduling policy to distribute the data. Also, some PPIs, such as FastFLow, can use unbounded lock-free queues in each stage for efficient parallelism [ADKT17a]. There is still a third operator, usually called "Collector", that gathers the result of the workers within the output stream.

However, data parallelism occurs in different ways in stateful and stateless operators. On the one hand, stateless operators can be replicated freely and consume any data item from the incoming stream. On the other hand, stateful operators are more complicated to replicate because, in a multithreaded context, data races can occur during concurrent reads and writes. In such cases, synchronization directives must be used. Some PPIs, such as TBB, support data parallelism with stateful operators but require the programmer to manually implement synchronization control.

Task Parallelism

In Stream Parallelism, task parallelism is a form of parallelism where independent processing operators are executed concurrently on the same or distinct data item (Figure 2.10.b) [SHGW15, AGT14]. A practical example is a multi-core processor, which can execute different tasks from different threads concurrently. Therefore, this kind of parallelism also occurs in a natural way, just like the pipeline, in a flow graph. The difference is that here the operators do not need to execute all the data items in a single sequential arrangement. The problem with this is that to ensure good load balancing, parallel operators need to have a similar processing time. However, traditional stream processing applications hardly have operators with these characteristics, so pipeline and data parallelism are the two predominant models.

2.2.5 Stream Processing on Parallel Architectures

As discussed in the last section, parallelism in stream processing can be explored by running the operators of an application concurrently in parallel. There are different ways to implement these strategies in parallel architectures, which depend on how the application graph is structured. Fundamentally, it consists of running different operators in different processing units [AGT14]. However, parallel architectures can take many forms, such as multi-core shared-memory machines, clusters, and cloud [RM19].

Multi-core Shared-Memory Stream Processing

Parallel architectures can take many forms, such as multi-core shared-memory machines, clusters, and cloud [RM19]. Stream processing systems running on a multi-core machine have scalability limited by machine size. To replicate operators and run in parallel, more threads are added until the architecture limit is reached. Nevertheless, this scalability limit for stream processing in multi-core systems is no longer a major concern.

Nowadays, most streaming workloads can be efficiently handled in a single multi-core machine without having to distribute it over a cluster of machines [PRR19]. Modern multi-core systems are already being developed with hundreds of cores [Mar20]. The parallelism offered by a single machine today is often sufficient to deploy pipelines with a large number of operators [PRR19]. The advantage of using these systems is that communication through shared memory is much faster than sending messages over the network, as in a distributed system. Therefore, real-time processing applications can achieve very low latency, highly improving performance [ZMK+19]. In addition, multi-core systems can also combine with accelerators, such as GPUs, to form heterogeneous architectures, significantly increasing scalability potential [RM19]. In a CPU+GPU architecture, operators

with high data parallelism potential can run on the GPU while the CPU manages sources, sinks, and other non-data-parallel stages, for instance.

Besides the existing advantages of doing stream processing in a shared-memory environment, there are new technologies that further increase these advantages. There are many stream systems focused on processing with absolutely no data storage [AHN+20]. These systems need to rely on the capacity of the memory to store the workload temporarily during computing. In distributed systems, this capacity is increased because each node has its independent memory system, which is not the case with multi-core systems. However, this scenario has changed rapidly. In addition to the increasingly low cost of memory, there are now specialized systems for fully in-memory multi-core stream processing for large workloads [Haz20].

In-memory processing has become an increasing trend within stream processing, especially in systems that need to process data stored in large databases (as opposed to systems that process data that arrives in real-time) [AHN+20]. These databases, if stored in conventional storage systems, could present a huge I/O performance bottleneck during processing. Conventional solutions are no longer able to efficiently handle the large volume of data generated by IoT devices, for instance [AHN+20]. Thus, multi-core systems are increasingly adopting persistent memory technologies, such as Intel® Optane™ [Int20], to meet these demands.

Besides general-purpose computing, there are also embedded multi-core processors, which can open an opportunity for the stream processing sub-domains to intersect with the emergence of Edge Computing. On the one hand are applications that would need to do stream processing in the cloud and that are limited by bandwidth and high latency. This was a challenge in implementing applications for virtual and augmented reality, and self-driven cars, for instance. These applications are needed to process data and make decisions in near real-time. However, on the other hand, are multi-core processors, which are on most user devices and embedded systems today. So these systems could do on-premise stream processing and send the non-critical parts to be processed in the cloud.

Distributed Stream Processing

Distributed systems can take many forms, such as clusters, cloud, fog computing, etc. These systems have different limitations. Clusters, for instance, have a fixed amount of computing nodes. Therefore, the scalability of stream processing applications in these systems is determined by the number of available nodes and processing units within each node [RM19]. These nodes in current clusters have multi-core processors. Thus, if a small cluster is assembled using the latest high-end multi-core processor technologies, as discussed in the subsection above, it can achieve a high scalability power.

Another type of distributed system is cloud computing. Most event processing applications run on this type of shared-nothing systems [ZMK+19]. Each node runs different operators, and, as in clusters, communication among them is done through message passing. These systems have much fewer limitations in terms of scalability than previous architectures because here, applications can simply allocate more computing nodes "unlimitedly" [RM19].

Another advantage of stream processing in decentralized distributed environments, such as cloud systems, is the fault tolerance they enable. Stream processing frameworks, such as Apache Kafka or Apache Flink, often incorporate features such as data replication, automatic data partitioning, and distributed processing, which allow for fault tolerance in the face of hardware failures, network issues, or software glitches. When a failure occurs, the stream processing system can automatically re-route data streams to healthy processing nodes, ensuring continuous data processing and minimizing downtime. Additionally, stream processing frameworks often provide robust error-handling mechanisms, such as event time-based windowing or time-based triggers, that enable the processing of out-of-order data or late-arriving events, thereby improving fault tolerance in handling data anomalies or delays. Fault tolerance in stream processing enables distributed computing systems to operate reliably and continuously, ensuring high availability and consistent results, even in the presence of failures.

But this type of cloud computing has its drawbacks as well. These systems have almost unlimited scalability but at the expense of performance. Stream processing applications that run on these systems have a higher processing latency because the communication between the data sources and the operators introduces a higher time delay than on the other architectures, being a critical factor for real-time processing applications [RM19]. Recent studies show that the platforms used to implement stream processing systems in this type of architecture, such as Apache Flink, Spark, and Storm, can perform better in multi-core systems [ZMK+19]. On the other hand, multi-core parallelism within each node can also be explored, considering stateless operators. In addition, these systems can also provide hardware accelerators, which can greatly increase performance gains [RM19].

2.2.6 Parallel Programming Interfaces for Stream Processing

The largest portion of the stream processing domain is represented by data stream processing applications, which are implemented in DSPS PPIs, as discussed earlier in Section 2.2.2. These DSPSs generally apply a distributed parallel producer-consumer model [ZMK+19, RM19]. The most common are Apache Flink [FT16], Apache Storm [Jai17], and Apache Spark [Nab16]). Most current DSPSs are implemented on top of clusters

or clouds. These systems generally rely on Java Virtual Machine (JVM) to abstract the underlying hardware and streamline the development process of these commercial systems. However, these systems cannot exploit high-speed networks or emerging hardware trends regarding multi-core systems [ZMK+19]. Although the JVM can abstract hardware for developers, it cannot easily provide efficient data access due to processing overheads induced by data serialization (and deserialization), objects scattering in main memory, virtual functions, and garbage collection [ZMK+19].

[MTG+19] says that these DSPS platforms, despite being able to achieve performance using powerful scale-up servers equipped with tens of colors and terabytes of memory, still need a lot of advancement in terms of high-level abstractions. Other authors have shown that the most common DSPSs in the industry are not able to extract the full potential of modern multi-core architectures [ZMK+19]. Based on these two problems, [MTC+21] proposed a high-level C++ PPI, called WindFlow, for data stream processing targeting multi-core and heterogeneous systems.

In addition to this growing trend of using multi-core systems for data stream processing, there are many PPIs specifically built targeting this architecture and which can support stream parallelism. Some of these PPIs are: TBB [VAR19], FastFlow [ADKT17b], SPar [Gri16a], OpenMP [DM98], ISO C++ Threads [Fou20], and GrPPI [dRADFG17]. Most of them are based on C++ and address programming abstractions at different levels for programmers, usually implementing common parallel programming patterns such as pipelining, task, and data parallelism [MRR12].

These PPIs offer several ways to explore parallelism, some of them primarily target stream processing, like FastFlow, SPar, and WindFlow. Others, such as TBB and GrPPI, are multi-paradigms, encompassing the stream processing domain by offering structured parallel patterns in this respect. It is also possible to explore stream parallelism using OpenMP and ISO C++ threads, but it requires a significant programming effort to implement synchronization, communication, and parallelism mechanisms in such PPIs. In this doctoral thesis, we implement parallelism and perform experiments with TBB, FastFlow, OpenMP, ISO C++ threads, SPar, GrPPI, and WindFlow.

Intel Threading Building Blocks (TBB) [VAR19] is a C++ template-based library for parallel programming from the industry. It provides a task-based parallelism model that can use multiple CPU cores to execute tasks in parallel. Stream processing with TBB can be explored using a pipeline of stages, each of which can be executed in parallel. TBB automatically distributes the work among the available cores to maximize performance. TBB also provides support for dynamic load balancing, which ensures that tasks are evenly distributed among the available cores, even if the workload changes dynamically during execution. This can help avoid situations where some cores are idle while others are overloaded.

FastFlow [ADKT17a] is a programming library implemented in modern C++ targeting multi/many-cores and, more recently, distributed systems [TTMD22]. It offers programmers a set of high-level ready-to-use parallel patterns and a set of mechanisms and composable components (called building blocks) to support dataflow streaming networks. In contrast to TBB, in FastFlow, tasks are statically executed by dedicated threads. While it allows for more flexibility in building custom graphs, it delegates most of the load-balancing task to the users. The communication among operators is done through Single-Producer Single-Consumer unidirectional and asynchronous lock-free FIFO queues carrying memory pointers.

OpenMP [DM98] is a popular programming model that enables parallelism in shared-memory environments. Its approach is unstructured, meaning it does not require a strict code organization or specific programming paradigm. Instead, OpenMP provides pre-compiler directives and library functions that allow developers to specify which parts of the code should be executed in parallel. This flexibility makes OpenMP a powerful tool for optimizing the performance of computational applications across a wide range of hardware platforms. However, for stream processing applications, which involve a continuous flow of data, OpenMP must be used with additional mechanisms to ensure that the data is processed correctly and efficiently. These mechanisms can be shared with the ISO C++ Threads stream parallelism. However, Threads require the programmer to define every aspect of the parallelism algorithm using a set of functions that interact with the operational system.

SPar [Gri16a] is a Domain-Specific Language (DSL) to express stream parallelism. This PPI allows programmers to use C++ attributes to implement parallelism. Its purpose is to prevent the programmer from rewriting the original application code. Therefore, it offers high-level abstractions inserted through annotations in the sequential code and uses its own compiler. This compiler parses the annotations and then generates FastFlow parallel code. Although experimental versions of SPar can also generate TBB and OpenMP parallel code, these versions are not publicly available.

GrPPI [dRADFG17] is an open-source generic and reusable parallel pattern programming interface. It accommodates a layer between developers and existing PPIs targeted to multi-core processors. It proposes to act as a switch between different PPIs, providing 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 [GdRA+20]. In its latest release, GrPPI allows running applications with four backends: ISO C++ threads, FastFlow, OpenMP, and Intel TBB.

WindFlow [MTC+21] is a C++17 header-only library for parallel data stream processing targeting heterogeneous shared-memory architectures. The library provides data

stream processing operators like map, flatmap, filter, fold/reduce, as well as sliding-window operators. Compared to traditional data stream JVM-based PPIs, WindFlow proposes to be a more effective alternative to exploit, at best, the potential of scale-up architectures such as single machines equipped with several multi-core CPUs and co-processors like GPUs and FPGAs.

2.3 Benchmarks

With the evolution of computer architectures, comparing the performance of different computing systems by looking at their specifications has become a problematic task [Gra92]. Historically, researchers used to rank performances using a variety of different metrics. This lack of standards was confusing, and often this information was unreliable. Gradually, comparative studies between systems converged and adopted standard benchmarks.

Benchmarks are standardized tests that are used to evaluate the performance of a system. They can be used to compare the performance of different hardware components, such as processors, graphics cards, and storage devices, as well as the performance of different software applications or configurations. By running benchmarks, it is possible to identify bottlenecks or weaknesses in a system and track improvements in performance over time. In addition, benchmarks can be used to assess the performance of a system under different workloads or conditions. Overall, the use of benchmarks is crucial for understanding and optimizing the performance of computers and computer systems. They are also helpful for researchers to evaluate and validate research results against state-of-the-art solutions.

2.3.1 Types of Benchmarks

[vKAH+15] define a benchmark as a "*Standard tool for the competitive evaluation and comparison of competing systems or components according to specific characteristics, such as performance, dependability, or security*". According to them, there are three major types of computer benchmarks: specification-based, kit-based, and a hybrid of the two. Specification-based benchmarks describe the desired functions, input parameters, and expected outcomes but leave the implementation up to the person running the benchmark. Kit-based benchmarks provide the implementation and do not allow for alterations to the execution path. Hybrid benchmarks combine elements of both specification- and kit-based benchmarks.

2.3.2 Properties of Benchmarks

[vKAH+15] analyzed several studies and benchmarks used in the industry and identified a group of desired benchmark characteristics.

Relevance

Relevance concerns how closely the behavior of a benchmark relates to the behavior of the scenarios of interest to users. Therefore, the relevance of a benchmark can be determined by its applicability and the degree of relevance within a specific area. Benchmarks designed to be highly relevant within an area naturally have narrow applicability. On the other hand, benchmarks designed for a broader spectrum tend to be less meaningful for any particular scenario [Hup09]. For example, a video processing benchmark for face recognition may be highly relevant for representing face recognition applications, somewhat relevant for representing video processing applications, and not at all relevant for representing fraud detection applications.

In addition to considering the application domain, the relevance can also be determined by how the user intends to use this benchmark. Suppose the goal is to evaluate HPC platforms and tools. In that case, the benchmark needs to be well scalable and allow multi-threaded or multi-process execution. It is not easy to achieve since HPC architectures significantly differ in available resources. Another factor that can determine the relevance of a benchmark is the type and quality of the metrics it provides. Metrics need to be transparent and representative for the expected scenarios of the benchmark's applicability.

Reproducibility

Reproducibility is the ability of a benchmark to produce consistent results in an equivalent execution environment. It concerns the results between consecutive runs and the ability of another user to achieve the same results in the same environment [vKAH+15]. Ideally, the result of a deterministic benchmark should be given only by the combination of hardware and software configuration. However, today's computers have a complexity that can introduce variability in the results. This variability can be introduced by conditions such as thread scheduling, physical disk layout, the interaction between background processes, and dynamic compilation [Hup09]. Benchmarks should be able to mitigate these variations in some way, such as running for long periods to dilute their impact. Another option is to run several times and see if the results are close and consistent.

The ability to reproduce tests in another test environment is also tied to the ability to build an equivalent environment [IT18]. It is often impossible because of a lack of descriptive details from previous runs. The hardware needs to be detailed so that another

user can reproduce it. So do the software versions and configurations, such as compiler, CPU governor policy, libraries, and operating system. Despite efforts to provide a sufficient description of the test environments, many scientists cannot even reproduce their own experiments after one year [DL15]. Therefore, benchmarks must look for ways to mitigate such problems, both with respect to variations during execution and the replication of the test environment.

Fairness

Fairness, or equity, ensures that the systems under test can be evaluated fairly without the benchmark artificially favoring one of them. Ideally, benchmarks should be built into an agreement between different developers and users. If multiple interested parties participate in the process of designing a benchmark, this helps to ensure greater fairness. A benchmark designed specifically to exploit a feature of a hardware component from one vendor but that does not address similar components from other vendors would not be a fair benchmark to compare those systems, for example. When evaluating software components, such as comparing the performance of different PPIs, fairness can be attested to by testing the components on multiple benchmarks and test beds.

Verifiability

Verifiability is the capability to verify that the result of a benchmark is correct. It is important that a benchmark allows itself to be verifiable so that users can trust its results. Often this verifiability is based only on comparing the user's results with previous results published by the benchmark's developers or other relevant academic papers. This practice is generally not ideal, as inconsistencies in the original results can potentially lead to more significant errors in derived works.

To mitigate such problems, good benchmarks should be able to perform some amount of self-validation [vKAH+15]. The goal is to ensure that the workload performs as expected and that all rules are followed. However, since benchmarks can take on many different configurations, complete verifiability can take much work to achieve. Verifiability features in benchmarks can be limited to specific configurations in such cases. Functional verification can also be adopted, which tests whether the benchmark's output is correct. One way to improve verifiability is to include more detail in the published results, using different settings and performance metrics. It gives users more room to find inconsistencies in their runs.

Usability

Benchmark users tend to have a sophisticated level of technical knowledge, making the usability of benchmarks a secondary factor to be considered by their developers [vKAH+15]. However, there are strong reasons that favor a greater concern with usability. Reproducibility and verifiability, for example, are properties tied to usability. Benchmarks that perform self-validation of results can prevent non-expert users from having to understand in-depth details of the workload and the result to confirm the correctness of the execution. Benchmarks that provide ways to facilitate their reproducibility also contribute to better usability. For example, a benchmark can provide its main library dependencies and an interface allowing users to make a standardized installation of them. It can also provide mechanisms to mitigate or highlight variations in the results. Good code and usage documentation, use cases, manuals, description of applications and workloads, and code organization are also key things regarding the usability of benchmarks.

2.3.3 Parallel Benchmarks

Parallel benchmarks are tests designed to evaluate the performance of parallel computing systems. These systems, which include parallel computers, clusters, and supercomputers, use multiple processors or computing nodes to perform tasks simultaneously in order to achieve faster performance than what is possible with a single processor.

There are many types of parallel benchmarks designed to test different aspects of parallel system performance. Some examples include benchmarks that test the speed of communication between processors, the efficiency of load-balancing algorithms, the scalability of applications, and the ability of a system to handle different types of workloads. In order to be effective, parallel benchmarks must be carefully designed to reflect the characteristics of real-world parallel computing applications' characteristics and isolate the factors being tested.

There are several reasons why parallel benchmarks are important in the field of parallel computing:

1. They provide a way to measure different parallel systems' performance and compare their relative strengths and weaknesses. It can be useful for choosing the best system for a particular application or workload.

2. Parallel benchmarks can help identify bottlenecks or weaknesses in a parallel system and guide efforts to optimize its performance.

3. Parallel benchmarks can be used to track the evolution of parallel computing technology and to measure the impact of new hardware, software, and programming models on performance.

Overall, parallel benchmarks are an essential tool for evaluating and improving the performance of parallel computing systems and comparing their relative strengths and weaknesses. Therefore, they play an important role in advancing the state of the art in parallel computing and enabling the development of more powerful and efficient systems.

There are many parallel benchmarks that are commonly used in the field of parallel computing to evaluate the performance of parallel systems. Most of them were developed to evaluate the hardware performance of computing systems. Some of the most well-known and widely used parallel benchmarks include:

- LINPACK [DLP03]: This benchmark measures the floating-point performance of a parallel system by solving a dense system of linear equations. It is commonly used to rank the world's most powerful supercomputers. It has been a standard benchmark for over four decades.

- HPL [Pet04]: The High-Performance Linpack (HPL) benchmark is an update to LINPACK that is specifically designed to test the performance of distributed-memory parallel systems, such as clusters and supercomputers. It measures the performance of a system using the same dense linear algebra algorithms as LINPACK but with additional features to support large-scale distributed-memory systems.

- STREAM [McC95]: This benchmark measures a parallel system's memory bandwidth and sustainable memory throughput. It uses a set of simple vector kernels to test the system's ability to transfer data between the processor and memory.

- NPB [BBB+91]: The NAS Parallel Benchmarks (NPB) are a suite of parallel benchmarks developed by the NASA Advanced Supercomputing Division to test the performance of supercomputers. The NPB suite consists of a range of scientific and engineering applications, and they include a range of problem sizes to test the scalability and different aspects of parallel system performance, including floating-point, communication, and I/O performance.

- PARSEC [BKSL08]: The PARSEC benchmark suite is a collection of parallel benchmarks that test the performance of a parallel system on a range of applications, including image processing, machine learning, and molecular dynamics. It is designed to reflect the characteristics of real-world parallel computing applications and to test the scalability of a system across a wide range of problem sizes.

Although these benchmarks target hardware performance evaluation, in 1971, it was already understood that benchmarks should not only be used to evaluate the perfor-

mance of architectures but also to test aspects of the software, such as the performance of multiprogramming and multiprocessing techniques [Luc71]. Therefore, benchmarks are used today to evaluate a diversity of aspects of both hardware and software.

In this work, we propose a framework to facilitate the creation of benchmarks to evaluate stream processing systems, PPIs, parallelism strategies, and related techniques for performance improvement. Other studies have already been concerned specifically with evaluating and comparing PPIs performance [GSG20a], but they have not entered the stream processing domain. This evaluation of PPIs is important to help developers and researchers to predict the performance of these tools under certain circumstances, such as parallelism technique, workload characteristics, hardware architecture, etc. Other benchmark suites that include stream processing benchmarks are either unrepresentative, challenging to use (e.g. by the lack of documentation), or target only specific categories of Stream Processing (SP) applications, such as data stream. Chapter 4 discusses more deeply related benchmark suites.

3. SPBENCH BENCHMARKING FRAMEWORK

In this chapter, we present the SPBench benchmarking framework. First, we discuss our motivations. Here we also review most of the research problems discussed in the Introduction chapter and how we address each of them. Then we discuss in detail all aspects of the conceptual framework and its architecture.

Contents

3.1 MOTIVATION **56**

3.2 SPBENCH FRAMEWORK **60**

3.2.1 FRAMEWORK API 64

3.2.2 SPBENCH SEQUENTIAL BENCHMARKS 67

3.2.3 SPBENCH PARALLEL BENCHMARKS 68

3.2.4 PERFORMANCE METRICS 70

3.2.5 BENCHMARK PARAMETERIZATION 72

3.2.6 COMMAND-LINE INTERFACE 72

3.3 RELATED WORK **77**

3.3.1 DISCUSSION 79

3.4 CHAPTER SUMMARY **82**

3.1 Motivation

In [Gri16a], the author proposed a DSL called SPar [GDTF17] for expressing high-level stream parallelism. The “high-level” term in that context is for parallelism abstractions that prevent the user from dealing with details concerning to parallel architecture optimizations, avoid code rewriting, and reduce the programming effort to support parallelism. SPar requires the programmer to only annotate the parallelism without having to rewrite the original code. Programmers identify the operators and their respective input and output data. There are stream processing applications with thousands of lines of code, where it can still be very difficult to identify the beginning and end of each operator and also identify all the data dependencies.

Listing 1 Mandelbrot function.

```
1  void mandelbrot(double init_a, double init_b, double range, long dim, long niter) {
2      double step = range/((double) dim);
3      unsigned char *M = new unsigned char[dim];
4      for(unsigned long i = 0; i < dim; i++) {
5          double im=init_b+(step*i);
6          for (unsigned long j = 0; j < dim; j++) {
7              double cr;
8              double a=cr=init_a+step*j;
9              double b=im;
10             unsigned long k = 0;
11             for (k = 0; k < niter; k++) {
12                 double a2=a*a;
13                 double b2=b*b;
14                 if ((a2+b2)>4.0) break;
15                 b=2*a*b+im;
16                 a=a2-b2+cr;
17             }
18             M[j] = (unsigned char)(255-((k*255/niter)));
19         }
20         ShowLine(M,dim,i);
21     }
22     delete[] M;
23 }
```

The difficulty of applying stream parallelism can appear even in shorter code. Mandelbrot, for example, is a classic application from other domains. However, it is often used as a benchmark for stream processing due to the lack of appropriate and representative benchmarks. A quick look at the kernel of this application, presented in Listing 1, may be enough to show how difficult it can be to identify what can be split into different pipeline stages, which stages can be replicated, and what data dependencies are involved. Therefore, exploring stream parallelism can be difficult, even using high-level abstractions and PPIs that provide structured parallel patterns. It is a complex task usually reserved for specialists.

Most PPIs for stream processing, such as FastFLow and TBB, provide parallel pattern abstractions for programmers, as discussed in Section 2.2.6. Besides the complexity of stream parallelism, abstraction strategies are still a way to bring more application developers closer to the stream processing domain. Our work contributes to this aspect and tries to add another layer of abstraction for stream parallelism. However, instead of developing abstractions for parallelism extraction, here we target application abstractions, considering the whole context of it, which goes from the definition of operators, data flow, and workload settings. We also target the surrounding environment of a stream parallel application, facilitating its installation, compilation, configuration, execution, and evaluation. The idea is to make it easier for users to parallelize and execute stream processing applications using our framework with some PPI.

Only applying basic stream parallelism is often not enough to meet performance requirements. There are different features and parallelism strategies that may be fine-tuned and customized to achieve better performance or improved efficiency of a parallel application. Not only concerning the parallel code implementation but also regarding how these PPIs and applications interact with specific architectures. We can mention the organization of operators, parallelism degree, task scheduling and queues access policies between operators, thread-to-core affinity, etc. Therefore, we argue that it is important to have tools that aim to assist programmers in this type of evaluation and comparison between PPIs, parallelism strategies, and technologies for stream processing. It can be helpful for researchers who develop these PPIs and programmers who seek fine-tuning parallelism to achieve maximum performance in a given architecture.

Choosing the most appropriate PPIs, settings, and parameters for specific scenarios can be challenging. While good PPIs should ideally balance the properties discussed in Section 2.1.3, it is common for some properties to be prioritized more than others. To provide a simple interface for better programmability, a PPI may forego fine-tuning, for instance. Programmers in industry and academia may spend significant time evaluating such properties when choosing a PPI for a given application. It is important to have a suite that provides a collection of benchmarks implemented in various PPIs for stream processing and a tool that allows easy and fast prototyping of benchmarks in this context.

We discussed the complexity of exploring stream parallelism from the perspective of a non-expert application programmer, where abstractions from PPIs can be not enough, given the complexity of SP applications. Then, we discussed it from the perspective of system programmers and researchers who develop solutions for this domain and have limited tools for evaluating their solutions. Our work aims to help these two types of programmers and go beyond a framework to ease parallelism exploration or a benchmark suite. One of our goals is to enable programmers to create their own benchmarks for stream parallelism. Through the framework API, programmers can easily create custom parallel benchmarks using different PPIs and parallelism approaches. These benchmarks

automatically include workload configuration mechanisms and representative performance metrics.

Therefore, the SPBench framework offers two main benefits for programmers and researchers: first, it adds abstraction layers in stream processing applications to make it easier for application developers to use parallel programming interfaces to explore parallelism in this domain; second, it gives the opportunity for researchers and developers to use real-world C++ stream processing applications to compare and evaluate their solutions easily.

The SPBench is presented in Chapters 3 and 4 of this thesis. We have separated it into two chapters to improve the readability and organization of the text. These chapters address the Research Problem 1, 2, 3, and 4 that were introduced in Chapter 1. Here, we review these research problems and discuss how this paper addresses each of them. Below we list each of them, followed by a summary of our solutions:

1. **It is not easy to explore parallelism in stream processing applications.**

   It is necessary to have a good understanding of the application to be able to apply stream parallelism. The programmer needs to recognize all operators individually, identify what the data dependencies are between them, understand what data each operator consumes and produces, etc. Along with this, achieving performance in these applications may require using combinations of parallelism strategies and specific configurations to achieve good load balancing.

Listing 2 Example of Mandelbrot function if implemented with the SPBench framework API.

```
1   void mandelbrot() {
2       while(1) {
3           spb::Item item;
4           if(!spb::Source::op(item)) break;
5           spb::Mandelbrot::op(item);
6           spb::Sink::op(item);
7       }
8   }
```

   To assist programmers in these aspects, our framework provides a set of applications implemented with an API (Section 3.2.1) that delivers individual operators as simple function calls. The API can eliminate the problem of manipulating the operators' data consumed and produced by encapsulating it in a generic item. These optimizations can help programmers quickly understand the application design and allow different parallelism strategies to be easily restructured. For instance, if Mandelbrot were an application included in SPBench, the code from Listing 1 would look like the code from Listing 2 after we rebuild it using the SPBench framework API. The resulting code becomes more readable and the main elements are evidenced.

2. **There are no representative benchmark suites for traditional stream processing.**

   Most benchmark suites for stream processing include only data stream applications with frameworks targeting distributed platforms. Benchmark suites that include traditional stream processing applications are outdated or are limited in terms of programming language, parallelism exploration, and usability. In this thesis, we gather a set of applications for traditional stream processing and through SPBench, we make available the sequential versions of them so that they can be parallelized with different PPIs. Then, we use these applications to build a parallel benchmark suite for stream processing in C++, including traditional stream processing benchmarks. Although we target multi-cores within the scope of this thesis, the SPBench framework concept may be extended to other architectures.

3. **Benchmarks for C++ stream processing are limited.**

   Even considering all stream processing paradigms, there are still no widespread C++ benchmark solutions. In Section 1.1 and Section 2.2, we address initiatives toward the exploration of data stream processing in multi-core architectures using C++ [BAJ+16, MTG+19, dDFG18, LHP+22]. However, to evaluate these technologies, researchers usually need to convert benchmarks initially implemented in JVM-based programming languages to C++. Although some of these researchers make their C++ benchmarks available, they often lack proper documentation or parameterization. Also, only parallel versions of the benchmarks are provided. The absence of sequential versions can make it more challenging to port it to other PPIs. This highlights the need for benchmarks that meet these demands. SPBench is extensible and modular, being able to incorporate applications from other SP paradigms. Therefore, besides a set of C++ applications for traditional stream processing, it also includes data stream processing applications.

4. **There is a lack of solutions targeting to ease the evaluation process of stream processing.**

   This research problem involves the previous three we described and goes beyond. The problems we have exposed so far have concerned programmers and researchers who want to evaluate and explore parallelism in stream applications. Therefore, our proposed solutions have involved providing benchmarks and abstractions to make it easier. However, the great contribution of this work benefits a more specialized audience, which is the programmers that actually develop the PPIs and other technologies for stream processing. As we discussed in Section 1.1, these programmers face many challenges in developing new features, optimizing PPIs by adding support for architectures, languages, programming models, parallel patterns, and other functionalities, such as self-adaptive parallelism techniques and automatic code generation,

etc. These programmers must test their work with different parameters, types of applications, and evaluate different metrics to validate their solutions.

One of the goals of this work is to offer an easy solution for programmers who need realistic customized benchmarks to evaluate different technologies for stream processing in C++. Here, our framework can help programmers to focus specifically on the technology under evaluation and eliminate concerns regarding the application code, such as operator definition and input/output data. Besides the abstractions provided within each application, the framework's API is expected to use the same implementation standard across all applications, keeping a similar base structure (see Section 3.2.1). So once the solution is implemented in one application, it can be easily extended to the whole set. Therefore, the solutions under test can be fully portable among applications.

Besides applications and code abstractions, the framework offers different workload configurations. Here the programmer can define different workload classes, adjust batch size settings, choose data source and sink from disk, memory, or network (in the future), and also adjust the item arrival rate (frequency) in the stream (Section 3.2.1). Implementing performance metrics in stream processing benchmarks is also difficult and error-prone. This complexity increases when trying to simulate scenarios of unbounded streams, which require sampling measurements. The SPBench automatically provides the most commonly used performance metrics for stream processing in literature, such as latency, throughput, and CPU and memory usage (Section 3.2.4). These metrics and other parameters can be adjusted through the framework's command-line interface or source-code routines on-the-fly (Section 3.2.6). This way, users can easily and rapidly prototype customized benchmarks, test them with specific configurations, and gather different metrics.

3.2 SPBench Framework

In this section, we introduce the SPBench framework. The main goal of SPBench is to enable users to easily prototype custom benchmarks from real-world stream processing applications and evaluate multiple parallel programming interfaces. As discussed in Section 3.1, applying stream parallelism to applications in this domain can be very challenging. Commonly users are faced with applications like the example in Listing 1, where operator boundaries and data dependencies are difficult to recognize. One of the primary purposes of the SPBench framework is to extract the difficult and laborious task of dealing with the application from the users' side and leave them to deal only with parallelism.

Figure 3.1 illustrates what was discussed above. In the SPBench way, applications are recoded beforehand to fit the SPBench API, which we will discuss in more detail later. It

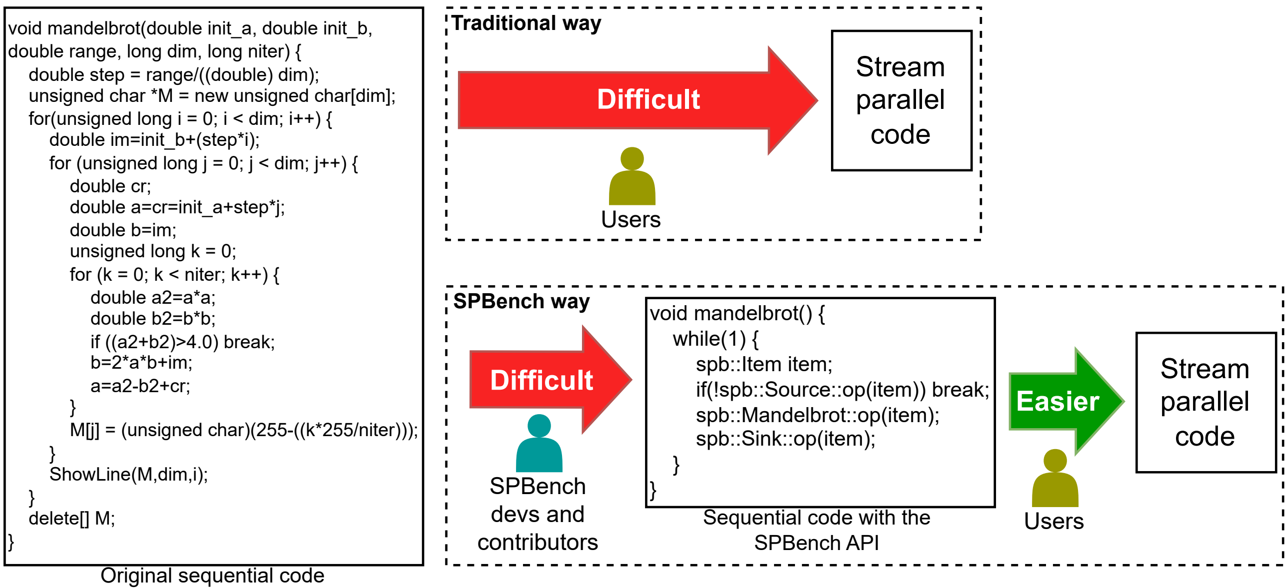


Figure 3.1: Users' perspective when writing stream parallel code: traditional vs. SPBench way.

is important to note that the difficulty extracted from the user side does not disappear. Part of it is transferred to the developers and other contributors who have added the application to the SPBench. They must encapsulate and adapt the original sequential code to fit in a specific structure. This structure put in evidence only the more relevant stream elements. Therefore, it becomes easier to work with rather than implementing parallelism from the original code, mainly in more complex applications. Of course, even if starting from the SPBench implementation of the sequential code, it can still be challenging for users to implement stream parallelism, especially beginners in this area.

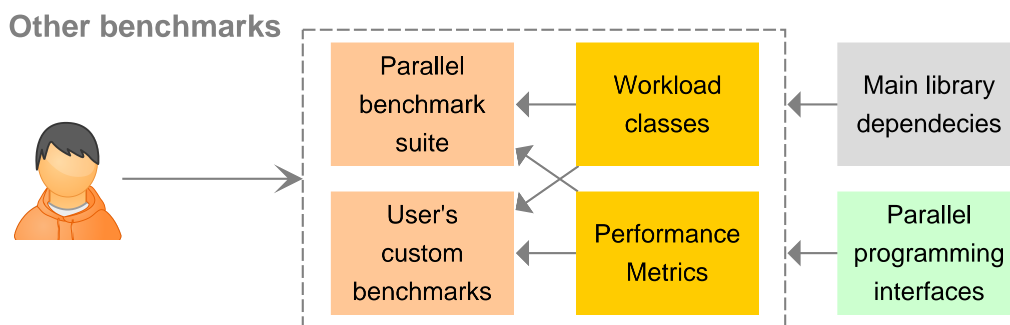


Figure 3.2: SPBench architecture.

Besides the sequential source code simplification, another of the main features of SPBench is the way users interact with the benchmarks. Figure 3.2 illustrates how this interaction usually occurs in other benchmarks/benchmark suites. In these cases, the interaction is usually direct and manual. A common interaction is: users select a benchmark, compile, and run it with a given workload, gathering some performance metric. However, it usually is not a smooth process. Benchmarks normally require different library

dependencies and leave the users with the task of finding, configuring, building, and linking them for compiling the benchmark. Sometimes they require specific library versions or configurations and add poor or no documentation about it. Thus, setting up and compiling a benchmark can by itself be a tough task. In addition, benchmarks often require input workloads that were not made available within it, and not rarely are they difficult to find. Regarding performance metrics, many benchmarks provide only basic metrics or none.

In stream processing, parameterization is also an essential factor. Achieving desired performance levels often relies on fine-tuning the application and workload settings. For more sensitive analysis, the option to turn specific performance metrics on or off can be used to avoid unwanted overhead. Related benchmarks often fall short in these respects.

In parallel benchmark suites, to add a new parallel implementation of a given benchmark, users need to do the integration manually as well. Also, most parallel benchmark suites do not even provide a sequential version of the parallel benchmarks for users to start a fresh implementation. In many cases, it can be more difficult to implement a parallel application starting from another parallel implementation due to the complex parallelism syntax that there may be. Therefore, one important advantage of SPBench is that it provides a sequential implementation of all benchmarks. In fact, providing highly readable, parameterizable, and customizable sequential benchmarks is one of the key features of SPBench. Adds to it a user interface and a management system to ease benchmark usability.

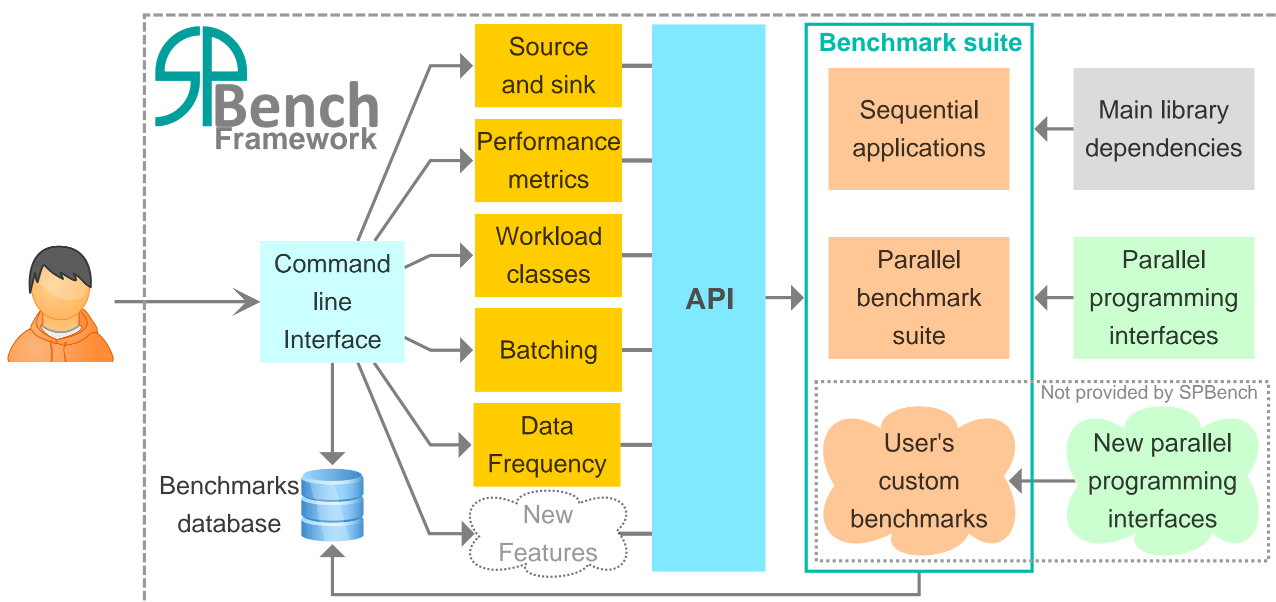


Figure 3.3: SPBench framework.

Figure 3.3 presents the SPBench Framework. In contrast with the other benchmarks, in SPBench, the user interaction occurs through a command-line interface (CLI). SPBench maintains a database containing all benchmarks added to its suite. With the CLI, users can access these benchmarks, modify them, and add new custom benchmarks

based on the supported applications. The CLI allows for easily integrating and configuring new benchmarks. Other secondary parameters can also be tuned via the CLI with simple commands, allowing users to select different data sources, workloads, performance metrics, and other settings. It also allows for automatically downloading and installing the main library dependencies and input workloads of the benchmarks. Therefore, it facilitates the installation, compilation, and execution process of the benchmarks.

The kernel of SPBench framework is an API that offers three main advantages to users:

1. It allows the implementation of stream processing applications in a modular, reconfigurable, and standardized way;

2. It automatically and transparently adds all the main benchmarking metrics used in this domain, enabling performance evaluation in many depth levels;

3. It offers a sort of workload customization, such as data input rate, batch size, different and multiple data sources, etc.

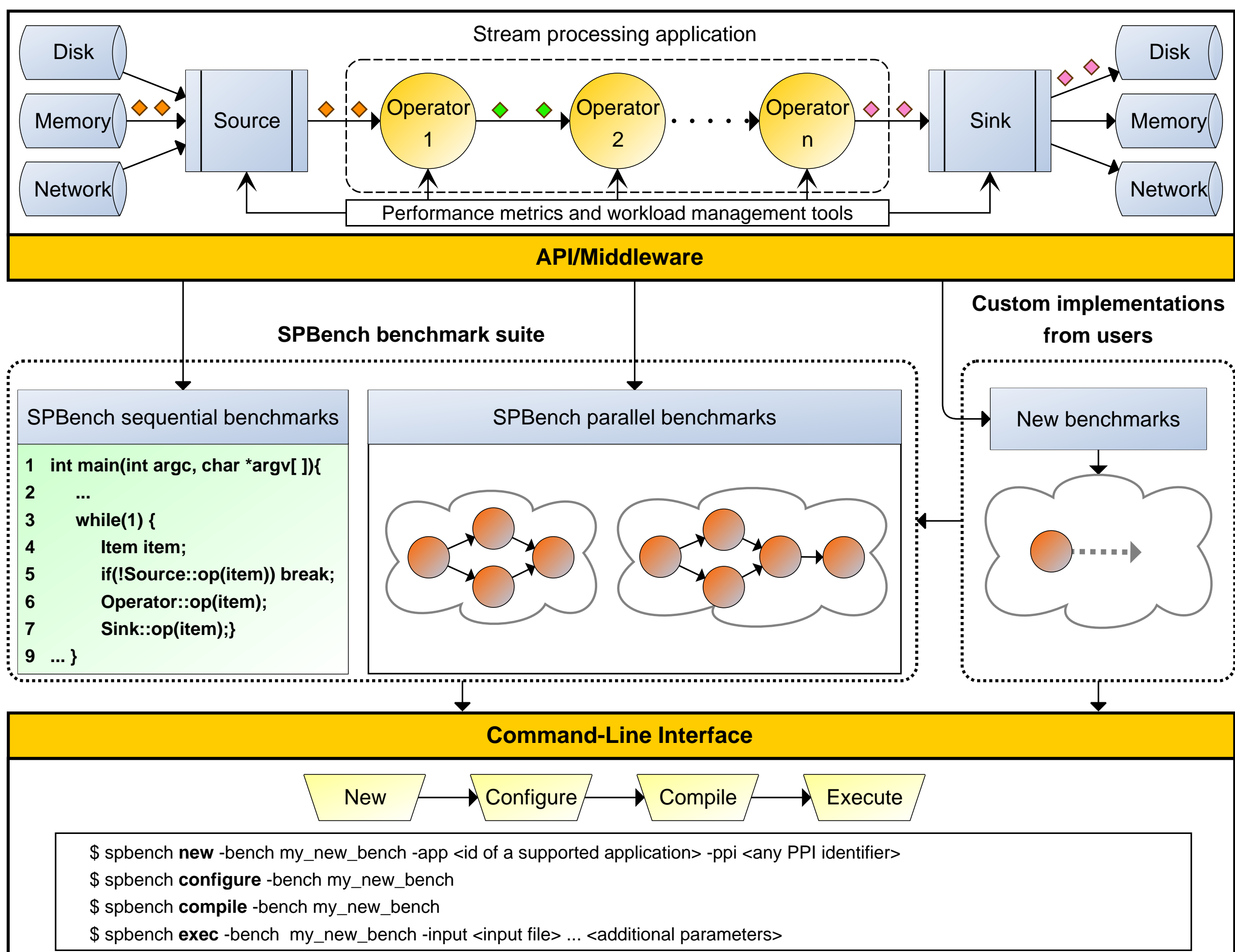


Figure 3.4: SPBench architecture.

Figure 3.4 presents a high-level representation of the SPBench architecture. It consists of three main parts: the API (at the top of the figure), the set of benchmarks implemented using the API (in the middle), and the command-line interface (at the bottom). In the next sections, we describe each of these parts in detail.

3.2.1 Framework API

The main functionality of the SPBench framework is an API that allows the implementation of applications in a modular, reconfigurable, and standardized way. Benchmark implementations using this API present the core of each application in a standard way and with few lines of code (such as the example of Listing 2). The original source code of such applications can be thousands of lines long. However, in SPBench, the low-level details of the applications are abstracted away and become transparent to the users. Therefore, SPBench allows users to focus entirely on writing and tuning the parallelism rather than spending time with potentially non-relevant aspects of each application.

To build this API, we have to disassemble all operators from the original application and restructure them to fit the API standards. Benchmarks can access the operators by including the header file of an application supported by SPBench. Besides the operator abstractions, the API also needs to encapsulate the data communicated among the operators, the parser for command-line arguments, the initialization routines for the application or the benchmarking metrics, the workload management routines, the metrics computation, and the final routines for memory management, output writing, or some other application requirement. All these elements are transparent for users in all SPBench benchmarks.

Operators

In the SPBench API, all operators of an application are encapsulated individually, from source to sink. For this, we first need to optimize and restructure the application code so that the different operators are highlighted and data dependencies are identified. Then these operators are separated into individual C++ classes that implement a static method containing the sequential code of the respective operator. These methods also implement time measurements for performance metrics. The Listing 3 shows what can look like the basic structure of an operator.

Within the benchmark, on the user side, this operator can be executed by calling `Operator::op()` and passing a data item object as a parameter. All applications have at least one source and one sink operator (`Source::op()` and `Sink::op()`). Therefore, in the sequential SPBench benchmarks, the intermediate operators are usually the only part that varies when changing the base application. The time measurements inside

Listing 3 Operator Class inside the API.

```
class Operator{
public:
    static void op(spb::Item &item);
    Operator(spb::Item &item){ op(item); }
    Operator(){}
    virtual ~Operator(){}
};

void Operator::op(spb::Item &item){
    // Time measurements
    ...
    // Batching mechanisms
    ...
    // Operator's sequential code
    ...
    // Time measurements
}
```

the `Operator::op()` consist of timestamps attributed to the data items if latency-related metrics are enabled. Each item is, in practice, a batch, so batching mechanisms are required to process each item individually when batching is enabled. We discuss more about batching in SPBench in Chapter 7.

Data Items

We encapsulate the data dependencies into a class called `Item`. Each application implements its own `Item` class, which implies that the item structures vary across applications. In the SPBench benchmarks, these items can be passed as an argument to the operators, such as `Operator::op(item)`.

The Listing 4 shows a basic item structure example. The data on these items is optimized to avoid oversizing since items carrying unnecessary data can increase memory consumption. In an application with more than one execution mode, such as data compression or decompression, these items are optimized to support both modes. This example shows an item class from an application that supports batching, which is implemented with vectors.

In some applications, the data item does not necessarily traverse all operators and can be dropped midway. An example is a fraud detection application, which discards the non-fraudulent data and keeps the potential frauds. This type of application has a filter operator that can drop an item if a given condition is true or false. In such cases, the `Item` class also defines an operator overload to allow it to be used as a boolean type.

Listing 4 Structure of a generic data item inside the API.

```

struct item_data {
    int integerData; // some data
    char *charData;  // some data
    /* ... */
    item_data(){}
};

class Item {
public:
    std::vector<item_data> item_batch; // batches are implemented with vectors

    Item(){}; // item constructor
    ~Item(){} // item destructor
};

```

Source and Sink

The source operator receives the task of initializing the data item with the data received from external sources. The sink works similarly but in reverse. It receives a processed item and generates an output result. The source also implements a frequency manager that allows for generating items at a specific rate. Users can also instantiate multiple sources. Source operators return `false` when there is no more data to process. The SPBench framework ideally should support three alternatives for source or sink: disk, memory, and network. Its current version supports disk and memory options.

Reading from disk is the simplest option to implement. The application can receive a pointer to a file stored on the disk and the source operator reads micro-batches of this file, which can be a video frame, a set of frames, a block of bytes, etc. In other scenarios, the data can be artificially generated in the source itself, not requiring an external file. The items are processed and then received by the sink, which builds the resulting file on disk if desired.

The “memory” option for the source consists basically of an in-memory execution. As we discussed in Section 2.2.5, specialized in-memory stream processing architectures are emerging and our framework also needs to serve this audience. With the in-memory execution enabled, the input file is loaded into memory before being sent to the source operator. This operator then receives only a pointer to the data and makes the assignment of the memory blocks to each item. The item can simply contain a pointer to a video frame stored in a vector or a pointer + the size of the data in the memory in order to read a variable-sized block of bytes.

The third external source/sink option is “network”. It is still a conceptual idea for future work since we did not implement it in the current version of SPBench. This option is important to represent more realistic scenarios. For instance, a person-recognition

application could receive video frames from a monitoring camera over the network in real time. This is a more complex functionality to implement in the framework. It must work in an independent system to be more realistic, simulating real external sources. This system will need to run in a parallel thread alongside the application and use communication protocols. In addition, it needs to be a generic system to meet all data requirements of the applications and be easily reconfigurable.

3.2.2 SPBench Sequential Benchmarks

We previously discussed how the API implements the main elements of the benchmarks that are transparent to the user. Here, we describe how those parts can be used to build a benchmark/application (sequential version). This process represents the middle part of the SPBench architecture in Figure 3.4. The middle part is divided into three distinct sets of applications. On the left are the sequential applications, which represent the starting point of all SPBench benchmarks. We present each of the supported applications in Chapter 4. In the center are the parallel benchmark implementations, which we discuss in detail in Chapter 4. And on the right are the custom benchmark implementations added by users, which, of course, are not provided by SPBench, as illustrated in Figure 3.3.

Listing 5 Structure of a benchmark built with the framework API.

```
int main (int argc, char* argv[]){
    spb::init_bench(argc, argv);    //Initialization procedures
    spb::Metrics::init();           //Global metrics initialization
    /*beginning of the stream region*/
    while(1){
        spb::Item item;
        if (!spb::Source::op(item)) break;   //Take an item from the source
        spb::Operator1::op(item);   //First internal operator
        spb::Operator2::op(item);   //Second internal operator
        ...
        spb::OperatorN::op(item);   //Last internal operator
        spb::Sink::op(item);        //Write the result
    }
    /*end of the stream region*/
    spb::Metrics::stop();           //Performance computation
    spb::end_bench();               //Memory deallocation and other final procedures
    return 0;
}
```

The sequential code example in the figure was presented in a simpler way to facilitate understanding. In practice, some more elements make up the benchmarks. Listing 5 presents an example of a complete implementation of a sequential benchmark using the API. All sequential benchmarks of the framework have a similar structure. Usually, the only part that changes in the benchmarks when changing the base application is the

internal operators, which in this example of the Listing are in lines 8 to 11. Therefore, the number of internal operators and their names vary from benchmark to benchmark.

Such standardization of code structure seeks to facilitate the understanding of the benchmark applications. Once the structure of a single benchmark is understood by users, they automatically understand the structure of all of them. Thus, keeping a similar structure across the sequential benchmarks is important to increase parallel code portability, as we discuss in Section 3.2.3. It is important to highlight that the kernel of SPBench is these sequential benchmark applications. All the SPBench parallel benchmarks were developed on top of these sequential benchmarks. Therefore, SPBench has nothing to do with parallelism itself. Its goal is to provide application and benchmarking abstractions to make it easier for users to use some parallel programming interface to implement stream parallelism and create benchmarks from it. In the next section, we discuss how users can do this.

3.2.3 SPBench Parallel Benchmarks

Besides the sequential benchmark applications, SPBench provides parallel benchmarks using several PPIs. Users can use these benchmarks to evaluate the PPIs and adapt them to evaluate other technologies or different parameters as their need. Also, inexperienced users can take these benchmarks to understand better how stream parallelism works. We added the parallel benchmarks the same way any user would do. The only difference is that we are making these benchmarks available within SPBench. They are also self-contained, i.e. the SPBench provides all the necessary library dependencies for them. Thus, what we discuss in this section applies to the “SPBench parallel benchmarks” and “New benchmarks” parts from Figure 3.4.

Listing 6 presents an example of a parallel benchmark application. The application in this example has three intermediate operators plus source and sink. Here, we use the SPar [Gri16a] PPI to apply parallelism. We parallelize it into four pipeline stages and the two intermediate stages can be replicated. As discussed earlier, except for the stream region, the rest of the code is mostly identical in all applications. Therefore, this code snippet shows only the stream region.

SPar is a DSL that adds annotations to the code to implement parallelism. Thus, rewriting the original source code is not needed. This example in Listing 6 shows two situations of using the intermediate stages. The second stage of the pipeline runs operator A, while the third stage combines `OperatorB` and `OperatorC`. We put them in a single pipeline stage to show how composable operators can be. The framework API allows these combinations to be easily arranged, and users can build different compositions for better load balancing.

Listing 6 Example of parallel benchmark implemented with SPar in SPBench (only the stream region).

```
/*beginning of the stream region*/
[[spar::ToStream]] while(1){
    Item item;
    if (!spb:Source::op(item)) break;  //Read a piece of data from a external source
    [[spar::Stage, spar::Input(item), spar::Output(item), spar::Replicate()]]
    {
        spb::OperatorA::op(item);       //A single operator in the pipeline stage
    }
    [[spar::Stage, spar::Input(item), spar::Output(item), spar::Replicate()]]
    {
        spb::OperatorB::op(item);       // Example of two operators combined
        spb::OperatorC::op(item);       // in the same pipeline stage
    }
    [[spar::Stage, spar::Input(item)]]
    {
        spb:Sink::op(item);            //Write the result
    }
}
/*end of the stream region*/
```

Listing 7 Customized benchmark implemented with FastFlow (stream region).

```
struct Emitter: ff_node_t<spb::Item>{          // First stage
    spb::Item * svc(spb::Item * task){
        while (1){
            spb::Item * item = new spb::Item();
            if (!spb::Source::op(*item)) break; // Get an item from the source
            ff_send_out(item);
        } return EOS;
    }
};
struct Worker: ff_node_t<spb::Item>{    // Second stage
    spb::Item * svc(spb::Item * item){
        spb::Operator::op(*item);       // Some operation
        return item;
    }
};
struct Collector: ff_node_t<spb::Item>{ // Third stage
    spb::Item * svc(spb::Item * item){
        spb::Sink::op(*item);           // Write the result
        delete item;
        return GO_ON;
    }
}Collector;
```

Listing 7 shows another example of parallel implementation, now using the FastFlow [ADKT17a] PPI. With this PPI, it is necessary to make a minimum change in the original code, which consists only in using a pointer to the data item, as we can see in line 4 of the Listing. Besides, it is necessary to rearrange the stream region to fit the FastFlow requirements. In this example, we used a single intermediate stage that runs a single operator. An example using Intel TBB, for instance, would follow a similar structure. Therefore, both

Listing 6 and 7 can exemplify what most parallel benchmarks look like within SPBench. We present the parallel benchmarks in detail in Chapter 4.

3.2.4 Performance Metrics

So far, we have discussed how the SPBench API works, how sequential applications are implemented from it, and how parallelism can be added. But the goals of this work go beyond giving users an application interface to facilitate writing parallel code. We aim to allow such implementations to be used as benchmarks for stream processing. A crucial aspect of turning these implementations into benchmarks is the addition of performance metrics. The Listing 3 shows in lines 10 and 16 where the measurement operations inside each operator are inserted. The Listing 5 shows in lines 3 and 15 where the metrics are initialized and finalized.

Benchmarks need to provide performance metrics that are representative for their domain. Several metrics can be used to evaluate the performance of stream processing. Bordin et. al [BGM+20] conducted a survey in this respect and identified the metrics that are more frequently found in stream processing benchmarks. These metrics are latency, throughput, and resource usage, such as CPU and memory. Therefore, SPBench also offers such metrics[1].

The SPBench benchmarks natively implement the most common metrics for performance evaluation. That is, parallel implementations in SPBench have these metrics automatically available and users can enable or disable them at execution time. The benchmarks can be evaluated under these metrics at different levels of accuracy and detail. Regarding accuracy, it consists of the granularity of the measurement (sample interval). Here users are able to choose a more course- or fine-grain granularity dynamically.

Regarding detailing, SPBench can actuate in 3 levels. The first level provides the user with a global average of the result of the selected metrics, considering the execution as a whole and measuring latency and throughput from end to end of the pipeline, for instance. This level is important for the user to evaluate the global performance of the application. The second level is able to present such results also as an average per operator individually (only latency in the current version). This is important for the user to evaluate aspects such as bottlenecks and load balancing between operators, allowing the optimization of the parallelism strategy, for instance.

The third level is the monitoring metrics. It can measure the metrics at specific instants of time and also compute instant latency or throughput, which is the average of it in short time intervals. After execution, the benchmarks generate log files containing all the results. This type of evaluation is more intrusive and may cause overheads, so it is

[1]https://spbench-doc.rtfd.io/en/latest/metrics.html

important to take this into account. The raw results are stored in memory, and only after the end of the execution are they computed and presented to the users.

Metric	**Granularity**	**Usage**
Latency	Global average (end-to-end) Global average (per-operator) Per time window Per item	Dynamic and static
Throughput	Global average Per time window	Dynamic and static
CPU and Memory Usage	Global average Per time interval	Static

Table 3.1: List of available performance metrics in SPBench.

Therefore, the metrics can be collected at different levels, such as per time unit, per operator, per source (when using multiple source operators), or global average. Some of them can be combined as well. There are two ways of operating these metrics: static or dynamic. The metrics with the available granularity and modes of operation are shown in Table 3.1.

Statically selected metrics are the metrics that are selected as an argument through the CLI before running the benchmark. It includes monitoring metrics, average latency and throughput, and resource usage. Dynamically selected metrics are the metrics that can be collected at execution time, such as throughput and latency. They can be measured at any point during the benchmark execution through function calls inside the source code. These metrics are available in two modes. The first one returns the global average results from the beginning of the execution to the current moment. The second mode is the instantaneous one, where the metrics are computed over a short period (time window). For instance, to get the average throughput of the last 5 seconds, a user could add `spb::Metrics::getInstantThroughput(5)` to the source code. This type of metric is relevant for testing self-adaptive systems, for example, since it can provide feedback on performance on the fly [VGDF22].

In addition to the performance metrics provided by SPBench, users are completely free to implement their own metrics in the user-side code. For example, they can encapsulate the `Item` class within another structure/class and add additional data for this purpose. Of course, it would not be available for the other benchmarks. However, users could then write a single header file with these new metrics and include it only once in a global configuration file that SPBench provides. It applies not only to new metrics but to any other users' custom mechanisms.

3.2.5 Benchmark Parameterization

In addition to performance metrics, the SPBench benchmarks accept other parameters[2]. These arguments allow users to modify the behaviour of the workload and the input stream. They can be statically set using specific arguments in the command-line interface when running the benchmarks. Here are some examples:

- `-in-memory`: enable in-memory execution.
- `-batch`: sets a fixed size for batches.
- `-batch-interval`: fills batches based on a time interval.
- `-frequency`: set a target number of generated items per second in the source.
- `-freq-pattern`: set a pattern of variation in data stream frequency.

With the exception of `-in-memory`, the other commands can be executed dynamically by inserting specific SPBench commands in the source code. The method `spb::SPBench::setFrequency()`, for example, can be executed at any time. We discuss the batching system in more detail in Chapter 7 and data stream frequency in Chapter 6.

3.2.6 Command-Line Interface

SPBench benchmarks offer a variety of configurations, metrics, and workload management options, as discussed in Sections 3.2.4 and 3.2.5. All these parameters (and many others) are intrinsically part of the benchmarks. That is, users can directly run the binaries of the benchmarks and use these parameters. Therefore, combining these metrics and other parameters with the API allows users to implement stream parallelism more easily and, as a result, obtain highly parameterizable benchmarks. However, to get to this point, users must first install dependencies for the application and only then be able to compile it. Building real-world applications with PPIs may not be a trivial task. In addition, in some cases, it is necessary to find and prepare the input workload of the application before you can run it. While this may not be a challenge for experienced users, repeating these processes over several benchmarks can be time-consuming and error-prone. Thus, as SPBench seeks to facilitate stream processing benchmarking in a comprehensive way, the framework also provides a user interface that facilitates and automates many of these aspects.

[2] https://spbench-doc.rtfd.io/en/latest/management_options.html

The SPBench command line interface (CLI) is implemented in Python. Its goal is to speed up organizing, programming, configuring, running, and analyzing the SPBench benchmarks. It acts as an abstraction layer between users and the whole SPBench system. The CLI is illustrated at the bottom of the SPBench architecture representation in Figure 3.4. Below are some of the main commands users can use with the SPBench CLI. These commands can be used as `./spbench [command]`.

- **‘`install`’** - Install SPBench. This command downloads the application dependencies, the PPIs, and the input workloads and builds everything. Specific applications can be selected.
- **‘`new`’** - Creates and adds a new benchmark to the SPBench suite. It requires the user to select which application will run under it. It needs to be one of the applications supported by SPBench. Users can choose whether to create a benchmark from scratch or to make a copy of an existing benchmark in the suite.
- **‘`edit`’** - Open the source code of a benchmark for the user to edit the source code and add parallelism.
- **‘`edit-op`’** - Open the source code of a specific operator from a given benchmark for the user to edit the source code and add parallelism.
- **‘`configure`’** - Open for editing a JSON file that allows the user to insert specific compilation commands for a given benchmark.
- **‘`global-config`’** - The same as the above, but the commands added to this JSON file are applied for all benchmarks in SPBench. It can overwrite the local configurations of each benchmark or add to them.
- **‘`compile`’** - Compile a benchmark or a set of benchmarks. SPBench automatically generates makefiles for each benchmark, applies users’ custom settings, and compiles the benchmarks.
- **‘`execute`’** - Execute a benchmark or a set of benchmarks. Here users can select workload classes, execution metrics, and other configurations and options, such as changing the benchmark executor or enabling output correctness testing.

Besides the commands that have been mentioned, several others are available. For example, users can use ‘`rename`’ to rename a benchmark, ‘`new-input`’ to add a custom workload class, and ‘`list`’ to list the available benchmarks. Listing 8 shows an example JSON configuration file of a benchmark. The global configuration JSON is similar but with a few extra keys. SPBench reads these JSON files and generates custom Makefiles from them. The user can select a specific compiler to compile only the benchmark, which is

Listing 8 Example of JSON file with compiling settings.

```
{
    "CXX"            : "<compiler for the base applications and SPBench infrastructure>",
    "CXX_FLAGS"      : "<compiling flags for the compiler above>",
    "BENCH_CXX"      : "<specific compiler for the user code, if any>",
    "BENCH_CXX_FLAGS": "<flags for the specific compiler>",
    "MACROS"         : "-DMY_MACRO",
    "PKG-CONFIG": {
        "myPKG_1"    : "pkg-config --cflags --libs mypkg-config",
        "myPKG_2"    : "",
        "myPKG_N"    : ""
    },
    "INCLUDES": {
        "myINC_1"    : "-I $SPB_HOME/path/to/include/",
        "myINC_2"    : "-I /another/path/to/include/",
        "myINC_N"    : ""
    },
    "LIBS": {
        "myLIB_1"    : "-L my/lib/path",
        "myLIB_2"    : "-L $SPB_HOME/another/lib/path",
        "myLIB_N"    : ""
    },
    "LDFLAGS"        : "-mylib1 -mylib2 -mylibn"
}
```

useful when working with a PPI DSL like SPar, for example, which has its own compiler. The '`$SPB_HOME`' is not a real variable. It is just a keyword recognized by the CLI that users can use inside these files to refer to the root path of the SPBench.

CLI Features to Increase User Productivity in SPBench

The most basic CLI commands already help make using benchmarks faster and easier. The commands for downloading and installing library dependencies and input workloads and for generating build files for each benchmark contribute a lot in this regard. However, the SPBench CLI has other features and usage modes that make it even easier and faster to create and use the benchmarks.

1. **Creating new benchmarks from existing benchmarks in the suite.**

   This feature is useful for quickly prototyping variations of existing benchmarks in the suite. This can be done using the '`./spbench new`' command and adding the '`-from <benchmark name>`' argument. For example:

   ```
   ./spbench new -bench ferret_tbb_farm
                 -from ferret_tbb_pipe-farm
                 -app ferret -ppi tbb
   ```

   The above command will create a benchmark called `ferret_tbb_farm`, which will be an exact copy of the `ferret_tbb_pipe-farm` benchmark. The new benchmark will

now be ready to compile and run. All the user has to do is edit the source code as desired. This could be an example of a user who wants to create a version using a single farm of a benchmark that implements a pipeline of farms. The need for the user to manually copy files, adjust paths, modify makefiles, etc., is eliminated. The new benchmark will inherit the same settings and any code changes applied within the operators of the source application.

2. **Moving the SPBench benchmarks to other systems.**

   A common situation: a user has created new benchmarks or modified existing benchmarks locally in SPBench and wants to move these benchmarks to run on a remote machine. For these scenarios, there is a simple solution. Users can simply move the `/benchmarks` directory and replace it in the SPBench root on the other machine. All benchmarks will be ready to run.

3. **Selecting sets of benchmarks.**

   Several CLI commands allow it to run over multiple benchmarks. The '`-bench`', '`-app`', and '`-ppi`' arguments can be used individually or combined to select subsets of the benchmarks. The '`-bench <benchmark name>`' argument, if used alone, will apply the respective command to a single benchmark. But '`-bench all`' applies the command to all benchmarks. The '`-app <application name>`' command applies the command to all benchmarks that implement the selected application. Finally, '`-ppi <PPI name>`' executes the command on all benchmarks implemented with a given PPI. The command below, for example, can be used to compile all benchmarks that use the FastFlow PPI:

   ```
   ./spbench compile -ppi fastflow
   ```

   As mentioned, these arguments can also be used in combination. The example below can be used to run all Lane Detection benchmarks, but only those implemented with TBB:

   ```
   ./spbench compile -app lane_detection -ppi fastflow
   ```

4. **Running benchmarks with multiple parallelism degrees.**

   Running experiments with parallel benchmarks using multiple degrees of parallelism is very common in this area. The most common way to run a parallel benchmark in SPBench with a given number of threads or degree of parallelism is to use the '`-nthreads <`*n*`>`' argument in the 'exec' command. In other benchmarks, this can often be done in a similar way. However, to do multiple executions varying the number of threads, the traditional way is to add the execution command inside a loop that

will iterate over the value of *n*. This usually involves creating execution scripts to automate the experiments.

In SPBench, the user can select multiple degrees of parallelism directly in the run command. For example, the argument '`-nthreads 4`' will execute a benchmark with parallelism degree 4. However, '`-nthreads 1:4`' will execute the benchmark four times with parallelism degrees 1, 2, 3, and 4, respectively. '`-nthreads 4:1`' executes the benchmark in the opposite way. It is also possible to add the iteration step size. For example, '`-nthreads 1:2:8`' runs a benchmark with nthreads = 1, 2, 4, 6, and 8.

5. **Running a benchmark multiple times.**

   Another common practice using benchmarks is to perform multiple runs to average results and compute the error. Doing this in the traditional way is similar to the previous item. Users will typically create execution scripts with loops that repeat the execution. But here, there is additional complexity involved in storing all the data and computing the results correctly.

   The SPBench also has a feature that facilitates this aspect. The user simply adds the '`-repeat` <*n*>' argument to the 'exec' command. This will make the CLI run the benchmark *n* times and, in the end, will present the averages of the selected metrics along with their standard deviation.

6. **Combining the features of the above items 3, 4, and 5.**

   Sometimes users want to perform a combination of what was described in items 3, 4, and 5. That is, run multiple benchmarks, varying the degree of parallelism, and run each one multiple times to get the average results. All this can be done with a single command in SPBench. For example, if users select a range for the degree of parallelism in the exec command (e.g. `-nthreads 2:16`) AND add the `-repeat` <*n*> argument, SPBench will automatically generate a performance log containing the average results for each metric along with the standard deviation. Each line of the log represents a degree of parallelism. If, in addition to this, the user selects multiple benchmarks, SPBench will generate a logo for each of them. Here is an example:

   ```
   ./spbench exec -ppi fastflow ... -nthreads 2:16 -repeat 10
   ```

   The above command will run all benchmarks that use FastFlow as PPI, varying the degree of parallelism from 2 to 16, and calculate the average performance results over 10 runs. This eliminates the need to build execution scripts in similar situations, saving the user time and effort. Also, the resulting log files organize the data as a table. Therefore, from them, it is very easy for users to extract information, analyze the data, and build performance graphs.

3.3 Related Work

The previous section of this chapter introduced the SPBench framework. While there are initiatives aimed at adding abstraction layers for writing parallel code [GDTF17, DDMMT18] or for other aspects of parallelism in stream processing [GVS+19, HH21b, LZS+22, VGDF22], few works aim to parameterize and add abstractions for the benchmark applications in this area. SPBench try to do that and be helpful for researchers to test and evaluate their parallelism solutions and technologies for stream processing. Now that we discussed what the SPBench is about, we can discuss our related work.

The discussion of related work in this thesis is not centralized in one chapter, it is divided between the chapters instead. Therefore most chapters will have a shorter discussion section on related work according to the subject matter of each one. It is essential to clarify that the work discussed here is specifically related to the framework itself. This way, we have searched the literature for frameworks that ease in some way the creation or use of parallel benchmarks for stream processing.

RSPLab [TDVMB17] is a cloud-ready open-source test driver framework to support empirical research for Semantic Web (RSP) and Stream Reasoning (SR). It has been designed to meet the requirements for an RSP test driver, as elicited from existing research on benchmarking RSP systems. The requirements include: (1) allowing integration of any benchmark, (2) engine independence, (3) minimal yet extensible performance metrics set, (4) continuous monitoring, (5) error minimization, (6) ease of deployment, (7) ease of execution, (8) repeatability, (9) data analysis, and (10) data publishing. The architecture of RSPLab comprises four independent elements: Streamer, Consumer, Collector, and Controller. The Streamer provides RDF streams, the Consumer exposes the RSP engines via REST APIs, the Collector continuously monitors performance statistics, and the Controller enables programmatic design and execution of experiments. The implementation of RSPLab uses Docker, InfluxDB, and other custom APIs.

NAMB (Not only A Micro-Benchmark) is a platform for the generation of prototype applications based on their high-level description [PHUK20]. It consists of a framework based on fundamental data stream characteristics that supports a configurable topology description. The aim is to avoid the user having to edit the application code. It can generate a set of synthetic/micro-benchmarks as well as prototypes of Java applications for Apache Flink, Storm, and Heron platforms. The framework also allows users to change input data frequency, parallelism degree, tuple size, etc. Figure 3.5 shows a case-use example for modeling a three-stage micro-benchmark. The authors tested the framework using a data stream benchmark application.

DSPBench is a suite of benchmark applications for distributed data stream processing systems. Besides the benchmark suite, the first version of DSPBench [BGM+20]

```yaml
pipeline:
  tasks:
  - name: word_generator
    parallelism: 1
    data:
      size: 8
      values: 100
      distribution: uniform
    flow:
      distribution: uniform
      rate: 1000

  - name: counter
    parallelism: 2
    routing: hash
    processing: 4.5
    parents:
      - word_generator

  - name: sink
    parallelism: 1
    routing: balanced
    processing: 0.5
    parents:
      - counter
```

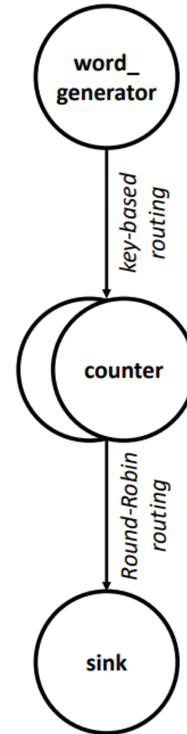


Figure 3.5: Example of a word-counter micro-benchmark generation using NAMB.

Source: [PHUK20]

proposed a framework that had a similar proposal to ours. The goal of their framework was to provide an API for stream processing applications while ensuring that they are executed in the same way (respecting the system's limitations) across the DSPSs. The API aimed to provide the programmer with adapters that allowed translating the application code to a specific version of the DSPSs included in the suite (Apache Flink, Apache Storm, and Apache Spark). Other adapters allowed programmers to add probes within these versions to get performance metrics. The framework could also validate the results to ensure correctness. They also implemented standardized input/output control through a Kafka system for providing the input streams for the applications of the benchmark suite, using a Cassandra storage system for the output. However, the DSPBench API, which was the part that shared common goals with our work, has been discontinued and is no longer available in the latest version of DSPBench[3] (v2.0.0). It moved from single API applications to platform-specific applications.

In [HH21b], the authors developed a framework called *Theodolite* for evaluating the scalability of DSPS platforms. This framework comprises a seven-dimension workload generator derived from industrial IoT for microservice architectures. The middleware is an

[3]https://github.com/GMAP/DSPBench/releases

Apache Kafka system that allows changing the characteristics of the stream, such as item frequency, and adding multiple sources. It includes four benchmarks with Apache Flink and Apache Kafka Streams.

Similar to [HH21b], there are several other frameworks focused on generating input workloads to test different aspects of DSPSs. In [KRK+18], their framework generates realistic workloads to evaluate the latency and throughput of windowing operations. [vTLv16] generates network security monitoring data streams to test the ability of the most common DSPS platforms to process this type of data. [LPDTP+12] proposes a framework to generate representative input workloads for social network applications to evaluate DSPSs. [MBDTE17] go in a similar line, but they created a feature test framework for distributed stream reasoning benchmarking. Instead of evaluating performance, the goal is to compare and evaluate different engines to see if they support specific data stream operations, such as aggregation, union, filter, sorting, etc. The two main components are a data generator that generates query streams and a collector that checks the correctness of the output.

SpinStreams [MDT18] is a framework for predicting the performance of a given stream application and statically restructuring its data flow topology to improve performance. The framework must receive as input a topology description in XML and java functions describing the workload of each task. Then, it tries to apply operations such as join and fission to pipeline stages to correct backpressure and bottleneck issues. It supports Java applications and generates parallelism for the Akka Streams engine [Dav18].

3.3.1 Discussion

Regarding related frameworks to SPBench, there are some interesting solutions. NAMB [PHUK20] was the only work we found that proposes to generate stream processing benchmarks from scratch from a high-level description. However, this framework only generates micro-benchmarks that exclusively target frameworks for DSPSs. Nevertheless, the authors point out the need for benchmarks with a clear workflow description, which can be easily and quickly customized and adapted to different data stream processing contexts. We add to this observation that the lack of such benchmarks for traditional stream applications is even greater, and this is one of the motivations of this work.

RSPLab [TDVMB17] is a framework that focuses on input workload generation. They surveyed from the literature a number of requirements that should be implemented in the framework. Although SPBench is in a slightly different context, it implements most of these requirements as well. RSPLab, however, is very limited in terms of workload parameterization, performance metrics, and benchmark applications.

The remaining related frameworks share some objectives with our work. Most of them try to bring abstractions to data stream workloads. [BGM+20] focuses on standardizing

the workload source with a Kafka system to serve all the benchmarks of the suite. [HH21b] focuses on varying input stream characteristics, such as frequency and complexity of items. In addition to these, [KRK+18, vTLv16, LPDTP+12] generate workloads for specific scenarios, such as window operations, social networks, and security networks. Finally, [MDT18] shares a similar goal as NAMB, but in contrast, it focuses on automatically generating optimizations for applications using parallel patterns from a high-level description.

Table 3.2: Main characteristics of the work related to the SPBench framework.

R.W.	Target applications	Platform	Stream modifiers	Provided metrics	Goal	PPI extensibility
[PHUK20]	Data stream	Distributed	- Data frequency* - Tuple size	–	- Application generation - Automatic parallelism - Synthetic data generation	Difficult
[MDT18]	Data stream	Distributed	- Data frequency	- Throughput	- Automatic parallelism - Performance optimization - Performance analysis	Difficult
[HH21b]	Data stream (microservices)	Distributed	- Data frequency - Multiple sources - Mixed batches	- Scalabillity	- Synthetic benchmark suite - Synthetic data generation - Performance analysis	Not easy
[LPDTP+12]	Data stream (linked data)	Multicore	- Data frequency - Dataset size - Multiple sources	- Throughput	- Input data generation - Performance analysis	Not easy
[vTLv16]	NetFlow processing	Distributed	- Multiple sources	- Throughput	- Micro-benchmark suite - Performance analysis	Not easy
[TDVMB17]	Data stream	Distributed	- Data frequency	- CPU/Mem. usage	- Input workload generation	Not easy
[MBDTE17]	Data stream	Distributed	–	- Correctness	- Input workload generation	Not easy
[KRK+18]	Data stream (social media)	Distributed	- Data frequency - Multiple sources	- Throughput - Event-time latency	- Synthetic data generation - Performance analysis	Easy

SPBench	Traditional and Data Stream Processing	Multicore (currently)	- In-memory processing
			- Data frequency
			- Freq. patterns
			- Batching
			- Multiple Sources
				- Throughput
				  - Average
				  - Instantaneous
				- CPU/Mem. usage
				- Latency:
				  - Processing-time
				  - Per-operator
				  - Average
				  - Instantaneous
					- Benchmark suite
					- Benchmark generation
					- Performance analysis
					- Ease of use
					- Self contained
						Easy

* It can not set data frequencies above one thousand tuples per second.

Table 3.2 summarizes and compares the main features of each related framework, including SPBench in the last row. Most related frameworks focus on distributed platforms and only target data stream applications, as shown in the second column of the table. All of them use JVM languages for hardware abstraction. SPBench targets traditional and data

stream processing in C++. It primarily tries to meet the growing trend of stream processing on multicore systems but also looks at supporting distributed architectures in the future.

The fourth column of Table 3.2 lists any modifications to the input stream that the frameworks support. The most common modification allows the user to change the data arrival frequency, use multiple data sources, or select tuple/batch/dataset size. For now, SPBench supports in-memory processing, data frequency generation (microsecond precision), frequency patterns, multiple sources, and dynamic batch size control.

Regarding metrics (fifth column), throughput is the most common. [HH21b] is the only framework that supports a specific composite metric called *scalability*. Latency can be evaluated in at least three dimensions: event-time, processing-time, and per-operator [KRK+18]. SPBench can also evaluate latency in these three dimensions and more. In addition, SPBench also evaluates memory and CPU usage. Our framework can evaluate and monitor these metrics in different dimensions, as discussed in Section 3.2.4. Hence, SPBench allows users to evaluate benchmarks comprehensively at different layers.

The second-to-last column of Table 3.2 tries to summarize the goals of each framework. Most of them include small benchmark suites with performance evaluation. Other goals revolve around the generation of applications/benchmarks, input data, or parallel code. However, this code/data generation part is usually tied to specific PPIs, which limits their extensibility. The goal of the SPBench is to enable the creation of stream processing benchmarks, provide a suite of benchmarks, facilitate performance analysis, be easy to use, and be self-contained, i.e., it provides all the dependencies specific to each benchmark.

Frameworks marked as difficult to extend (in the last column of the table) are those that require a lot of programming effort to support a new PPI. When it is a task most reserved for the framework's developers. By "Not easy" to extend, we mean that the framework's code still requires some modification, but experienced users can do this. And easy to extend means that no change is required on the framework's source code to add an implementation with a new PPI. SPBench is in this category, where the user only needs to write the parallelism and describe the building dependencies.

To conclude, in our work, we focus on benchmarks for traditional and data stream processing targeting the C++ community. This way, we also target real-world representative stream applications similar to those found in PARSEC. We did not find related frameworks that support this type of application. SPBench is highly extensible and is able to incorporate C++ stream applications from different domains. Our work also includes most metrics identified as important by related work and more. The most used metrics in these frameworks and benchmarks are latency, throughput, and resource usage (CPU and memory) [ABD+16, BGM+20].

The SPBench framework also provides input workload abstractions. It allows users not only to select different workload classes but also to vary item frequency, batch settings,

and data sources. All this is transparent to the programmer, who only needs to set these options in the Command-Line Interface (CLI) at execution time. Furthermore, our framework also focuses on being extensive and is not limited by specific PPIs. So we have not found any other framework that targets the same kind of application as the SPBench, or that is as highly parameterizable as it is, or that provides comprehensive metrics, or that shares the same goals, or that targets ease of use to the same extent.

3.4 Chapter Summary

In this chapter, we introduce the SPBench benchmarking framework. It has as its core element an API that allows you to implement real-world stream processing applications in a simplified, readable, and standardized way. While simplifying the code, it adds workload management mechanisms and performance metrics. It also provides a command-line interface that improves the usability of benchmarks and implements mechanisms that potentially increase the productivity of users. In our literature review, we did not find a framework that proposes and implements something similar to SPBench.

In the next chapters, we will go into more detail about the SPBench benchmarks and the data frequency and batching management system. Chapter 4 presents the set of applications that are currently supported by the framework. From these applications, parallel benchmarks can be generated. Chapter 4 shows how we use SPBench to create the parallel benchmarks that make up the suite. In Chapter 5, Chapter 6, and Chapter 7, we perform performance analysis and evaluate the benchmarks under several parameters.

4. SPBENCH APPLICATIONS AND PARALLEL BENCHMARK SUITE

This chapter first presents the set of applications supported by SPBench. That is, the benchmarking applications that are available for users to use or create new benchmarks from. The second part of this chapter shows how we used the SPBench framework with different parallel programming interfaces to build a parallel benchmark suite.

This chapter is organized as follows. Section 4.1 gives some context about the SPBench and its applications and the scope of our work in this aspect. Section 4.2 presents each application in detail. Section 4.3 discusses the input workloads, how they can be used, and how they are organized into classes in SPBench. In Section 4.4, we describe how we used the PPIs FastFlow, Intel TBB, SPar, OpenMP, ISO C++ threads, GrPPI, and WindFlow to implement parallelism in the SPBench applications. We discuss the related work regarding benchmark suites for stream processing in Section 4.5.

Contents

4.1 CONTEXT **84**
4.2 SPBENCH APPLICATIONS **85**
4.2.1 BZIP2 85
4.2.2 FACE RECOGNITION 86
4.2.3 LANE DETECTION 87
4.2.4 FERRET 88
4.2.5 FRAUD DETECTION 89
4.3 WORKLOAD CLASSES **91**
4.3.1 INPUT WORKLOADS 91
4.3.2 USING CUSTOM INPUT WORKLOADS 93
4.3.3 CORRECTNESS TESTING 94
4.4 BUILDING THE PARALLEL BENCHMARKS **94**
4.4.1 FASTFLOW 95
4.4.2 THREADING BUILDING BLOCKS 96
4.4.3 SPAR 99
4.4.4 OPENMP AND ISO C++ THREADS 99
4.4.5 GRPPI 101
4.4.6 WINDFLOW 103
4.5 RELATED BENCHMARK SUITES **104**
4.5.1 DISCUSSION 105
4.6 CHAPTER SUMMARY **107**

4.1 Context

Ideally, benchmark applications should represent, to some extent, the target scenario they are intended to evaluate, as discussed in Session 2.3. They should provide a means to assess the performance of a system on a range of tasks and workloads. It is important because the performance of a system can vary depending on the specific application or workload it is running. In addition, the applications included in a benchmark suite should be designed to test the system's scalability by providing a range of problem sizes. It allows testing the system's ability to handle larger and more complex workloads, which is an essential aspect of a system's performance. Overall, the applications included in a benchmark suite are important because they provide a way to evaluate the performance of a system on a range of real-world tasks and workloads. It helps to identify the object's strengths and weaknesses under analysis and guide efforts to optimize its performance.

Currently, the SPBench suite comprises five real-world applications. Four are used in traditional stream processing, and one is used in data stream processing. While the type and number of applications provided are important to a benchmark suite, SPBench is not a typical benchmark suite. Thus, we decided to prioritize other aspects during its development. We believe that focusing the initial work on developing the framework, the API, and other features would result in more relevant contributions. Adding more applications is manual work that becomes easier as the API and its documentation are improved. Therefore, adding a large number of applications to the SPBench at the beginning of its development would result in a significant development cost for the framework. Extensive updates to the API, frequent in the design phase, would imply a lot of application code rewriting work. This development cost would have hindered the implementation of more outstanding features and contributions to users.

However, the SPBench framework has reached a more mature level concerning documentation[1], API, CLI, parameterization, and other features. Moreover, these aspects have already been extensively tested and evaluated. Therefore adding new applications should be an important way to go in future work. There is a recent initiative by other researchers to implement sequential C++ versions of the DSPBench benchmarks [BGM+20]. It highlights the demand for such benchmarks and the need for sequential versions of parallel benchmarks. Fraud Detection was the first such application, and we have recently added it to the SPBench. Besides these, we should also look for new applications that exploit traditional stream processing.

All the parallel benchmarks of SPBench were developed on top of these sequential applications it provides. However, the main goal of the SPBench framework itself is to provide sequentially implemented applications with an API that makes it simple for users to

[1] https://spbench-doc.rtfd.io

create benchmarks for stream processing. In other words, it is the role of users to create benchmarks using SPBench. On the other hand, one of the research goals of this work is to expand the area of benchmarks for stream processing. Naturally, this involves the development of benchmarks in this context since there is a lack of C++ stream processing benchmarks. Therefore, to achieve this research goal, we did as any other user would do and used SPBench to build such benchmarks.

Some of the parallel benchmarks provided by SPBench are simply the result of the use of the framework during its development to create benchmarks to test and validate its features. The difference is that we already make these benchmarks available in SPBench, while users would have their custom benchmarks only available locally. In addition, some of the benchmarks were developed by other researchers and contributors that use SPBench. Thus, we encourage new users to contribute and submit their benchmarks to the SPBench suite.

4.2 SPBench Applications

The current application set of SPBench comprises five real-world applications: Bzip2, Lane Detection, Face Recognition, Ferret (PARSEC [BKSL08]), and Fraud Detection. These applications have already been studied and used as benchmarks in prior work [GHDF18a, Víl20, GHDF17, HLGF22, BKSL08, GHDF18b, PLH+21]. However, most of them were not even available in a public repository. Part of the problem was the lack of documentation and software licenses. In SPBench, we have solved such problems, rebuilt the applications to fit the API structure, and made them publicly available.

4.2.1 Bzip2

Bzip2 [Sew17] is a lossless data compression algorithm developed by Julian Seward in 1996. It is a free and open-source software tool widely used for compressing files and has become a popular choice for file compression due to its high compression ratio and fast decompression speed. Bzip2 combines Burrows-Wheeler transform, Huffman coding, and run-length encoding to compress data. It is often used to compress large files, such as database backups and log files, and is supported by various operating systems and software applications. Overall, Bzip2 is a reliable and efficient compression algorithm that has proven to be an important tool for data storage and transmission.

Although Bzip2 compresses data more effectively than the older LZW (.Z) and Deflate (.zip and .gz) algorithms, it is considerably slower. Therefore, parallel implementations that increase compression speed are helpful. This application can be divided into a

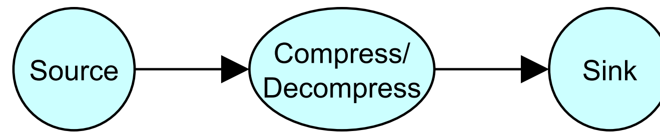


Figure 4.1: Bzip2 flow graph.

Source: [GGSF23] ©2023 Springer Nature.

three-stage Pipeline (source, compress/decompress, sink), as shown in Figure 4.1, and has two operation modes: compress and decompress.

4.2.2 Face Recognition

Face recognition is a computer application that seeks to identify and verify individuals based on their facial features. This technology has a wide range of potential uses, including security and surveillance, law enforcement, and social media. In the field of security and surveillance, face recognition can be used to identify and track individuals as they move through a public space, such as an airport or shopping mall. In law enforcement, face recognition can be used to match a suspect's face to a database of mugshots. In the social media industry, face recognition can be used to automatically tag people in photos or to suggest friends to users based on their facial features.

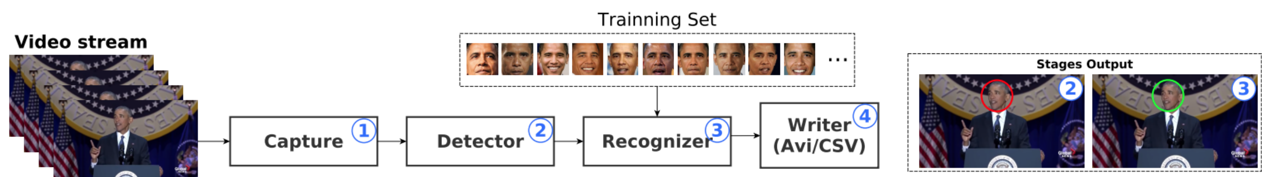


Figure 4.2: Face Recognizer workflow.

Source: [GHDF17]

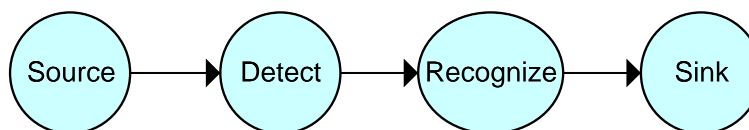


Figure 4.3: Face Recognizer flow graph.

Source: [GGSF23] ©20XX Springer Nature.

In SPBench, the Face Recognizer [Aru13] application tries to match human faces from a video frame against a database of faces. For each video frame, it applies a detection algorithm to detect all the faces in it. Then, it uses a set of face images and compares each

detected face in the frame with the faces on that set. The recognized faces are marked with a circle, and then the frames are written to the output file. This process is illustrated in Figure 4.2. Therefore, this application can be divided into a four-stage Pipeline, represented by Figure 4.3. This application is described with details in Reference [GHDF17].

4.2.3 Lane Detection

Lane detection is a computer application that aims to detect and track the lanes of a road. This technology has a wide range of potential uses, including improving driver safety, reducing traffic accidents, and helping self-driving vehicles navigate roads. In the field of driver safety, lane detection systems can alert drivers if they drift out of their lane. Regarding self-driving vehicles, lane detection is a crucial component for enabling the vehicle to navigate roads safely and avoid collisions. Lane detection systems typically use sensors, such as cameras, to gather data about the surrounding environment and to identify the location and position of lanes on the road.

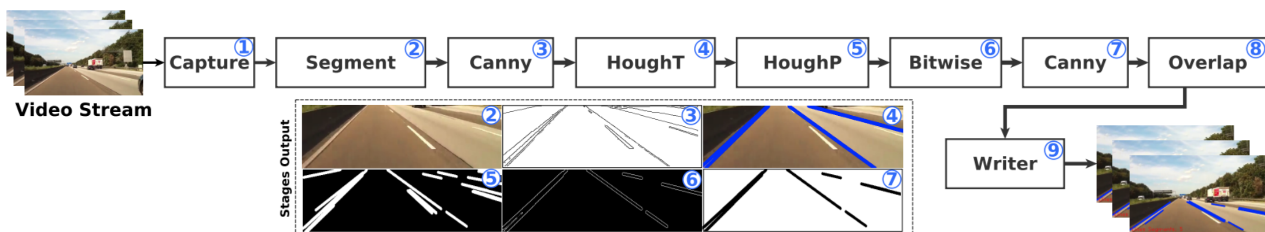


Figure 4.4: Lane Detection workflow.

Source: [GHDF17]

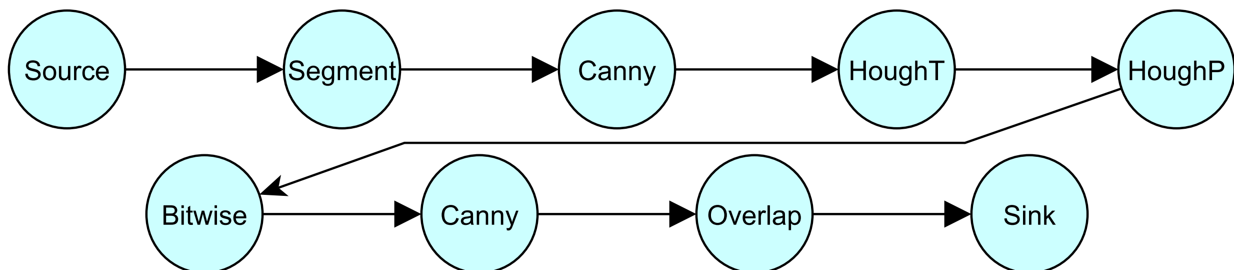


Figure 4.5: Lane Detection flow graph.

Source: [GGSF23] ©20XX Springer Nature.

In SPBench, the Lane Detection application captures each frame of an input video file and applies three computer vision algorithms. It is based on the Lane Detection benchmark used by Griebler et al. [GHDF17]. It can be divided into a nine-stage Pipeline, as shown in Figure 4.5. Through these stages, the detected lanes are marked with straight

lines in a new frame. This new frame with the marked lanes is then overlaid on the original, and the resulting frame is written to the output file. This process is illustrated in Figure 4.4.

4.2.4 Ferret

Ferret is a PARSEC (Princeton Application Repository for Shared-Memory Computers) [BKSL08] application intended for a content similarity search in data such as video, audio, and images. It basically simulates the process of searching and filtering through a large dataset of document records, similar to how a search engine might work. In PARSEC, Ferret was configured for image similarity search [BKSL08], so it uses a dataset of images and a search engine that processes queries and returns results from the dataset. This benchmark is designed to stress the memory hierarchy and interconnect of a shared-memory system and to evaluate its performance under different workloads and configurations. It is often used as a benchmark to compare the performance of different systems or to evaluate the impact of hardware or software changes on the performance of a system.

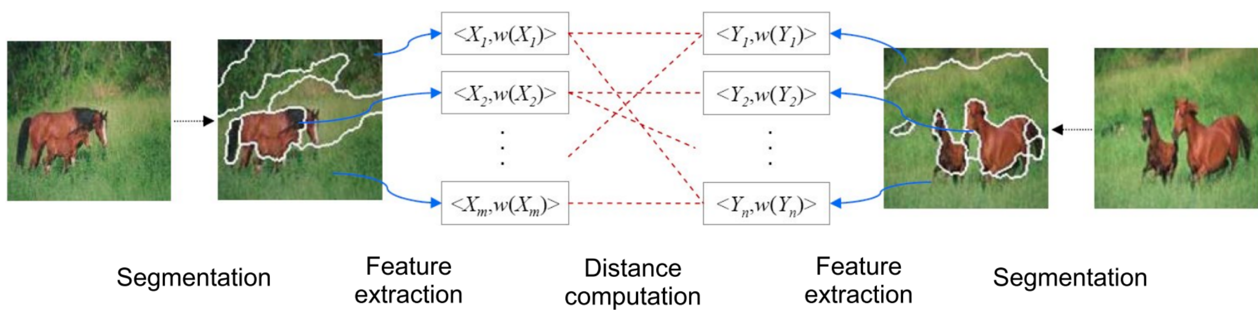


Figure 4.6: Ferret workflow.

Source: [BKSL08]

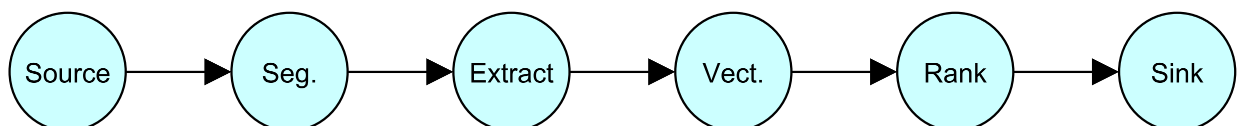


Figure 4.7: Ferret flow graph.

Source: [GGSF23] ©20XX Springer Nature.

In PARSEC, the parallelism is implemented with POSIX Threads using a six-stage Pipeline. In SPBench, we keep the same structure, as represented in Figure 4.7. The first and last ones are source and sink. The second stage performs segmentation, a process that organizes the regions of each image into sets. The third stage computes a 14-dimension feature vector for each of the segmented regions of each frame. The computing process

of the second and third stages is illustrated in Figure 4.6. The fourth stage (vectorization) applies an indexing method, which selects possible similar images. The fifth stage (rank) performs a refined search of the images selected in the fourth stage, ranking the most similar images.

4.2.5 Fraud Detection

Fraud detection is an application that uses Markov chains to predict the probability of a transaction being fraudulent. A Markov chain is a mathematical system that undergoes transitions from one state to another according to certain probabilistic rules [CN06]. In the context of fraud detection, a Markov chain can be used to model the sequence of events that occur during a fraudulent transaction or activity.

The basic idea behind using a Markov chain for fraud detection is to identify patterns in the sequence of events that are indicative of fraudulent activity. These patterns can be represented as transitions between different states in the Markov chain. For example, one state might represent a legitimate transaction, while another might represent a fraudulent transaction. The probability of transitioning from one state to another can be used to calculate the likelihood that a given transaction is fraudulent.

To use a Markov chain for fraud detection, data about past transactions is used to build a model of the chain. This can be done by analyzing the sequence of events that occurred in each transaction and identifying patterns that are indicative of fraud. The model can then be used to evaluate new transactions as they occur in real-time and flag those likely to be fraudulent based on their probability of transitioning between different states in the chain [SKSM08].

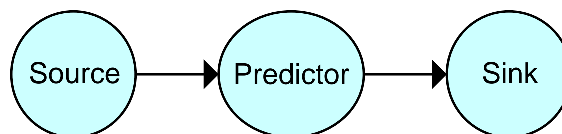


Figure 4.8: Fraud Detection flow graph.

Source: [GGSF23] ©2023 Springer Nature.

This SPBench application was built based on the Fraud Detection benchmark from the DSPBench suite [BGM+20]. It has three processing operators, as shown in Figure 4.8. The first operator reads a database of transactions and keys and generates tuples from that data. The second operator, the Predictor, filters out non-fraudulent transactions and forwards only the fraudulent ones to the third stage. This way, the Predictor operator acts as a filter. The final operator, the sink, receives the fraudulent transactions and may write them to a file if required.

Until recently SPBench did not support any data stream applications, only traditional stream processing applications. Fraud Detection was the last to be added to the suite and it is an application that represents the data stream domain. Its implementation within SPBench is presented in Listing 9.

Listing 9 Sequential Fraud Detection implementation in SPBench.

```
#include <fraud_detection.hpp>

// Predictor state
// It is modified at each iteration by the predictor operator
Markov_Model_Predictor predictor;

int main (int argc, char* argv[]){
    spb::init_bench(argc, argv);
    spb::Metrics::init();
    while(1){
        spb::Item item;
        if(!spb::Source::op(item)) break;
        spb::Predictor::op(item, predictor); // State given as argument
        if(item) spb::Sink::op(item); // Only fraudulent items proceed (filter)
    }
    spb::Metrics::stop();
    spb::end_bench();
    return 0;
}
```

This application posed new challenges to the SPBench API for the following reasons:

1. It has a stateful operator.

   The other SPBench benchmarks only have stateless operators. Keeping states had not been a concern until now. The solution, in this case, was to add the attribute that requires keeping the state in the sequential code and make it evident to users. This way, it is up to the users to find the best way to deal with it. In the sequential version, we handle this by allowing the state to be passed as an argument to the stateful operator, as per line 13 of Listing 9.

2. It has a filter operator.

   While a filter operator is natural in the parallel scenario, we had to add extra control structures to implement this sequentially. Previously the SPBench didn't foresee that items could be dropped along the way. To solve this problem we overloaded the operator in the Item class and now it can be used as a boolean type. This way, it can be filtered with a simple syntax in the sequential version, as line 14 of Listing 9 shows.

3. It requires Source and Sink parallelism to scale performance.

This involved creating new mechanisms to keep performance metrics consistent. Until then, a single item counter within the Source or Sink was enough to measure throughput. But with parallel sources, it was necessary to synchronize the counting of these items. So we created the SPBench Metrics class attribute called "`my_items_counter`". It can be used when users intend to run parallel sources and sinks. It basically overwrites the original item counters with this custom counter that can be manipulated on the user's side.

4. The parallel Fraud Detection requires key-by partitioning of data.

   A key, in the SPBench case, is an attribute of the class Item. However, SPBench implements batches natively and does not support partitioning data by key within batches. Therefore, we had to disable the batching system to implement key-based data partitioning in parallel versions. Anyway, some PPIs like WindFlow are able to run themselves with batches if required, not relying on SPBench for this.

4.3 Workload Classes

SPBench provides different classes of input workloads for each application[2]. In the current version, all applications have the classes named `test`, `small`, `medium`, `large`, and huge. Before using it, users need to download these input files from a remote repository. This can be done through the '`./spbench download-inputs`' command.

```
./spbench download-inputs
    -app <application_name> (optional)
    -class <class_id> (optional)
```

This command will download all required files and generate the input workload classes for all applications. Users can download the inputs for a single application using the '-app' argument. Similarly, the argument '`-class`' can be used to download a single class. This is useful to avoid downloading large files if they will not be required. Users must provide an input workload class when running the './spbench exec' command. This command has an '`-input <input_class>`' argument for this purpose.

4.3.1 Input Workloads

Here we describe what the input workload classes are for each of the SPBench applications. Some of these benchmarks that are now part of the SPBench were used by

[2]https://spbench-doc.rtfd.io/en/latest/workloads.html

other authors in past work without a clear definition of the input workloads. Some of these inputs were already outdated, and the workload needed to be increased to test newer and larger systems better. We added new inputs for Bzip2 and Lane Detection and restructured those for Face Recognizer and Fraud Detection. The definition of the workload classes is based on a simple methodology. We started by defining a class called 'huge'. It should take at least 100 seconds to process sequentially on a 1GHz processor. The lower classes are derived from this.

- Bzip2

  The input workloads for Bzip2 are dump files from the Wikipedia database[3]. The input workload classes for it in SPBench are organized as follows:

  - `Test` class: enwiki-20211120-pages-articles-multistream-index9.txt (16.8 MB)
  - `Small` class: enwiki-20220601-pages-articles-multistream15.xml (129.7 MB)
  - `Medium` class: enwiki-20211120-all-titles-in-ns0 (349.1 MB)
  - `Large` class: enwiki-20211120-pages-articles-multistream9 (2.1 GB)
  - `Huge` class: enwiki-20211120-pages-articles-multistream9-2x.xml (4.2 GB)

  Users can use the class names with the '_d' suffix (e.g., large_d) to access the compressed file versions (`.bz2`) for running the benchmarks in decompress mode.

- Lane Detection

  The input workload for the Lane Detection application is a single video file. The video was recorded from a vehicle driving on the road. It is an mp4 video with 640×360 or 1280×720 resolution with 30 frames per second (H.264 codec). This same video is used for all workload classes but at different lengths.

  - `Test` class: 3-second video (LQ = 0.57 MB and HQ = 2 MB)
  - `Small` class: 15-second video (LQ = 5.5 MB and HQ = 14.8 MB)
  - `Medium` class: 30-second video (LQ = 12 MB and HQ = 32.1 MB)
  - `Large` class: 60-second video (LQ = 25.4 MB and HQ = 66.1 MB)
  - `Huge` class: 120-second video (LQ = 50 MB and HQ = 130 MB)

  The default video resolution is 360p. However, SPBench also provides the same videos in 720p resolution. To access it, users can add a '-HQ' suffix (e.g., large-HQ).

[3] https://dumps.wikimedia.org/enwiki/

- Face Recognizer

  The Person Recognition input files consist of a set of pictures of former US president Barack Obama's face plus a video recorded during a public talk that contains images of his face and also the faces of other people in the audience at some points. The set of pictures is used for training the application. The video (360p resolution) is used to try to recognize the face of the former president.

  All workload classes use the same video but at different lengths, as described below:

  - `Test` class: 0.3-second video
  - `Small` class: 1-second video
  - `Medium` class: 3-second video
  - `Large` class: 15-second video
  - `Huge` class: 30-second video

- Ferret

  For Ferret, the original workloads available on the PARSEC website[4] are used. In SPBench, the '`native`' workload class of PARSEC is available under the 'huge' alias. It consists of 3.500 image queries, a database with 59.695 images, and finding the top 50 images [BKSL08].

- Fraud Detection

  This application receives as input a file with the chain model and a dataset of transactions and keys. Since this is an application from the DSPbench suite [BGM+20], we use the same dataset they provide[5]. In this application, we use a different logic to build the workload classes inside the SPBench. Instead of being tied to the size of the input file, here, the classes are defined as execution time. So in the 'huge' class, the application will process transactions for 60 seconds. This execution time is reduced to 30, 10, 5, and 1 second for the `large`, `medium`, `small`, and `test` classes, respectively.

4.3.2 Using Custom Input Workloads

To run a benchmark, users must choose an input workload class. However, users can also run a benchmark with custom workload classes. For instance, a user may want to use a different input video file for the Lane Detection benchmarks or change some parameters. To do that, this new input must first be registered in SPBench to enable the new class. This process can be done using the '`new-input`' command in the CLI.

[4] https://parsec.cs.princeton.edu/download.htm
[5] https://github.com/GMAP/DSPBench/wiki/Workloads

4.3.3 Correctness Testing

Some SPBench benchmark applications allow the result to be checked at the end of the execution. This is done by comparing the `md5` hash value of the benchmark output against an expected `md5` value, which is a pre-computed `md5` hash stored in the input database of SPBench. The expected `md5` results from running the respective input with the sequential benchmark. Users can add correctness testing for a custom input using the '`-md5`' argument with the '`new-input`' command.

Of course, this md5-comparison method only works when the benchmark produces an output file and is deterministic. This is true for Lane Detection, Person Recognition, and Bzip2. These applications have ordering constraints and the correct output is always the same. However, this is not true for Ferret, which processes out-of-order items, or Fraud Detection, which processes an arbitrary number of items every time. In the future, we may look for better and new methods to address such cases.

4.4 Building the Parallel Benchmarks

The SPBench is a framework that provides applications that support stream parallelism in a readable, simplified, highly parameterized, and metrics-driven way. But to reiterate, SPBench alone does not provide the means to implement parallelism. To implement stream parallelism in the SPBench benchmarks, some external parallel programming interface (PPI) must be used. However, some of the most relevant state-of-the-art PPIs have been used to create parallel benchmarks in SPBench and these benchmarks come with the framework.

SPBench provides benchmarks using the PPIs OpenMP, ISO C++ Threads, Intel TBB, FastFlow, SPar, WindFlow, and SPar. We believe that "all-inclusive" is a way to improve the framework's usability and benchmarks. Therefore, the different PPIs used in the benchmark suite are also provided by SPBench. The exceptions are C++ threads and OpenMP. The C++ threaded implementations use only the standard C++ library. In the case of OpenMP, this is a classic parallelism library that is easy to install and is usually already present in most HPC systems. So for these two PPIs, the SPBench provides only one library that implements a shared queue that can be used to communicate between the stages of the pipeline. Of course, users can implement their own mechanisms for this.

The easiest way to create a benchmark with one of the PPIs included in the SPBench is to create a new benchmark from a copy of a similar benchmark implemented with the same PPI. That way, the new benchmark will already have a ready-made JSON configuration file, and the user can reuse some of the parallel code from the original

benchmark. Even if the new benchmark requires some modification to the configuration file, this would still be the easiest way to go. In Section 3.2.6, we show an example of how this works.

Creating a new benchmark using a PPI not available in the SPBench is also simple. Users only have to create a new benchmark in SPBench and add the corresponding information, such as the PPI path, to the configuration file. In this case, users need to be aware that when moving benchmarks to another system, they will also need to install the PPI in the new location. They will not be able to rely on SPBench for this task. For example, this may mean editing the configuration file and updating the paths.

Below we briefly discuss how parallelism is exploited with each PPI in SPBench. All source codes we discuss are publicly available in the SPBench repository[6].

4.4.1 FastFlow

Listing 10 Example of a SPBench benchmark implemented with FastFlow.

```
struct Emitter: ff::ff_node_t<spb::Item>{
    spb::Item * svc(spb::Item * task){ /* Emitter code */ }
};
struct Worker: ff::ff_node_t<spb::Item>{
    spb::Item * svc(spb::Item * item){
        spb::Operator::op(*item);
        return item;
    }
};
struct Collector: ff::ff_node_t<spb::Item>{
    spb::Item * svc(spb::Item * item){ /* Collector code */ }
} Collector;
int main (int argc, char* argv[]){
    ... // SPBench starting routines
    std::vector<std::unique_ptr<ff::ff_node>> workers;
    for(int i=0; i<spb::nthreads; i++){ workers.push_back(ff::make_unique<Worker>()); }
    ff::ff_OFarm<spb::Item> farm(move(workers));
    Emitter E;
    farm.add_emitter(E);
    farm.add_collector(Collector);
    farm.run_and_wait_end();
    ... // SPBench stopping routines
}
```

Listing 10 presents an example of how parallelism has been exploited in benchmarks with FastFlow. This example shows a parallel implementation with a single farm. FastFlow is a header-only, pattern-based library and already provides a built-in Farm structure. In the case of the example, an ordered farm is created, defined as `ff::ff_OFarm` in FastFlow.

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

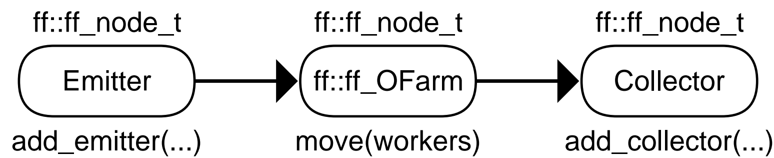


Figure 4.9: Farm implementation in FastFlow.

Source: [GGSF23] ©2023 Springer Nature.

The Emitter, the Workers, and the Collector are structures that extend the FastFlow class `ff::ff_node_t`. In line 15 of the listing, a vector of smart `ff::ff_node` pointers is created. On line 16, this vector is filled with a respective number of workers given by `spb::nthreads`. This variable `spb::nthreads` has the value set by the `-nthreads` argument when running the benchmark. The farm object of type `ff::ff_OFarm` is created from this vector of workers. In this farm, the `add_emitter` and `add_collector` methods are used to finish building the Farm. Thus, we can see the resulting structure as a three-stage pipeline with replicated middle stage. Figure 4.9 illustrates this structure.

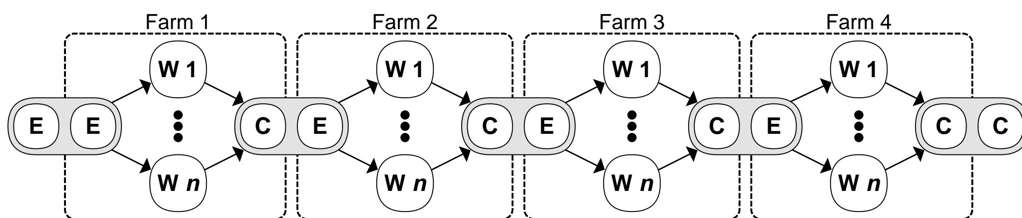


Figure 4.10: Structure of a Pipeline of Farms implementation with FastFlow in SPBench.

Source: [GGSF23] ©20XX Springer Nature.

The pipeline of farms composition in FastFlow is more complex and demands many more lines of code. Its complexity increases when trying to build an optimized version of it that uses fewer nodes and, consequently, fewer threads. The lack of documentation does not help in this respect. We present an example of our pipeline of farms implementation in the Listing 11. To build it, we set up a farm for each intermediate stage of the application and then add these farms to their respective positions in an `ff::ff_pipeline`. Figure 4.10 illustrates the structure of a pipeline of farms in SPBench. Notice that Emitter and Collector nodes were merged into a single node to optimize the use of threads.

4.4.2 Threading Building Blocks

The single farm implementations of TBB benchmarks are not that different from FastFlow implementation. The difference is that TBB does not provide a built-in farm structure. Therefore it is necessary to simulate this structure using a pipeline. Here we separate the application operators into three classes that extend `tbb::filter`, as

Listing 11 Pipeline of farms implemented with FastFlow in SPBench (Ferret benchmark).

```
struct Source: ff::ff_node_t<spb::Item>{...};
struct Segmentation: ff::ff_node_t<spb::Item>{...};
struct Extract: ff::ff_node_t<spb::Item>{...};
struct Vectorization: ff::ff_node_t<spb::Item>{...};
struct Rank: ff::ff_node_t<spb::Item>{...};
struct Sink: ff::ff_minode_t<spb::Item>{...};
int main(int argc, char *argv[]) {
    spb::init_bench(argc, argv);

    /* Segmentation stage */
    ff::ff_pipeline *pipeSeg = new ff::ff_pipeline;
    ff::ff_farm farmSeg;
    std::vector<ff::ff_node *> segWorkers;
    for(int i=0;i<spb::nthreads;++i) {
        segWorkers.push_back(new Segmentation());
    }
    farmSeg.add_workers(segWorkers);
    Source *source = new Source();
    farmSeg.add_emitter(source);
    pipeSeg->add_stage(&farmSeg);

    /* Extract stage */
    ff::ff_pipeline *pipeExt = new ff::ff_pipeline;
    ff::ff_farm farmExt;
    std::vector<ff::ff_node *> extWorkers;
    for(int i=0;i<spb::nthreads;++i) {
        extWorkers.push_back(new Extract());
    }
    farmExt.add_workers(extWorkers);
    pipeExt->add_stage(&farmExt);

    /* Vectorization stage */
    ...
    /* Rank stage */
    ...
    Sink *sink = new Sink();

    ff::ff_pipeline pipe;
    pipe.add_stage(pipeSeg);
    ...
    pipe.add_stage(sink);

    spb::Metrics::init();
    pipe.run();
    pipe.wait();
    spb::Metrics::stop();
    spb::end_bench();
    return 0;
}
```

presented in Listing 12. Then a `tbb::pipeline` object is created and the stages are placed in their respective order within the pipeline: `Source -> Worker -> Sink`. Source and Sink are sequential (serial) stages, and this example implements an ordered farm. Therefore

Listing 12 Example of a SPBench benchmark implemented with Intel TBB.

```
class Source : public tbb::filter{
    Source() : tbb::filter(tbb::filter::serial_in_order) {}
    void* operator() (void*){ /* Source code */ }
};
class Worker : public tbb::filter{
public:
    Worker() : tbb::filter(tbb::filter::parallel) {}
    void* operator() (void* new_item){
        spb::Item * item = static_cast <spb::Item*> (new_item);
        spb::Operator::op(*item);
        return item;
    }
};
class Sink : public tbb::filter{
    Sink() : tbb::filter(tbb::filter::serial_in_order) {}
    void* operator() (void* new_item){ /* Sink code */ }
};
int main (int argc, char* argv[]){
    ... // SPBench starting routines
    tbb::task_scheduler_init init_parallel(spb::nthreads);
    tbb::pipeline pipeline;
    Source source;
    pipeline.add_filter(source);
    Worker worker;
    pipeline.add_filter(worker);
    Sink sink;
    pipeline.add_filter(sink);
    pipeline.run(spb::nthreads * 10); // number of tokens
    ... // SPBench stopping routines
}
```

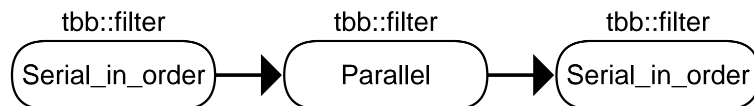


Figure 4.11: Structure of a farm implementation with TBB in SPBench.

Source: [GGSF23] ©2023 Springer Nature.

both are defined as `serial_in_order` filters. The Worker stage is the stage that can be replicated, so it is a parallel filter. This structure is illustrated in Figure 4.11.

Unlike in FastFlow, where each node is statically assigned to a thread, in TBB, this is dynamic. This is because TBB uses a different task scheduling policy [VAR19] based on a work-stealing strategy. Here the maximum parallelism is set by the `task_scheduler_init init_parallel()` command, and parallelism occurs more dynamically. Implementing a pipeline of farms in TBB just involves creating more parallel filter stages and adding them to the pipeline.

4.4.3 SPar

TBB and FastFlow require the sequential application to be somewhat reorganized into classes/structures. The PPI SPar, however, is a DSL based on C++ code annotations and aims to apply parallelism with minimal change to the original code. In Listing 6 of Chapter 3, we have shown an example implementation with the SPar in the SPBench. It basically consists of adding the `[[spar::ToStream]]` annotation to denote the stream region and `[[spar::Stage]]` to denote the stages of the pipeline. A Farm can be built from this by adding `spar::Replicate()` to the middle-stage annotation. Building a pipeline of farms consists of just adding more stage annotations.

4.4.4 OpenMP and ISO C++ Threads

Listing 13 Example of operator and queue structures used to implement a farm with OpenMP and ISO C++ threads in SPBench.

```
class SharedQueue { /* Shared queue code */ }

void emitter(SharedQueue<spb::Item> * outputQueue){ /* Emitter code */ }
void worker(SharedQueue<spb::Item> * inputQueue, SharedQueue<spb::Item> * outputQueue){
    /* Worker code */
}
void collector(SParSharedQueue<spb::Item> * inputQueue){ /* Collector code */ }

int main (int argc, char* argv[]){
    ... // SPBench starting routines
    SharedQueue<spb::Item> * queueA = new SharedQueue<spb::Item>(spb::nthreads);
    SharedQueue<spb::Item> * queueB = new SharedQueue<spb::Item>(spb::nthreads);

    /* OpenMP or C++ threads specific code */

    ... // SPBench stopping routines
}
```

The implementations using ISO C++ threads and OpenMP share some aspects. Both use the same classes for the stages and the shared queue for communication among the stages. This is exemplified in Listing 13. SPBench provides a header file with the SharedQueue class if users want to use it when creating new benchmarks. This class implements a priority queue. The construction of the pipeline and parallelism occurs in different ways, of course.

Listing 14 shows how this is done with C++ threads. A thread is created to run the Emitter, another to run the Collector, and a thread vector to run the n Workers, similar to how it is built in FastFlow. But here, the input and output queues of each stage must

Listing 14 Example of a C++ threads farm implementation in SPBench (parallel region).

```
...
std::thread firstStage(emitter, queueA);
std::vector<std::thread> secondStage;
for(int i = 0; i < spb::nthreads; i++)
    secondStage.push_back(std::thread(worker, queueA, queueB));
std::thread thirdStage(collector, queueB);

firstStage.join();
for (auto& t : secondStage) t.join();
thirdStage.join();
...
```

be defined manually. Also, it is necessary to synchronize the threads at the end of the execution (lines 8-10 of the listing).

Listing 15 Example of an OpenMP farm implementation in SPBench (parallel region).

```
...
omp_set_num_threads(spb::nthreads + 2);
#pragma omp parallel shared(queueA, queueB)
#pragma omp single nowait
#pragma omp taskgroup
{
    #pragma omp task
    {
        emitter(queueA);
    }
    for(int i = 0; i < spb::nthreads; i++){
        #pragma omp task
        {
            worker(queueA, queueB);
        }
    }
    #pragma omp task
    {
        collector(queueB);
    }
}
...
```

The specific code implementation with OpenMP is in Listing 15. Note that at the beginning, the `omp_set_num_threads` is set to `spb::nthreads + 2`. This “+2” represents the two extra threads needed to run the Emitter and the Collector. The implementation of the pipeline stages in OpenMP uses the `#pragma omp task` directive. For both C++ threads and OpenMP, implementing a farm pipeline consists of adding more parallel stages to the pipeline and creating extra queues for the new stages.

4.4.5 GrPPI

GrPPI (generic and reusable parallel pattern programming interface) is another PPI that provides structured parallel patterns for stream processing. It is highly modular, allowing easy composition of parallel patterns. It proposes to act as a switch between different parallel programming interfaces. In its latest release, GrPPI allows running applications with four backends: ISO C++ threads, FastFlow, OpenMP, and Intel TBB.

Listing 16 Example of a Bzip2 benchmark in SPBench using GrPPI with a single farm.

```
1  void farm_func(grppi::dynamic_execution & ex) {
2    tbb::task_scheduler_init init(spb::nthreads);
3    grppi::pipeline(ex,
4      []() std::mutable -> optional<spb::Item> {
5        spb::Item item;
6        if(!spb::Source::op(item)) {
7          return {};
8        } else { return item; }},
9      grppi::farm(spb::nthreads,
10     [](spb::Item item) {
11       spb::Compress::op(item);
12       return item;
13     }),
14     [](spb::Item item){ spb::Sink::op(item); }
15   );
16 }
17 grppi::dynamic_execution execution_mode(){
18   std::string backend = spb::SPBench::getArg(0);
19   bool ordering = true;
20   int queue_size = 1;
21   if (backend == "tbb"){
22     int tbb_tokens = spb::nthreads*10;
23     auto tbb_exec = grppi::parallel_execution_tbb(spb::nthreads, ordering);
24     tbb_exec.set_queue_attributes(queue_size, grppi::queue_mode::blocking, tbb_tokens);
25     return tbb_exec;
26   } else if ...
27 }
28 int main (int argc, char* argv[]){
29   spb::init_bench(argc, argv);
30   spb::Metrics::init();
31   auto ex = execution_mode();
32   farm_func(ex);
33   spb::Metrics::stop();
34   spb::end_bench();
35   return 0;
36 }
```

Listing 16 shows what a SPBench benchmark with GrPPI looks like. It represents almost a complete implementation of a Bzip2 benchmark using a GrPPI farm. To implement the `farm_func` function, we rely on the code examples that GrPPI provides. 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. It was necessary to use the `tbb::task_scheduler_init()` (line 2) since we could not control the number of TBB threads directly through GrPPI.

The function `execution_mode` is used to dynamically select and configure a backend. We enable 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 omit the rest of this function because it is basically the same code used for `parallel_execution_tbb` but replicated to the other backends.

Backends are dynamically selected at runtime. The command `SPBench::getArg()` gets a custom execution argument from the user. Thus, to run this benchmark with the `GrPPI-TBB` backend in SPBench, varying from 1 to 40 threads, repeating the execution 3 times, and getting performance metrics, the following command can be used:

```
./spbench exec -bench bzip2_grppi_farm -input huge -nthreads 1:40
  -repeat 3 -latency -throughput -memory-usage -user-arg tbb
```

Listing 17 Pipeline of farms implementation with GrPPI.

```
1  grppi::pipeline(ex,
2      []()std::mutable->optional<spb::Item item>{/* first stage */},
3      grppi::farm(nthreads, [](spb::Item item) {/* second stage */}),
4      ...
5      grppi::farm(nthreads, [](spb::Item item) {/* second last stage */}),
6      [](spb::Item item){/* last stage */}
7  );
```

The pipeline farm pattern we used with GrPPI is described in Listing 17. It is similar to the single farm in Listing 16, but we add more farms in sequence.

Listing 18 Pipeline-Farm-Pipeline implementation with GrPPI (we call it “farm of pipelines” or “farm-pipeline” to simplify).

```
1  grppi::pipeline(ex,
2      []()std::mutable->optional<spb::Item item>{/* first stage */},
3      grppi::farm(spb::nthreads,
4          grppi::pipeline(
5              [](spb::Item item) {/* second stage */},
6              ...
7              [](spb::Item item) {/* second last stage */}
8          )
9      ),
10     [](spb::Item item) {/* last stage */}
11 );
```

To better show the ability to build composite patterns in GrPPI, we also implemented another composition called farm of pipelines/farm-pipeline. This second version has a pipeline with a single farm inside, and each farm worker runs another pipeline. Listing 18 shows how we built this implementation in GrPPI.

4.4.6 WindFlow

Listing 19 Fraud Detection pipeline stages with WindFlow.

```
// Source_Functor class
class Source_Functor{
private:
    long generated_tuples; // each source have its item counter
public:
    Source_Functor() { generated_tuples = 0; }
    void operator()(wf::Source_Shipper<spb::Item> &shipper){
        while (1){ // generation loop
            spb::Item item;
            if(!spb::Source::op(item)) break;
            shipper.push(std::move(item));
            generated_tuples++;
        }
        spb::Metrics::my_items_counter.fetch_add(generated_tuples);
    }
    ~Source_Functor(){}
};

// Predictor_Functor class
class Predictor_Functor{
private:
    Markov_Model_Predictor predictor; // predictor state
public:
    Predictor_Functor(){}
    bool operator()(spb::Item &item){
        spb::Predictor::op(item, predictor);
        if(item) return true;
        return false;
    }
    ~Predictor_Functor(){}
};

class Sink_Functor{ /* Sink code */};
```

WindFlow [MTG+19] is a PPI for data stream parallelism in shared-memory architectures. So far SPBench includes only one application from this domain (Fraud Detection) and it has only been parallelized with WindFlow. To write the parallel code we have relied entirely on the implementation made available by the author of WindFlow in a repository called StreamBenchmarks[7]. The authors implement some DSPBench benchmarks in this repository using WindFlow, including the Fraud Detection benchmark. Listing 19 presents how the stages of this application are implemented with WindFLow inside SPBench.

Listing 20 presents the parallelism construct with WindFlow in Fraud Detection. We create a `wf::MultiPipe` object and add its respective stages. Notice on line 15 that the item has a key that can be used to partition the data appropriately. The degree of

[7] https://github.com/ParaGroup/StreamBenchmarks

Listing 20 Parallel Region of WindFlow Fraud Detection.

```
int main (int argc, char* argv[]){
    ... // SPBench starting routines
    wf::PipeGraph topology("FraudDetection",
                            wf::Execution_Mode_t::DEFAULT,
                            wf::Time_Policy_t::INGRESS_TIME);
    Source_Functor source_functor;
    wf::Source source = wf::Source_Builder(source_functor)
                        .withParallelism(stoi(spb::SPBench::getArg(0)))
                        .withName("Source")
                        .build();
    Predictor_Functor predictor_functor;
    wf::Filter predictor = wf::Filter_Builder(predictor_functor)
                        .withParallelism(spb::nthreads)
                        .withName("Predictor")
                        .withKeyBy([](const spb::Item &item)->size_t {return item.key;})
                        .build();
    Sink_Functor sink_functor;
    wf::Sink sink = wf::Sink_Builder(sink_functor)
                    .withParallelism(stoi(spb::SPBench::getArg(1)))
                    .withName("Sink")
                    .build();
    wf::MultiPipe &mp = topology.add_source(source);
    mp.add(predictor);
    mp.add_sink(sink);
    topology.run();
    ... // SPBench stoping routines
}
```

parallelism of the Source and Sink is captured and set through `SPBench::getArg()`, which takes command-line arguments from the users.

4.5 Related Benchmark Suites

In [SCS17], the authors analyze the main characteristics and the behavior of IoT applications, describing common task patterns used in streaming applications for IoT. From this study, they create RIoTBench, a micro-benchmark suite that regroups a set of 27 IoT tasks that cover different patterns. Then, they use these tasks to build the graphs of four data stream benchmark applications. The authors executed the benchmarks using Apache Storm and evaluated throughput, latency, resource usage, and a metric called *jitter*, which calculates the variation between the expected and actual output throughput.

StreamBench [LWXH14] provides a benchmark suite with the goal of evaluating DSPSs. It comprises 7 micro-benchmarks based on Weblogs and network traffic processing applications. These micro-benchmarks are organized into four workloads to test a system's performance, fault toleration, and durability. For performance evaluation, they measure throughput and latency. The benchmark covers different dataflow composition patterns and common tasks like grep and WordCount and compares Storm and Spark Streaming.

There is another benchmark called StreamBench [Wan16] that also specifically targets DSPSs. Its suite comprises three micro-benchmarks: AdvClick, WordCount, and K-Means. They provide implementations using Apache Flink, Apache Storm, and Apache Spark. In their experiments, they evaluated the latency and throughput performance of the DSPSs.

SparkBench [ABD+16] is a framework-specific benchmark suite for Apache Spark. It includes four categories of applications like graph computation and SQL queries. The benchmarks evaluate CPU, memory, disk, and network IO, intending to identify the best configurations to improve Spark's performance

Bordin [BGM+20] proposed a benchmark suite to provide a common reference for DSPS evaluation. It includes 16 benchmark applications from several domains. The benchmark suite comprises parallel implementations using Apache Flink, Apache Storm, and Apache Spark. It also provides Java Threads implementations for single-node execution. The authors identified the most frequently used metrics in related work and used them to evaluate the benchmarks regarding latency, throughput, scalability, tuple loss, and resource usage.

StreamIt [TA10] is a compiler and programming language focused on stream processing applications. The compiler provides performance optimizations, while the language can provide programming abstractions. Although it comes within a benchmark suite, it only supports the StreamIt language and architecture. Moreover, StreamIt benchmarks are limited only to the dataflow and data stream processing domain.

The Princeton Application Repository for Shared-Memory Computers (PARSEC) [BKSL08] is a benchmark suite that comprises computationally intensive multi-threaded programs developed to test multi-core architectures. Although it includes 13 representative real-world applications, only 3 are from the traditional stream processing domain (Dedup, Ferret, and x264). The parallelism is explored using POSIX Threads, Intel TBB, and OpenMP.

4.5.1 Discussion

Although some benchmark suites for stream processing are available, there are still gaps to fill. We summarize the related work in Table 4.1 and compare them with our work. Most of the related work targets only data stream processing applications [SCS17, LWXH14, Wan16, ABD+16, BGM+20]. These applications intersect the domains of Big Data and IoT and are developed using frameworks for DSPSs targeting distributed platforms. Besides, they use only JVM languages, such as Java and Scala. Also, none provide sequential versions of the benchmarks, making it more difficult to port the benchmarks to other platforms. The only benchmark suite for data stream processing that provides more

Table 4.1: Related benchmark suites.

RW	Num. of SP applications	Type of benchmarks	PPIs	Provide Seq. implementations	SP Domain	Program. Language	Goal
[SCS17]	4	Synthetic	Storm	No	Data-stream	Java	Evaluate DSPSs for IoT
[LWXH14]	7	Micro-benchmark	Storm and Spark	No	Data-stream	Scala	Evaluate PPIs for DSPSs
[Wan16]	3	Micro-benchmark	Flink, Storm and Spark	No	Data-stream	Java	Evaluate PPIs for DSPSs
[ABD+16]	8	Syntethic/ Micro-benchmark	Spark	No	Data-stream	Scala	Evaluate Spark configurations
[BGM+20]	16	Micro-benchmark and real-world	Flink, Storm, Spark, and Java Threads	No	Data-stream	Java	Evaluate PPIs for DSPSs
[TA10]	9	Micro-benchmark and real-world	StreamIt	Yes	Traditional SP	C	Evaluate the performance of the language and compiler
[BKSL08]	3	Real-world	PThreads, TBB, and OpenMP	Yes	Traditional SP	C	Evaluate multi-core Architectures
SPBench Benchmark Suite	5	Real-world	OpenMP, FastFlow, TBB, SPar, GrPPI, ISO C++ threads, and WindFlow	Yes	Any	C++	Evaluate PPIs for C++ stream processing

realistic benchmarks is [BGM+20]. The others comprise small tasks and combine those tasks to build synthetic and micro-benchmarks.

On the other hand, are the benchmark suites like StreamIt [TA10] and PARSEC [BKSL08]. Both include traditional stream processing applications. However, both are pretty limited in terms of programming language, parallelism exploitation, execution metrics, and parametric options. [DSDMT+17] realized the limitations of PARSEC and the need for benchmarks that explore structured parallel programming and extended the suite by adding some versions implemented with FastFlow, SkePU2, and C++ Actor Framework (CAF). There is also an initiative towards translating the DSPBench benchmarks to WindFlow with C++[8]. Such efforts further highlight the need for C++ stream processing benchmarks.

In SPBench, we first added applications similar to those found in PARSEC to fill the major lack of benchmarks we found, which concerns traditional stream processing applications. However, the framework is not bounded by that. Recently we have added a data stream processing to show that the framework can be elastic and support applications from different SP subdomains. SPBench also provides the most relevant metrics for stream processing, as identified by [ABD+16] and [BGM+20]. In addition, it implements the benchmarks through an API that makes them more readable, easy to use, and highly parameterizable. It also comprises a framework that makes it easy and fast to port benchmarks to other PPIs. In this way, SPBench differs from other benchmarks by offering real-world SP benchmarks, including traditional stream processing, with representative metrics, and bringing a strong concern with usability to the table.

[8]https://github.com/ParaGroup/StreamBenchmarks

4.6 Chapter Summary

In this chapter, we first presented the currently supported applications by SPBench. It comprises 5 applications that users can use to create custom benchmarks. Some of these applications are from other benchmark suites, such as Ferret from PARSEC [BKSL08] and Fraud Detection from DSPBench [BGM+20]. The other ones have been used for benchmarking stream processing by different studies in this area. In SPBench, we have rewritten these applications into a modular and more readable form, adding performance metrics and parameterization options to fit them into the framework's API. SPBench also manages the downloading and installation of these applications' main library dependencies.

In the second part of this chapter (Section 4.3), we present the input workloads of the applications. In SPBench, these inputs are organized into workload classes. They range from small workloads for debugging purposes (test class) to larger loads for evaluating the scalability of benchmarks (huge class). SPBench also allows users to use custom input workloads and easily create new workload classes from them. As well as application dependencies, the SPBench interface also manages the download and installation of inputs.

Section 4.4 presented how parallelism is implemented in the SPBench benchmarks. The suite includes benchmarks implemented using everything from high-level abstraction PPIs, such as SPar and GrPPI, to PPIs that do not provide structured parallel patterns, such as C++ threads and OpenMP. Additionally, SPBench includes benchmarks using FastFlow, Intel TBB, and WindFlow. The benchmarks also explore different compositions of parallel patterns for stream processing, such as pipeline, farm, pipeline of farms, and farm of pipelines.

Since all benchmarks are built using a similar and very simple structure, with single generic examples, we were able to show how all benchmarks were parallelized in the SPBench. It shows its ability to implement a generic API that allows for high code reuse and rapid prototyping of benchmarks. This characteristic is further evidenced by the programmability evaluation of the PPIs in Section 5.7 of the next chapter. Therefore, practically every new application added to the SPBench in the future can easily be parallelized using the supported PPIs.

Finally, in Section 4.5, we present the related benchmark suites. We have identified that no other benchmark suite targets the evaluation of PPIs for C++ stream processing. Those that come closest to this, such as PARSEC and StreamIt, have many limitations. The most current and comprehensive benchmark suites target distributed platforms and JVM-based languages. Recent initiatives to improve the PARSEC benchmarks and to translate the DSPBench benchmarks into C++ show that there is demand for stream processing benchmarks in C++. The SPBench goes beyond providing a suite of benchmarks and also provides a framework that facilitates the creation and use of benchmarks in this context. In

the next chapter, we characterize the workloads and evaluate the benchmarks regarding performance, resource usage, and programmability/productivity.

5. PARALLELISM AND PERFORMANCE EVALUATION

Processing streaming data in real time is a challenging task in the field of computer science. To handle this kind of workload, parallel programming interfaces (PPIs) for stream processing are commonly used. Such PPIs may provide high-level abstractions to simplify the process of writing and executing parallel programs. However, with the proliferation of PPIs, it is not always clear which is the best choice for a particular task. Each PPI has its own strengths and weaknesses and choosing the wrong one can lead to suboptimal performance. Therefore, evaluating and comparing the performance of PPIs is important.

In this chapter, we evaluate the performance of the following PPIs: OpenMP, TBB, FastFlow, and C++ threads, SPar, GrPPI, and WindFlow. OpenMP is a widely used framework for shared-memory parallel programming. TBB is a task-based parallel programming library that provides a high-level interface for parallelism. FastFlow is a streaming parallel programming framework specializing in high-speed and low-latency parallelism. C++ threads provide a low-level interface for parallel programming. SPar is a C++11 domain-specific language for expressing stream parallelism through code annotations. GrPPI parallel library that adds an abstraction layer between developers and different PPIs, acting as a switch among the PPIs. For the last, WindFlow is a C++17 header-only library for data stream processing targeting shared-memory architectures.

The structure of this chapter is as follows. Section 5.1 further contextualizes the research problem we are addressing in this chapter. Section 5.2 discusses related work. We do a workload characterization of the SPBench applications in Section 5.4. This analysis is important to better understand the experimental results, not only of this chapter, but also of Chapters 6 and 7. In Section 5.5, we present the experimental results of the benchmarks with respect to throughput and latency. In Section 5.6, we address the memory usage of the PPIs. The programmability/productivity results are discussed in Section 5.7. Finally, in Section 5.9, we summarize the analysis done in this chapter and outline some conclusions.

Contents

5.1 CONTEXT **111**

5.2 RELATED WORK **112**

5.2.1 DISCUSSION OF RELATED WORK 116

5.3 EXPERIMENTAL SETUP **119**

5.4 WORKLOAD CHARACTERIZATION **120**

5.5 LATENCY AND THROUGHPUT PERFORMANCE **122**

5.5.1 EXPERIMENTAL METHODOLOGY 124

5.5.2 TBB, FASTFLOW, OPENMP, AND ISO C++ THREADS RESULTS 125

5.5.3 GRPPI RESULTS 128
5.5.4 COMPARING HANDWRITTEN FASTFLOW, SPAR-FASTFLOW, AND GRPPI-FASTFLOW 132
5.5.5 CUSTOM PARALLEL COMPOSITIONS RESULTS 136
5.5.6 DATA STREAM PERFORMANCE 139
5.6 MEMORY USAGE 142
5.7 PROGRAMMABILITY EVALUATION 145
5.8 OVERVIEW OF THE RESULTS 148
5.9 CHAPTER SUMMARY 150

5.1 Context

In Chapter 3, we discussed how this thesis addresses each of the research problems 1, 2, 3, and 4 that were introduced in Chapter 1. Here in this chapter, we address research problem 5, which is: **Performance analysis of PPIs for C++ stream processing is usually incomplete.**

Speedup or execution time are the state-of-the-art metrics to evaluate the performance of parallel applications. However, in stream processing, with potentially endless data streams, throughput (data processed per time unit) is a more representative metric than execution time since its computation is not always linked to the ending of the execution [vDVdP20]. Besides, SP applications can be tuned to achieve different performance goals, such as reduced latency, increased throughput, or efficient resource usage.

Throughput is commonly a more relevant performance metric for many traditional SP applications, such as data compression and a variety of multimedia processing. However, metrics such as latency are essential for stream processing applications that need to respond quickly to real-time data inputs [HJHF14]. Many applications have low-latency requirements, such as lane detection, object tracking, high-frequency trading, augmented reality, anomaly detection, online gaming servers, etc [DDMMT15, NXC19, LLG19, YLL+22]. In these cases, the desirable latency should often be less than a few milliseconds or even sub-milliseconds. Delays in processing can lead to traffic collisions, missed opportunities, poor user experience, or financial losses. In any case, achieving low latency in C++ stream processing applications typically requires careful attention to software design and performance optimization techniques, such as using specialized lock-free data structures and cache-efficient algorithms, minimizing memory allocations, reducing context switching, and leveraging parallel processing techniques [DDMMT15].

When it comes to resource usage, since these applications deal with possibly infinite data streams, memory usage can also be a critical factor to evaluate [TKPP20, MJP+19]. Also, the increased stream processing in embedded devices requires managing the device resources, especially memory and power consumption. These devices can benefit from programming languages like C++ because of their high performance and low-level memory management.

Therefore, properly evaluating the performance of stream processing applications and PPIs can be challenging since performance can be tuned to achieve multiple goals. In addition, it is also important to evaluate PPIs with regard to their usability since a slight performance gain may not be worth a significant increase in programming effort. This way, usability/programmability/productivity should not be disregarded when evaluating and comparing PPIs or other technologies in this context.

Regarding traditional SP, most works in the literature only address speedup, execution time, or throughput. There is a lack of comprehensive analysis comparing multiple PPIs with multiple performance metrics. Part of it may be related to the lack of stream processing benchmarks for C++, possibly aggravated by the difficulty of implementing or finding appropriate metrics. Properly implementing performance metrics in SP can be pretty complex, especially for beginners and in situations of unbounded streams, where metrics need to be computed by sampling.

We address this research problem in this chapter and partially in Chapter 6 and Chapter 7. We use SPBench to comprehensively analyze state-of-art PPIs that support stream processing in C++. In this chapter, we evaluate and compare the throughput and latency performance of Intel TBB, FastFlow, OpenMP, ISO C++ threads, SPar, GrPPI, and WindFlow. We also evaluate most of these PPIs regarding memory usage and programmability/productivity. In Chapter 6, we evaluate the impact of data stream frequency on FastFlow and TBB applications. Chapter 7 analyses how these two PPIs behave when using micro-batches on multi-cores. Most results and analyses here have already been published in relevant Journals and Conferences [GGSF22b, GGSF23, GGSF21, GGSF22a].

5.2 Related Work

In this chapter, we evaluate and compare the PPIs used to create the parallel SPBench benchmarks in different aspects. The PPIs are OpenMP, ISO C++ Threads, TBB, FastFlow, GrPPI, and SPar. Therefore, as related work, we consider studies that have also evaluated and compared the stream parallelism performance of these PPIs. We did not include studies that evaluated the performance of a single specific PPI, as our goal is to compare performance across multiple PPIs. Also, we did not include papers that only evaluate PPIs in a heterogeneous or distributed context since it is the scope of this thesis.

In this work, we also evaluate WindFlow's performance. However, since it was the last PPI added to SPBench and data stream support is in the early development stage in SPBench, we did not conduct a comprehensive analysis of it in this work. Nor did we compare it to any other data stream PPIs. We use it more as a proof of concept to show that SPBench can assess data stream applications. Thus, we are not including related work regarding WindFlow.

In [KHAL+14], the authors proposed a task-based runtime system called HPX. It extends the C++11/14 standard to facilitate distributed parallel processing in a global address space. They compared the HPX performance against the PPIs OpenMP, MPI, TBB, and QThreads. They evaluated these PPIs using two benchmarks (MiniGhost and N-Body) and evaluated Floating-point Operations Per Second (FLOPS) and throughput. However, only task parallelism is addressed since HPX is a task-based system.

The authors in [HLLL22] proposed Taskflow: a lightweight parallel and heterogeneous task graph computing system. They have evaluated and compared the performance, resource usage, and programmability of Taskflow against the OneTBB, OpenMP, HPX, and StarPU PPIs. Both micro-benchmarks and real applications were used in the experiments. Performance was evaluated regarding throughput, speedup, and execution time. Resource usage analysis addressed memory and CPU usage. To evaluate programmability/productivity, they hired 5 PhD-level C++ developers and measured the time they required to apply TaskFlow in the benchmarks. The amount of time spent implementing the real-world benchmark was about 3.9 hours for Taskflow, 6.1 hours for oneTBB, and 4.3 hours for StarPU. Programmability was also evaluated using code-based metrics, such as Lines of Code (LOC), Cyclomatic Complexity Number (CCN), and the number of tokens.

Pipeflow [CHGL22] is a task-parallel pipeline programming model for users to create a pipeline scheduling framework without data abstractions. It runs on top of TaskFlow [HLLL22]. The authors evaluated the throughput, execution time, and memory usage of Pipeflow vs. OneTBB, using two Computer-Aided Design (CAD) benchmarks. In their experiments, Pipeflow presented better performance than OneTBB. However, since Pipeflow is a task-based PPI it only addresses task parallelism.

In [LLS+15], authors proposed Cilk-P, a system that extends the Cilk parallel-programming model to support on-the-fly pipeline parallelism. They evaluated the performance of Cilk-P regarding speedup, execution time, and scalability and compared it with PThreads and TBB. The benchmarks used in the experiments were Ferret, Dedup, and x264, the streaming benchmarks from PARSEC. The stream parallelism was explored using compositions of farms and pipelines. Cilk-P performed better or equivalent to PThread and TBB in most evaluated cases.

MHPM (Multi-Scale Hybrid Programming Model) is an interface that targets easy expression of task, data, and pipeline parallelism in C++ [KLLDS12]. It is implemented as a C++ framework named XPU. In [KLLDS12], the authors evaluate its performance using a synthetic multimedia application and compare it against the PPIs OpenMP, TBB, and PThreads. They evaluated the execution time under different problem sizes and throughput under different parallelism degrees. Also, the authors compare LOC of TBB and XPU with a sequential benchmark, where results showed that TBB required 14x more LOC than XPU.

In [ATM09], authors aimed to show that FastFlow was more efficient than the other state-of-the-art PPIs. They compared FastFlow's speedup to TBB, OpenMP, and Cilk. These PPIs were used to implement a synthetic micro-benchmark and Smith-Waterman benchmark using a farm parallel pattern. FastFlow presented better performance using different computational grains in the tested scenarios.

In its Ph.D. thesis, Griebler developed a Domain-Specific Language (DSL) called SPar [Gri16b], which provides parallel abstractions for stream processing applications through the use of standard C++ code annotations. It basically generates FastFlow stream

parallel code using high-level abstractions. The author compared SPar performance against the PPIs TBB, OpenMP, and handwritten FastFlow. The experiments addressed execution time, throughput, speedup, latency, efficiency, energy consumption, and memory usage. The PPIs were used to implement benchmarks from Mandelbrot, Prime Numbers, K-Means, Sobel Filter + Gaussian Blur applications. However, these applications can be adapted to express stream parallelism, they are not traditionally from the stream processing domain. The author pointed out the lack of traditional stream processing benchmarks to better evaluate his work, which is one of the motivations for our work. The PPIs were also evaluated regarding LOC, where SPar presented the best results in all scenarios.

In [GHDF18a], the authors evaluated and compared the performance and programmability of the PPIs FastFlow, TBB, SPar, and PThreads. The experiments addressed execution time and speedup, and Dedup, Ferret, and Bzip2 were used as benchmarks. All benchmarks implemented the farm parallel pattern. SPar presented around 10% performance degradation w.r.t PThreads and FastFlow in the worst cases. Regarding programmability, SPar presented the lowest LOC and CCN.

The paper [GHDF18b] evaluates stream parallelism with ordered data constraints on multi-core systems. The authors proposed a new data-ordering technique for streaming applications that must process data in order. They apply their solution to FastFlow, SPar, and TBB and compare the throughput performance of these PPIs using benchmarks with ordering constraints, such as Lane Detection, Face Recognizer, Denoiser, Dedup, and Bzip2. The results showed that their solution could be integrated into the PPIs, introducing low overheads and opportunities to increase the application's throughput.

In [HGDF20], the authors extend the SPar DSL to generate TBB code. They compare the performance of their SPar-TBB solution to SPar-FastFlow, TBB, FastFlow, and PThreads. Similarly, in [HLGF22], the authors proposed OpenMP as a SPar backend. They compare the performance of their SPar-OpenMP solution to handwritten OpenMP and also to the same PPIs evaluated in [HGDF20]. For that, both works implemented benchmarks from the applications Lane Detection, Face Recognizer, Bzip2, and Ferret. In general, they observed that all SPar-generated code performed only 2.49% less when compared to the respective handwritten solution. The authors also compared programmability, where SPar, at the highest amount of LOC, introduced 9.85% extra LOC compared to sequential benchmarks.

RPL (Refactoring Pattern Language) is a DSL for designing and implementing parallel C++ applications [JBM+16]. It is a refactoring tool-support to offer semi-automatic parallelism via skeleton libraries. The authors apply their solution to the PPIs TBB, OpenMP, and FastFlow and compare their performance. The experiments involve Image Convolution and Ant Colony benchmarks implemented as a pipeline of farms. Results showed that RPL could be used to derive efficient parallelism in the tested scenarios.

In [dRADFG17], the authors propose GrPPI, a generic and reusable parallel pattern programming interface. It can run parallel applications with different backends, such as Intel TBB, ISO C++ threads, and OpenMP. They evaluated GrPPI-TBB, GrPPI-THR (ISO C++ threads), GrPPI-OMP (OpenMP) against a handwritten benchmark using these PPIs. Although the authors explore various parallelism compositions with pipeline, farm, and stencil, only one application was used as a benchmark in the experiments. The authors also did not investigate how performance scales for different parallelism degrees. They did use different problem sizes, however.

In [dDFG18], the authors extend GrPPI to support parallel patterns for data stream applications. They evaluated their work with applications from domains that typically require low latency, such as signal and sensor data processing. However, they used only speedup as a performance metric. Muñoz et al. [MnDdRA+18] added MPI as a GrPPI backend, allowing GrPPI applications to run on distributed platforms. They evaluated the speedup of their proposal against GrPPI-THR. The experiments varied the number of distributed nodes and the degree of parallelism within each node. In [LGFd+19], the authors extend the work from [MnDdRA+18] and compare their proposal against a natively implemented version using Boost-MPI. They also compare it against an implementation in Spark, a distributed data stream processing framework. GrPPI-MPI achieved significantly higher speedups than Spark, highlighting the performance benefits of C++ over JVM-based frameworks.

The authors from [GBdRAGC19] have implemented parallelism with GrPPI in a real-world MRI application. They exploited the pipeline-farm pattern, where GrPPI-THR and GrPPI-OMP achieved the best speedup results. They also evaluated memory usage, where TBB was the parallel backend of GrPPI that used the least memory and FastFlow used the most. However, no backend used less memory than the handwritten OpenMP application. In [GdRA+20], the authors have extended GrPPI to support FastFlow as a parallel backend framework. They tested their solution using four synthetic benchmarks for each parallel pattern ported from FastFlow. They evaluated the execution time of the GrPPI backends against implementations of the benchmarks using native FastFlow. They also evaluated programmability regarding lines of code (LOC) and McCabe's cyclomatic complexity number (CCN), where GrPPI presented similar results to handwritten FastFlow. Although FastFlow benchmarks achieved better or equivalent performance results on most of the parallel patterns evaluated, the GrPPI-TBB backend performed better with the Farm pattern.

[Víl20] evaluated the performance of GrPPI-THR and GrPPI-OMP using four benchmarks from the PARSEC suite. They compared performance against versions of the benchmarks originally implemented in Pthreads and OpenMP. In most cases, GrPPI showed performance equivalent to the original versions regarding execution time. In [BJB+20], the authors propose software refactoring techniques to semi-automatically introduce instances of GrPPI patterns into sequential C++ code. It supports pipeline and Farm parallelism, and they tested it with GrPPI-THR and GrPPI-OMP. Four benchmarks were used to compare the

speedup of their solution against manually written versions in ISO C++ threads. They were able to achieve performance equivalent to the baseline benchmark.

[AGSF23] assessed the programmability/productivity of PPIs for C++ stream processing on multicores. They asked a group of developers of distinct knowledge levels to implement stream parallelism with FastFlow, TBB, and SPar. Participants of the experiment have to implement it on an RGB Channel Selection benchmark. Although participants achieved the best speedups with FastFlow, the methodology used in the experiments favored it. To assess the programmability/productivity, the authors measured the development time. SPar demanded the lowest development time overall.

5.2.1 Discussion of Related Work

To contextualize our evaluations in this chapter, we reviewed related work that evaluated and compared the performance of PPIs for stream parallelism in C++. We focused on studies evaluating and comparing performance, resource usage, or programmability multiple PPIs rather than single specific ones. Table 5.1 summarizes and compares key attributes of related work.

Regarding PPIs, several studies evaluated and compared the same PPIs we target in this thesis. The motivation of most related work is to validate a new PPI solution by evaluating and comparing it against the state-of-the-art PPIs. Considering all the works we found, the most used PPIs to compare performance against were TBB (13 times), OpenMP (10 times), FastFlow (8 times), and PThreads (6 times). Other PPIs were used at most twice, such as MPI and ISO C++ Threads, or were the object under validation, such as SPar and GrPPI. High-level abstraction PPIs that allow users to run or generate parallel code to several lower-level backends are naturally more present in related work since every new backend they support requires performance validation. That is the case with SPar and GrPPI. In this work, we evaluate and compare the performance of the PPIs TBB, OpenMP, ISO C++ Threads, FastFlow, GrPPI-(FF, THR, OMP, TBB), and SPar-FF. Though SPar can generate TBB and OpenMP code, we found no publicly available version of it. We also evaluated WindFlow PPI, but it is a preliminary study and we do not compare it with other related PPIs.

The third column of Table 5.1 shows the parallel patterns used in related work. Although [KHAL+14, HLLL22, CHGL22, KLLDS12] target task parallelism, they also explore pipeline parallelism in some cases. Thus, we included these works. Most of the remaining related work evaluates parallel patterns commonly used in traditional stream processing, such as pipeline, farm, and compositions of pipelines and farms. The exception is [dDFG18], which proposes a GrPPI extension to data stream processing and evaluates stream operations from this domain.

Table 5.1: Summary table of related work regarding performance evaluation of PPIs that support stream processing in C++.

Rel. work	PPIs	Stream par. pattern	Performance metrics	Program. metrics	Benchmark applications
[KHAL+14]	OpenMP, HPX, MPI, TBB, QThreads	Task-pipeline	FLOPS, Throughput	–	MiniGhost, N-Body
[HLLL22]	OneTBB, HPX, TaskFlow, StarPU, OpenMP	Task-pipeline	Throughput, Speedup, Exec. time, Memory and CPU usage	Dev. time LOC, CCN, Num. tokens	Micro-benchmarks CAD application, Sparse Deep Neural Net.
[CHGL22]	Pipeflow, TBB	Task-pipeline	Throughput, Exec. time Mem. usage	–	Two CAD, benchmarks
[LLS+15]	Cilk-P, PThreads, TBB	Farm, Pipeline-farm	Speedup, Exec. time, Scalability	–	Ferret, Dedup, and x264
[KLLDS12]	XPU, OpenMP, TBB, PThreads	Task-pipeline	Exec. time Throughput	LOC	Synthetic multimedia application
[ATM09]	FastFlow, TBB, OpenMP, Cilk	Farm	Speedup	–	Micro-benchmark, Smith-Waterman
[Gri16b]	Spar-FF, TBB, FastFlow, OpenMP	Farm	Exec. time, Throughput, Speedup, Latency, Efficiency, Energy and Mem. usage	LOC	Mandelbrot, K-Means, Prime numbers, Sobel filter + Gaussian blur
[GHDF18a]	SPar-FF, TBB, FastFlow, PThreads	Farm	Exec. time, Speedup	LOC, CCN	Dedup, Ferret, Bzip2
[GHDF18b]	FastFlow, SPar-FF, TBB	Farm	Throughput	–	Lane Detection, Face Recognizer, Denoiser, Dedup, Bzip2
[HGDF20]	SPar-(TBB, FF), TBB, FastFLow, PThreads	Farm	Exec. time	LOC	Lane Detection, Ferret Face Recognizer, Bzip2
[HLGF22]	SPar-(OMP, TBB, FF), OpenMP, TBB, FastFLow, PThreads	Farm	Exec. time	LOC	Lane Detection, Face Recognizer, Bzip2, Ferret
[AGSF23]	SPar-FF, TBB, FastFlow	Farm	Speedup	Dev. time	RGB Channel Selection
[JBM+16]	TBB, OpenMP, FastFlow	Pipeline-farm	Speedup	–	Image Convolution, Ant Colony
[dRADFG17]	GrPPI-(TBB, THR, OMP), TBB, OpenMP, CUDA, ISO C++ Threads	Pipeline-farm	Throughput	–	Gaussian blur + Sobel filter benchmark
[dDFG18]	GrPPI-(TBB, THR, OMP)	Stream-Pool, Window-Farm, Stream-Iterator	Speedup	LOC, CCN	FM-Radio and 3 synthetic benchmarks: Traveling salesman, “Sensor” and “Image”
[MnDdRA+18]	GrPPI-(THR, MPI)	Pipeline-Farm	Speedup	LOC, CCN	Mandelbrot + Gaussian Blur
[LGFd+19]	GrPPI-(THR, MPI), Boost-MPI, Spark	Pipeline-farm	Speedup	–	Gaussian blur + Sobel filter, Mandelbrot + Gaussian Blur
[GBdRAGC19]	GrPPI-(TBB, OMP, THR, FF), OpenMP	Pipeline-farm	Exec. time Mem. usage, hardware metrics	–	pHARDI
[GdRA+20]	GrPPI-(TBB, OMP, THR, FF), FastFlow	Farm	Exec. time	LOC, CCN	Four synthetic benchmarks with simple math and vector operations
[Víl20]	GrPPI-(THR, OMP), PThreads, OpenMP	Pipeline-farm	Exec. time	LOC	From PARSEC: Swaptions, Blacksholes, Streamcluster, and Ferret
[BJB+20]	ISO C++ thr., GrPPI-(THR, TBB)	Pipeline-farm	Speedup	–	Ant colony optimization, Mandelbrot, Matrix multi., and Image convolution
This work	**GrPPI-(TBB, OMP, THR, FF), FastFlow, OpenMP, SPar-FF, TBB, ISO C++ Threads**	**Farm, Pipeline-farm, Farm-pipeline**	**Throughput, Latency, Memory usage**	**LOC, CCN, PHalstead**	**Lane Detection**, **Bzip2**, **Face Recognizer**, **and Ferret (PARSEC)**

With respect to performance metrics, the most used ones are execution time and speedup. These are two metrics commonly used to evaluate parallelism performance. However, stream processing applications can also be evaluated using a more representative metric like throughput. One of the characteristics of stream processing is that the end of the data stream is often undefined. Throughput can address such never-ending data stream scenarios.

Regarding hardware resource usage, memory was the most concerning attribute in related work. This is as expected since stream processing applications may use many buffers and receive an undefined amount of data to process in a continuous flow. We also evaluate the memory usage of the SPBench benchmarks in this work. [Gri16b] also evaluated energy consumption. Besides, it is the only related work we found that evaluates latency. The evaluated PPIs presented distinct latency behavior with different benchmarks in their experiments. In addition, their benchmarking workloads presented poor scalability. Also, most of them are not representative applications from the stream processing domain.

The programmability/productivity is often a secondary aspect when assessing PPIs in the literature. Most related work relied on code-based metrics to measure it, such as LOC, CCN, and the number of tokens of code. The PPIs that target providing high-level abstractions, such as SPar, GrPPI, and TaskFlow, presented the best programmability results. Since code-based metrics may present some inaccuracy [AGS+22], [HLLL22] and [AGSF23] also evaluated the PPIs concerning development time. The results helped to ensure that high-level abstractions can reduce the programming effort of parallel systems with a minimal performance penalty. In our work, in addition to LOC and CCN, we also evaluate the PPIs by applying the Halstead method. For that, we used a tool called PHaltead [AGS+22], which we further discuss in Section 5.7.

The last column of Table 5.1 includes the benchmark applications used in related work. Most benchmarks represent video and image processing applications. Around half of the studies used more realistic applications. Often the same or similar to the ones we use in our work. The other half uses micro-benchmarks and image filters to build synthetic benchmarking workloads or use applications commonly used for data parallelism, such as Mandelbrot, matrix multiplication, and prime numbers. Here, we evaluate the PPIs using the benchmark applications Lane Detection, Face Recognizer, Bzip2, and Ferret. We also evaluate, to a less extent, the WindFlow PPI using the Fraud Detection benchmark from SPBench.

Overall, most related works evaluated performance regarding execution time or speedup only. Only [Gri16b] evaluated PPIs' latency, an increasingly relevant metric for real-time processing, which is one of the goals of stream processing. Memory usage evaluations were also quite limited, not considering different strategies and degrees of parallelism or different stream processing applications. Regarding programmability, in addition to LOC and CCN, we use Halstead's method for parallel applications (PHalstead)

from [AGS+22]. We perform our experiments using four real-world application benchmarks, varying parallelism degrees, and different compositions of stream parallel patterns. We also use larger input workloads than the studies that evaluated PPIs using the same benchmark applications and run experiments on a newer and more robust multi-core architecture than most related work. Thus, our analyses differ from related work mainly by comparing the performance of a larger number of PPIs, using more representative metrics for SP, including latency analysis, using real-world benchmarks, comparing different parallelism patterns, and using a new metric to evaluate the usability/programmability of PPIs.

5.3 Experimental Setup

All experiments in this thesis were done on three different computers. In this section, we will describe the main architectural and system components of these computers. To simplify the identification of each computer in our discussions, we will refer to each one by its processor model. We used two computers with Intel processors and a computer with an AMD processor.

Table 5.2: Overview of the computer systems used in the experiments.

OPERATING SYSTEM INFORMATION			
OS	20.04.4 LTS	18.04.5 LTS	20.04.4 LTS
Kernel	Linux 5.4.0-105-generic	Linux 4.15.0-156-generic	Linux 5.4.0-109-generic
Architecture	x86-64	x86-64	x86-64
CPUs INFORMATION			
Model name	Intel® Xeon® Silver 4210	Intel® Xeon® CPU E5-2620 v3	AMD Ryzen 5 5600X
Clock Frequency	2.20GHz	2.40GHz	3.70GHz
Number physical cores	20 (2x10)	12 (2x6)	6
Number of threads	40 (2x20)	24 (2x12)	12
L1d cache	640 KiB	32K	192 KiB
L1i cache	640 KiB	32K	192 KiB
L2 cache	20 MiB	256K	3 MiB
L3 cache	27.5 MiB	15360K	32 MiB
Governor	Performance	Performance	Performance
MEMORY INFORMATION			
Total RAM	144 GB	32 GB	16 GB

Table 5.2 presents an overview of the computer we used in the experiments. The first Intel one has 144 GB of RAM and two Intel® Xeon® Silver 4210 CPU @ 2.20GHz processors (a total of 20 cores and 40 threads). The other Intel computer has two Intel® Xeon® E5-2620 v3 @ 2.40 GHz processors (total of 12 cores and 24 threads) and 32 GB of RAM. The AMD one has 16 GB of RAM and an AMD Ryzen™ 5 5600X CPU @ 3.70GHz (a total of 6 cores and 12 threads).

We used GCC 9.4.0 with `-O3` flag, and `performance` governor was enabled in all of them. The Xeon E5-2620 has Ubuntu 18.04.5 LTS operating system with Linux kernel

4.15.0-156-generic, and the others have Ubuntu 20.04.4 LTS, with Linux kernel 5.4.0-105-generic. The video processing benchmarks OpenCV 2.4.13.6. Handwritten TBB benchmarks and GrPPI-TBB used Intel TBB 2020 Update 2 (TBB_INTERFACE_VERSION 11102). SPar used FastFlow 3.0, while handwritten FastFlow benchmarks used version 3.0.1, the last available. GrPPI-FastFlow used FastFlow 2.2.0, the current version it supports.

5.4 Workload Characterization

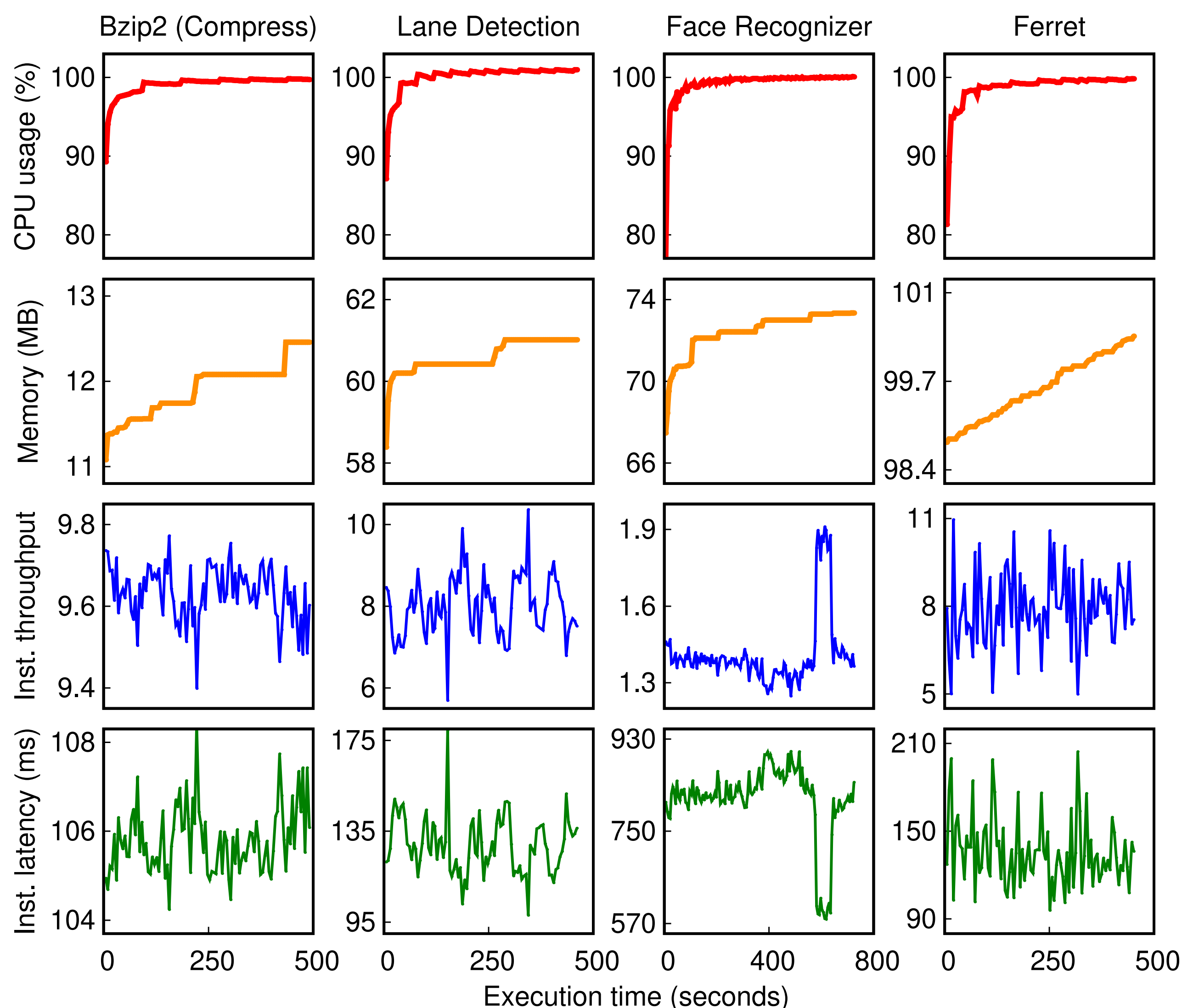


Figure 5.1: Characterization results (specific y scales for each application).

The SPBench workloads are quite diverse and allow the representation of different scenarios. In Figure 5.1 and Figure 5.2, we present performance results from sequential benchmarks of each SPBench application (except Fraud Detection). Both figures show the same results, but in Figure 5.1, we use individual y scales for each application to better visualize the workload behavior within itself. On the other hand, Figure 5.2 shows the

data using the same y scale for all applications. This helps visualize how the application workloads compare to each other. The metrics were obtained using the SPBench monitoring mode and five-second samples during the execution of the benchmarks. All experimental results can be obtained in SPBench by simply executing the following command once:

```
./spbench exec -ppi sequential -input huge -monitor 5000
```

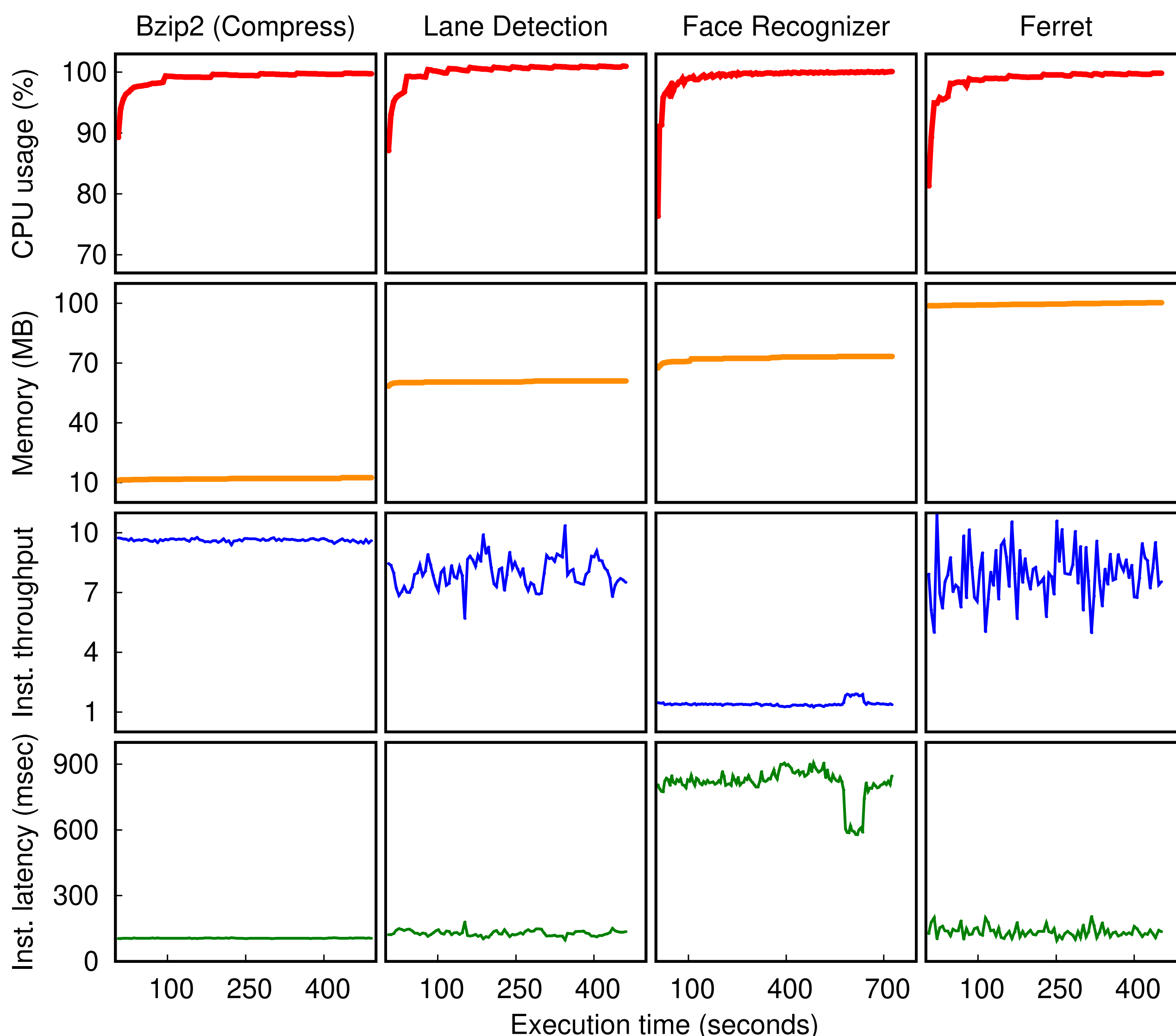


Figure 5.2: Characterization results (same y scales for all applications).

Source: [GGSF23] ©20XX Springer Nature.

The above command runs all sequential applications with the Huge workload class and monitors performance every 5000 milliseconds. Each column in Figure 5.1 and Figure 5.2 represents one of the applications. In the first row of graphs, we evaluate CPU usage. All benchmarks show high CPU usage, quickly reaching 100%, indicating great potential for improving performance through parallelism.

The second row of graphs in the figures presents memory usage. The benchmarks show relatively diverse behavior in this respect. Bzip2, for example, is the SPBench application that uses the least memory, and Ferret is the one that uses the most. Lane Detection and Face Recognizer have quite similar memory usage. Although there is an increase in memory consumption throughout the execution of these applications, this increase is relatively low.

The third metric is instantaneous throughput, i.e., average throughput measured over a short time interval. In these cases, where we are testing the sequential applications that take longer to execute, the time interval we used was 5 seconds (the same for latency). Although the Bzip2 throughput seems to present a high oscillation in Figure 5.1, when comparing it with the other applications, in Figure 5.2, we can see that the Bzip2 throughput varies the least. It varies less than 0.4 items per second throughout the execution, a steady workload. The one that varies the most is Ferret, with up to 6 items per second variation, similar behavior for Lane Detection.

The last row of graphs in both figures shows the instantaneous latency (5-second average). As expected, latency pretty much mirrors throughput. Latency spikes indicate regions of the input stream that demand more computation and vice versa. For example, in Lane Detection, when a car changes lanes or during intersections on the road, there will be more lanes to detect. In Face Recognizer, there are moments in the input where several faces are in the frame, creating latency spikes, and moments with no faces at all, which is the case of that big drop in latency in the graph. Therefore, the workloads used are pretty diverse and represent multiple scenarios.

5.5 Latency and Throughput Performance

In this section, we present performance results of TBB, FastFlow, OpenMP, ISO C++ Threads, GrPPI, SPar, and WindFlow regarding latency and throughput. We divide the experimental results discussion into different sections. In Section 5.5.2, we evaluate the performance of TBB, FastFlow, OpenMP, and ISO C++ Threads. We run the benchmarks on three different computers. Section 5.5.3 evaluates and compares GrPPI and its four backends against the handwritten implementations. Section 5.5.4 compares the performance of SPar and GrPPI with FastFlow backend against the handwritten benchmark using FastFlow. We test the Ferret benchmark using distinct combinations of parallelism degrees in a custom pipeline-farm implementation with FastFlow in Section 5.5.5. For last, in Section 5.5.6, we evaluate the performance of a data stream processing application (Fraud Detection) in SPBench using WindFlow.

Comparing PPIs performance is not easy since each may implement distinct mechanisms for concurrency, data communication, task scheduling, data ordering, etc. Taking as

an example the TBB's dynamic task scheduler based on the work-stealing model [VAR19], it differs significantly from the other PPIs. That is, in TBB, tasks are dynamically assigned to the threads. The parallelism degree we define in `tbb::task_scheduler_init init_parallel()` will be the maximum number of threads simultaneously running. On the other hand, in FastFlow, we define parallelism degree as the number of workers on the farms. It means that FastFlow single farm benchmarks create two extra threads to run Emitter (Source) and Collector (Sink) pipeline stages. The same is true for the other PPIs we use in SPBench, except GrPPI running with TBB backend and WindFlow.

```
-------------- AVERAGE LATENCY ---------------
      Operator Source        = 1.063938
      Operator Segmentation  = 2.748891
      Operator Extract       = 0.318422
      Operator Vectorization = 3.945016
      Operator Rank          = 10.126422
      Operator Sink          = 0.014719

     End-to-end latency (ms) = 18.222
----------------------------------------------
```

Figure 5.3: Example of SPBench latency results from the execution of Ferret (sequential).

Figure 5.3 shows part of the SPBench latency results from the execution of the sequential Ferret benchmark. SPBench can measure the average internal latency of the operators, i.e., how long data items take to be processed inside the operator. It helps to see the computational load in each operator and identify performance bottlenecks. In the SPBench benchmarks, the workload is unbalanced among the pipeline stages, as can be seen in Figure 5.3. Emitter and Collector stages often are less computing intensive than the farm workers. In the other PPIs, the threads running both stages are idle during most of the execution, while in TBB, idle threads can run any other ready-to-run task [VAR19]. Therefore, it is challenging to find ways to fairly compare PPIs' performance. In addition, SP applications can be tuned to reach different performance goals, such as throughput, latency, and efficient resource usage.

Regarding the pipeline of farms implementations, a fair performance comparison among these PPIs is even worse to achieve. In our benchmarks, TBB sees a single farm implementation the same way it sees a pipeline of farms. There is virtually no difference. That does not happen with the FastFlow-like static task-scheduler model used by the other PPIs. There is no way to run a Ferret with a six-stage parallel pipeline creating less than 6 threads in these cases. If this six-stage pipeline has four farms as middle stages and is unbalanced, as in Ferret's case, the ideal scenario would be to set different parallelism degrees to match the load of each stage. However, finding the best configurations by hand is a really complex task with many parallel stages. There are initiatives toward finding the

best configuration through self-adaptive parallelism [VGDF22]. However, few PPIs provide mechanisms that enable such technologies and using them is out of the scope of this work.

5.5.1 Experimental Methodology

In our experiments, we target tuning applications to balance three performance goals: throughput, latency, and efficient resource usage. For instance, by reducing the size of communication queues among the stages of a pipeline, the application can present reduced latency and lower memory usage. Although it can add some performance penalty on throughput, the impact is minimal on our test cases. We also try to use simple parallelism strategies and configurations that make a more fairly performance comparison of the PPIs. By enabling a blocking behavior, threads that otherwise would be in a busy waiting state can free up computational resources. Therefore, FastFlow and SPar, we enabled an `on-demand + blocking` configuration [ADKT17a], as recommended by [GHDF17]. In GrPPI, OpenMP, and ISO C++ Threads, we simulate an on-demand behavior by setting queue sizes to 1.

For the pipeline of farms experiments with the Ferret benchmarks, we choose an over-subscription methodology. Ferret implements a six-stage pipeline, with the four middle stages being a farm each one. We set the same parallelism degree for all the farms. It means that if we add 10 workers to each of the four farms of Ferret pipe-farm, the application will use 40 threads only to run those worker stages. Since Ferret stages are highly unbalanced, as can be seen in Figure 5.3, most threads would be idle most of the time. If running the application in a 40-core processor without any additional configuration, there would not be available resources to efficiently run 40 workers threads plus Emitters and Collectors. However, enabling a blocking behavior in the threads makes it use computing resources more efficiently and dynamically. Therefore, in a highly unbalanced pipeline of farms, we could run each farm with many more threads than the number of available processor cores and still potentially achieve performance improvement. We call this strategy “over-subscription” and use it to run the pipeline of farms implementations with FastFlow, SPar, and GrPPI in the next sections.

All performance results represent an average of at least five executions of the benchmarks. The standard deviation is presented as error bars in all the performance graphs. For these applications (except Fraud Detection), making an in-memory execution did not show any significant performance advantage. Therefore, we run the benchmarks reading data directly from disk, which is the default configuration in SPBench. Also, we did not use any kind of custom thread affinity mechanism.

5.5.2 TBB, FastFlow, OpenMP, and ISO C++ Threads Results

In this subsection, we compare the latency and throughput performance of TBB, FastFlow, OpenMP, and ISO C++ Threads. Figure 5.4 and 5.5 present throughput and latency results, respectively. Each column of charts in these figures represents experiments on a different computer where we ran the benchmarks. Each line of charts shows the results of a specific benchmark running on the three computers. The lines are the PPIs. The x-axis shows the maximum number of workers on the farm. Notice that the y-axis uses the same scale for the executions on the three computers, which makes it easier to compare the results.

Regarding throughput, in Figure 5.4, higher is better. The PPIs presented equivalent throughput performance in most cases when not using hyper-threading. However, all the benchmarks presented a reduced performance increase when using hyper-threading. In some cases, it also leads to a slight throughput decrease, as we can observe with the Face Recognizer benchmark in the AMD processor.

Although the AMD computer has half the number of cores of the Xeon E5-2620 v3, the AMD Ryzen 5 5600X is a newer processor model with a higher clock frequency. As such, it can deliver higher throughput than the Xeon E5-2620 computer. However, although the Xeon Silver 4210 has an even lower clock frequency (2.2 GHz), its high number of processing cores allows it to achieve a higher throughput performance.

Regarding latency, in Figure 5.5, lower is better. The benchmarks achieved the best latency results when running on the AMD computer, followed by the Xeon E5-2620 computer and then by the Xeon Silver computer. Here, the clock frequency of the processors again plays a major role since it can process individual data items faster.

TBB is the PPI that presents the best latency results in all our test cases. Its latency is about half of the others PPIs' latency. TBB uses a dynamic task scheduler with a work-stealing policy [VAR19] that benefits from the type of applications we use in these experiments. Each thread created in the other PPIs runs the same computation over different data items, i.e., they statically run a single stage of the pipeline. Items move from one stage to the next through queues, which adds extra waiting time, increasing latency. In TBB, a single thread can basically run a data item through the pipeline until it faces some barrier, such as an ordering constraint. However, this TBB model tends to reduce item clutter and items stay a shorter time in the buffers, presenting reduced average latency.

The throughput decrease and latency increase when increasing the parallelism of the Lane Detection benchmark with OpenMP and C++ Threads may be caused by different factors. The first factor, and probably the one with the most impact, is data contention caused by locking mechanisms. OpenMP and C++ Threads benchmarks use the same communication queues. The shared queue they use has a locking system to avoid data

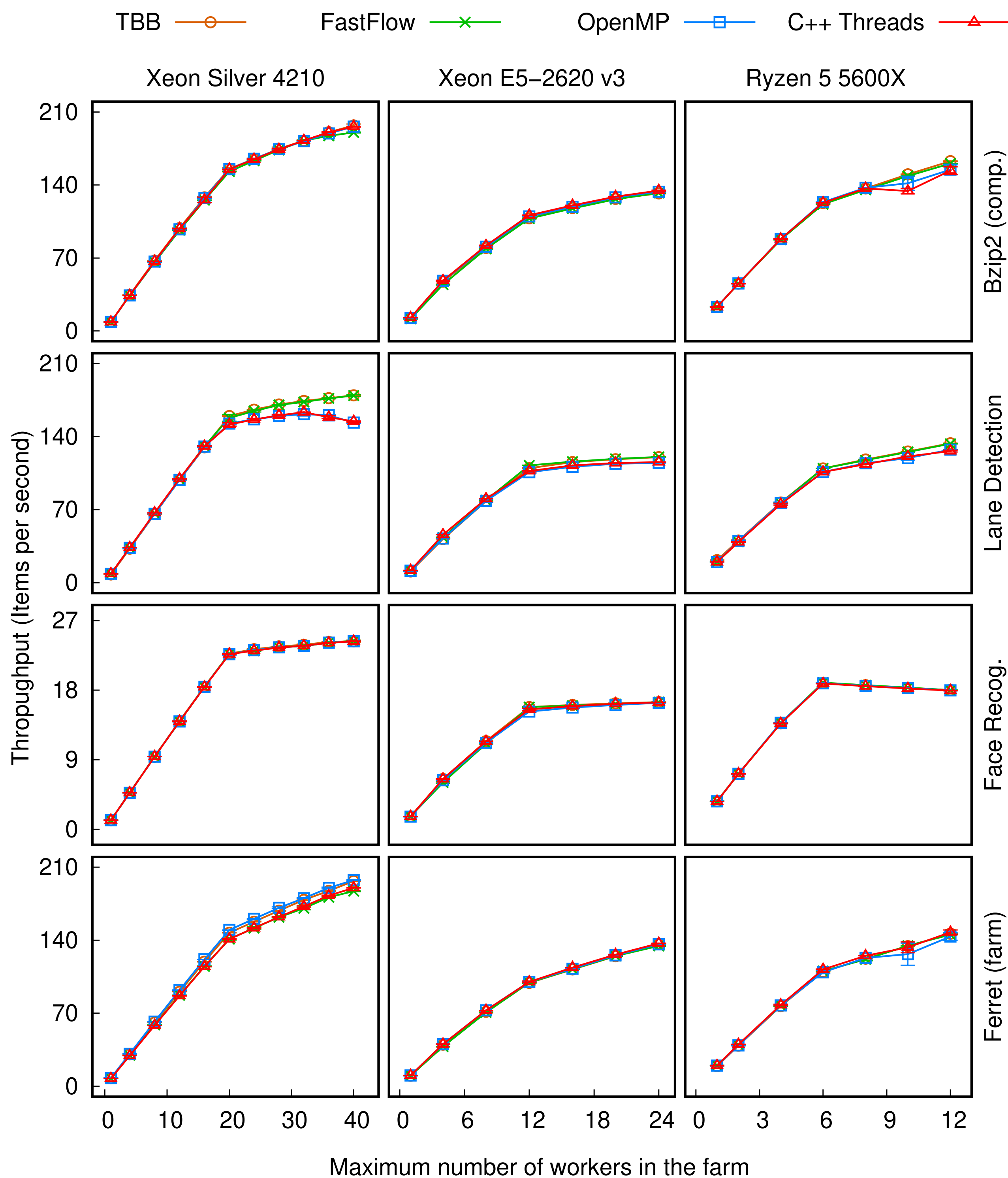


Figure 5.4: Throughput results of the TBB, FastFlow, OpenMP, and ISO C++ Threads Farm implementations in different computers.

racing. This way, when using a high number of parallel workers, the average time they have to wait to access the queues increases. Consequently, items take longer to be processed, increasing the latency and reducing throughput. FastFlow, on the other hand, uses lock-free queues. TBB also uses different mechanisms to synchronize access to the buffers.

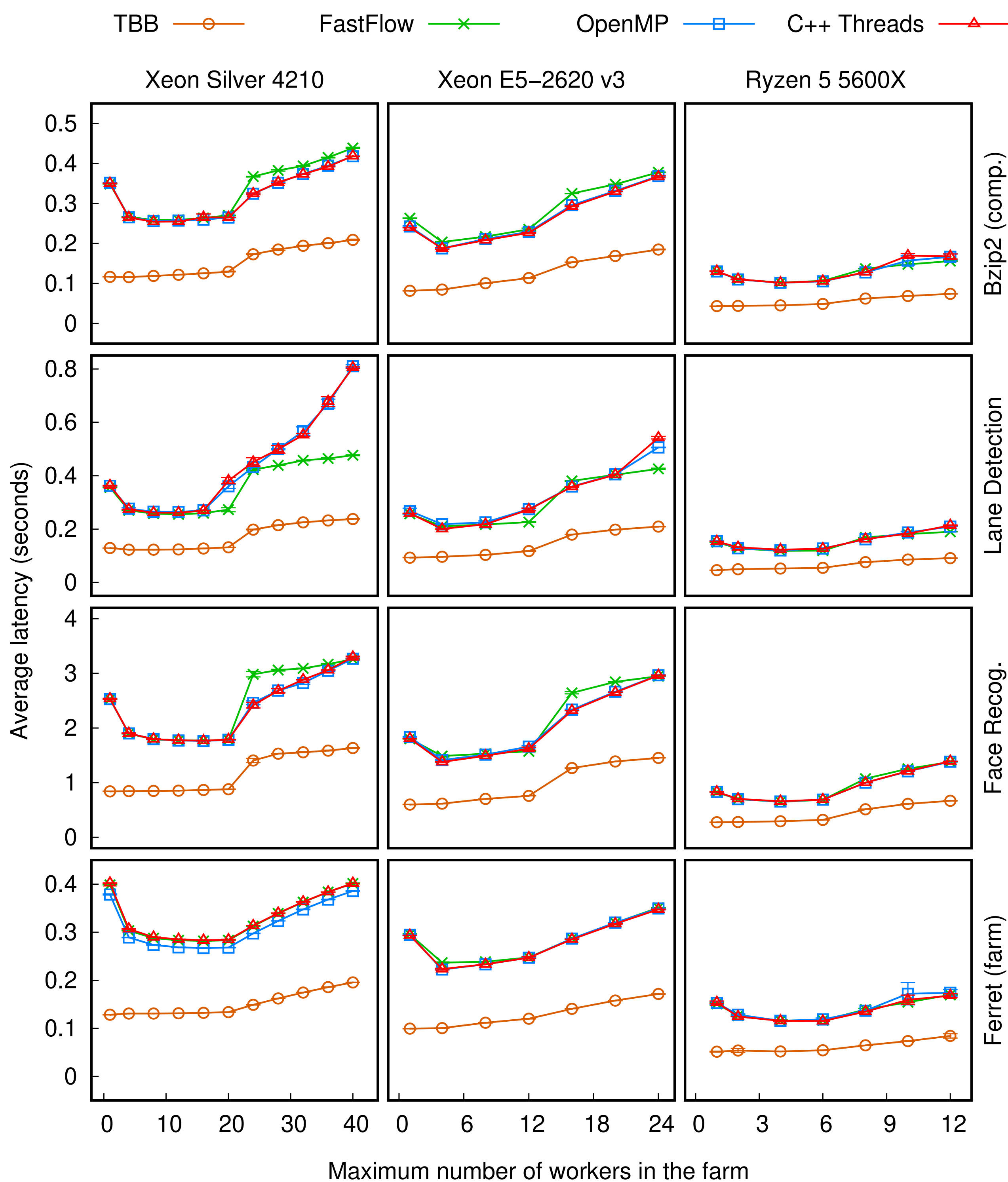


Figure 5.5: Latency results of the TBB, FastFlow, OpenMP, and ISO C++ Threads Farm implementations in different computers.

Another factor may be related to data ordering. OpenMP and C++ Threads also use the exact same ordering algorithm. It uses different structures and may not be as optimized as those used by FastFlow and TBB. in this case, the application characteristic is probably playing an important role in this respect. Lane Detection processes as many items per second as Ferret and Bzip2. However, while Ferret has no ordering constraints, Bzip2

has a balanced input load. Thus, Lane Detection put together the disadvantage imposed by data ordering requirements with an unbalanced load. This combination incurs highly disordered data items in the stream. This way, any slight difference in the implementation of the queuing and ordering mechanisms can lead to processing time overheads, impacting latency and throughput.

The SPBench workloads showed scalable performance up to more than 40 parallel threads. In all cases, the benchmarks present a reduction in throughput performance and increased latency when using hyper-threading, which is an expected behavior. These results will be used as a baseline to help evaluate the performance of the higher-level PPIs, such as GrPPI (Section 5.5.3) and SPar (Section 5.5.4).

5.5.3 GrPPI Results

GrPPI is PPI that provides structured parallel programming patterns for stream processing. It allows, from a single parallel implementation, to run the application with the backends OpenMP, TBB, FastFlow, and ISO C++ Threads. In this section, we evaluate GrPPI with all its backends and compare their performance against the handwritten parallel implementations presented in Section 5.5.2. Here, we run the experiments only in the Xeon Silver 4210 computer since the PPIs presented similar behavior across architectures in Section 5.5.2. Thus, we measure latency and throughput by varying the degree of parallelism in each farm stage from 1 to 40.

As discussed in Section 5.5.1, we enabled blocking mode on the PPIs as this allows more efficient use of resources and can improve performance when using hyper-threading, especially in applications that implement a pipeline farm [GGSF22b]. Figures 5.6, 5.7, 5.8, 5.9, and 5.10 present each application's latency (left chart) and throughput/items per second (right chart) results.

All four applications were implemented with a single farm. In the case of Ferret-farm, we unified the internal operators of the application into a single pipeline stage. However, one of the goals here is to check whether GrPPI is flexible enough to build different compositions for all the backends from a single generic code or not. This way, we also implemented Ferret benchmarks using a pipeline of farms and a farm of pipelines compositions. The pipeline of farms is the original parallel structure of Ferret in the PARSEC suite [BKSL08].

The x-axis of the graphs represents the maximum number of workers on each farm. It is fundamental to point out that the actual degree of parallelism of the farms and the number of threads varies according to how each PPI implements it. In fact, one may notice that GrPPI-OpenMP backend is not run up to 40 parallel workers but only up to 38 instead. This is a matter of internal implementation logic within GrPPI. In FastFlow, for

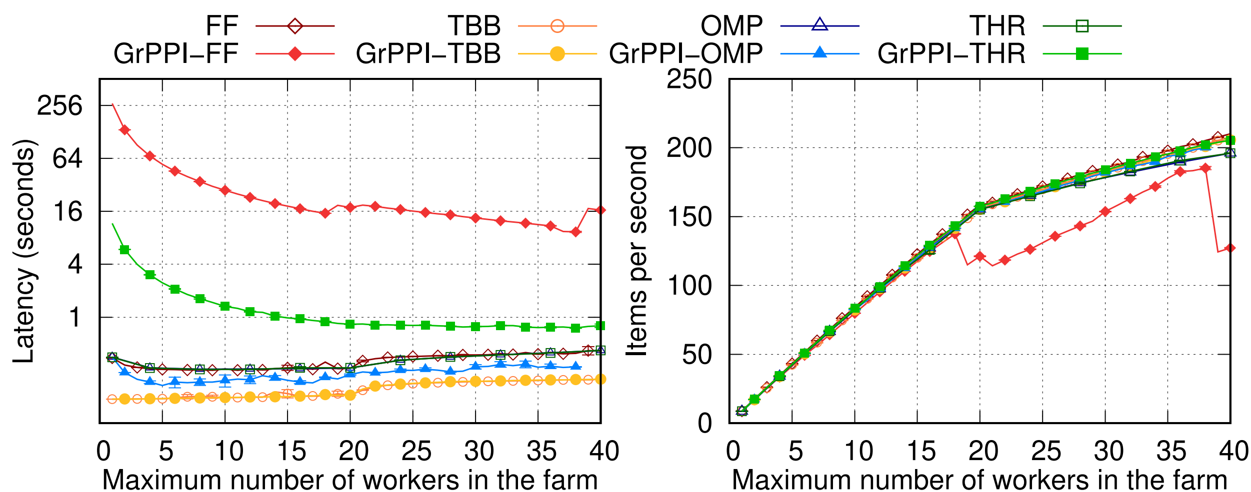


Figure 5.6: Latency and throughput of the Bzip2 (compress mode) benchmarks with different GrPPI backends and PPIs.

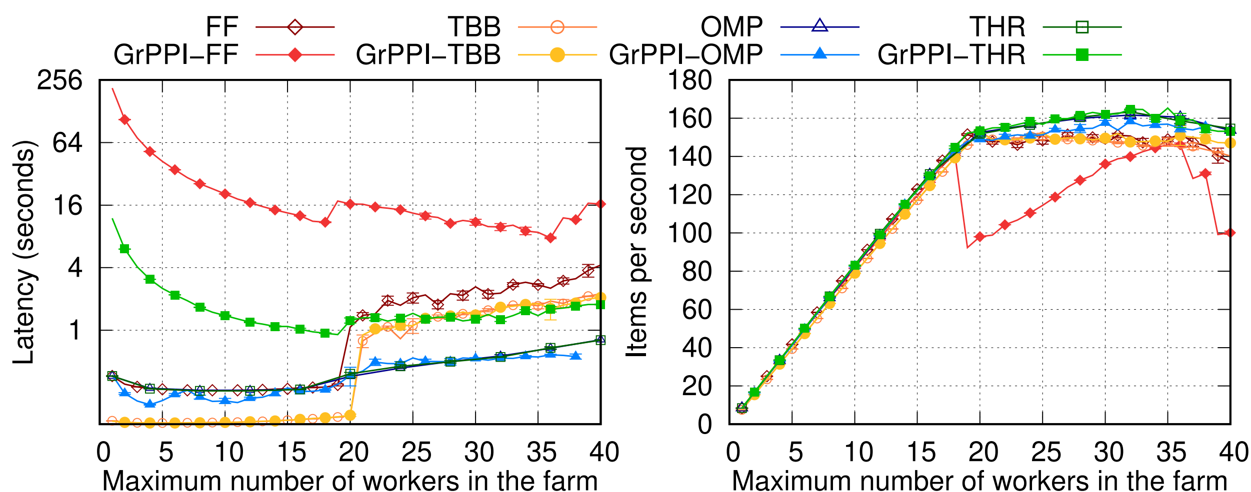


Figure 5.7: Latency and throughput of the Lane Detection benchmarks with different GrPPI backends and PPIs.

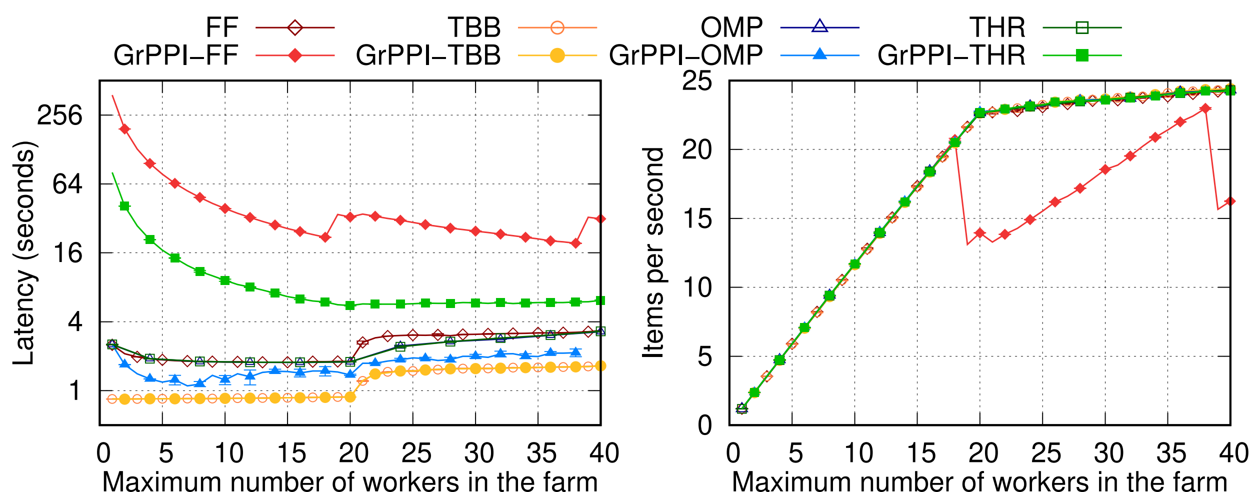


Figure 5.8: Latency and throughput of the Face Recognizer benchmarks with different GrPPI backends and PPIs.

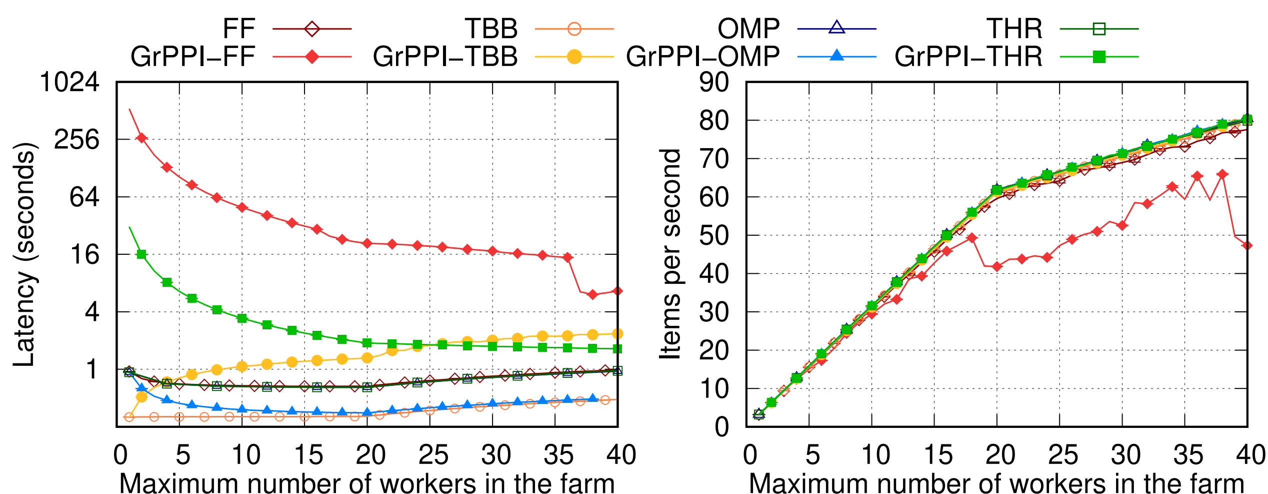


Figure 5.9: Latency and throughput of the Ferret (farm) benchmarks with different GrPPI backends and PPIs.

instance, if the user builds a single farm with two workers, it will run four threads. Two threads for the workers plus two threads to run the Emitter and Collector stages. It works the same way with GrPPI-FastFlow and GrPPI-Threads. For the OpenMP backend, however, GrPPI does not behave the same way. For the two-work farm in GrPPI-OpenMP, users must explicitly set to four the farm parallelism degree attribute. Therefore, in a single farm, for the same farm parallelism degree, GrPPI-OpenMP runs with two fewer workers than the FastFlow and C++ threads backends. This way, to make the results more comparable, we have shifted GrPPI-OpenMP results by two. It is not a concern with TBB, however, because of its work-stealing task scheduler that behaves entirely differently.

Overall, GrPPI's throughput with the TBB, OpenMP, and ISO C++ threads backends is equivalent to that of the benchmarks with handwritten code in the single farm benchmarks. In the Lane Detection application (Figure 5.7), GrPPI-THR and GrPPI-OMP even achieve better performance using hyper-threading. A similar behavior occurs in Figure 5.6 with the Bzip2 benchmark, where GrPPI backends (not including GrPPI-FF) perform better than handwritten OpenMP and C++ Threads. With Face Recognizer and Ferret (farm) benchmarks, in Figures 5.8 and 5.9, the throughput performance of GrPPI is equivalent or better than the handwritten benchmarks. In the case of GrPPI-FF, it achieves throughput comparable to the other PPIs with lower degrees of parallelism. Still, the inability to enable blocking mode in GrPPI's FastFlow knocks down performance when using hyper-threading above 20 workers.

Regarding latency, the performance of the PPIs varies widely among the applications. 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 for all applications.

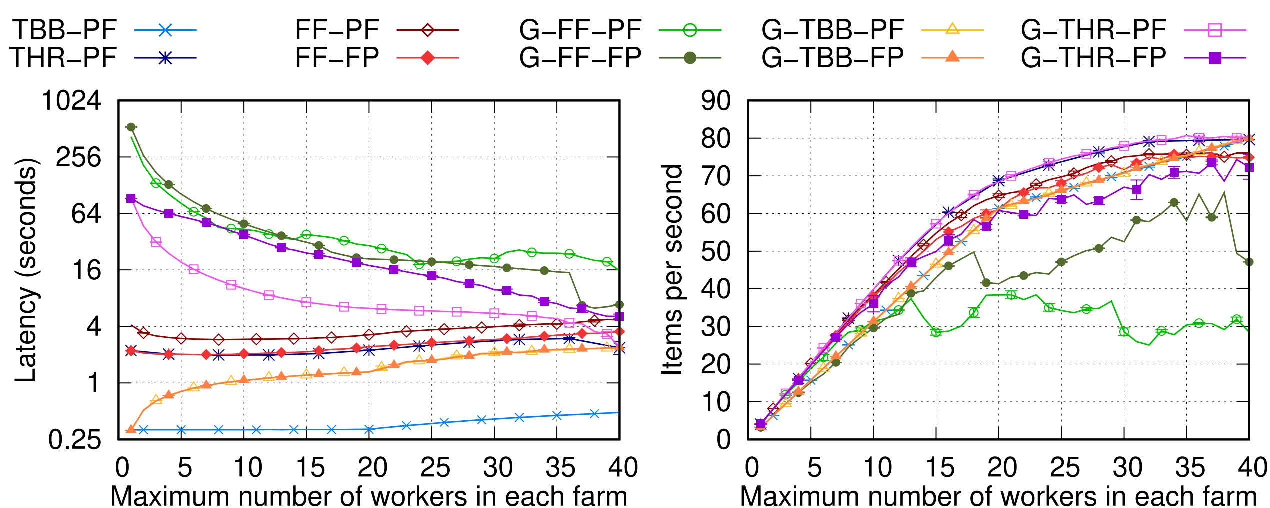


Figure 5.10: Ferret with compositions of pipelines and farms.

Although GrPPI-THR has the best throughput performance, it has the second-worst latency performance overall. However, it has similar latency to the TBB benchmarks in Lane Detection and Ferret when using hyper-threading. GrPPI-TBB has comparable latency to the handwritten TBB in all applications, except in the Ferret (Figure 5.9). In this case, it presents an increasing latency from the beginning, an unexpected behavior. Ferret differs from the other applications because it does not require item order, and how GrPPI implements it in TBB may explain this difference. GrPPI-OMP presented a similar and even better latency than handwritten OpenMP and C++ Threads in Lane Detection over 20 workers.

Figure 5.10 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 present in the graph. For this reason, we also did not include the results of handwritten OpenMP. In addition, after we ran the experiments, we noticed that we omitted the -03 compiler optimization flag when building this benchmark with all PPIs. Since all PPIs still are under the same parameters in this regard, it should not affect the performance comparison discussion.

The highest throughput was achieved by ISO C++ Threads (THR-PF) and GrPPI-THR-PF, at all degrees of parallelism, followed closely by the handwritten version of FastFlow. A pipeline of farms in TBB behaves very similarly to a single farm in an application with no ordering requirement (this case) since a TBB thread 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. This performance difference is due to the fact that a lower number of queues is required for a pipe of farms, where all works in a farm share a single queue. Consequently, the number of queues is independent of the farm multiplicity. So it requires fewer threads and has fewer shared queues, leading to better load balancing and mitigating the performance impact.

Except for the TBB benchmarks, all the PPIs considerably increased latency with pipeline-farm implementations. The difference between TBB and FastFlow in these situations has been extensively discussed in Section 5.5.2. 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. With GrPPI-THR, however, the pipeline of farms (PF) composition 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.

5.5.4 Comparing handwritten FastFlow, SPar-FastFlow, and GrPPI-FastFlow

In this subsection, we compare the performance of the handwritten FastFlow benchmarks against FastFlow-generated code from SPar and GrPPI with the FastFlow backend. The goal is to observe what impact high-level abstractions can have on the performance of PPIs and also what their limitations are with respect to tuning performance. Since the public release of SPar only supports the FastFlow runtime, we compare its performance only against the other FastFlow benchmarks in SPBench.

Figure 5.11 presents the results of the benchmarks that exploit a single farm as the parallel pattern. In most cases, SPar matches its performance perfectly with the handwritten FastFlow code, both regarding latency and throughput (average processed items per second). SPar's throughput doesn't show that drop above 18 workers, which is when the system starts using hyper-threading. This indicates that SPar is able to activate FastFlow's blocking mode. With the blocking mode active, idle threads free up computational resources for the others. This way, the system prioritizes the thread that runs the farm's Emitter and blocks idle workers. When Emitter and Collector need to compete with idle Workers, a bottleneck occurs and this results in the behavior observed in GrPPI-FF as soon the number of created threads reaches the number of physical processing cores of the computer (20).

In Lane Detection benchmarks, however, SPar performs worse than the handwritten FastFlow when using hyper-threading. The difference among the PPIs is more prominent regarding latency in this case. The FastFlow source code generated by SPar followed the

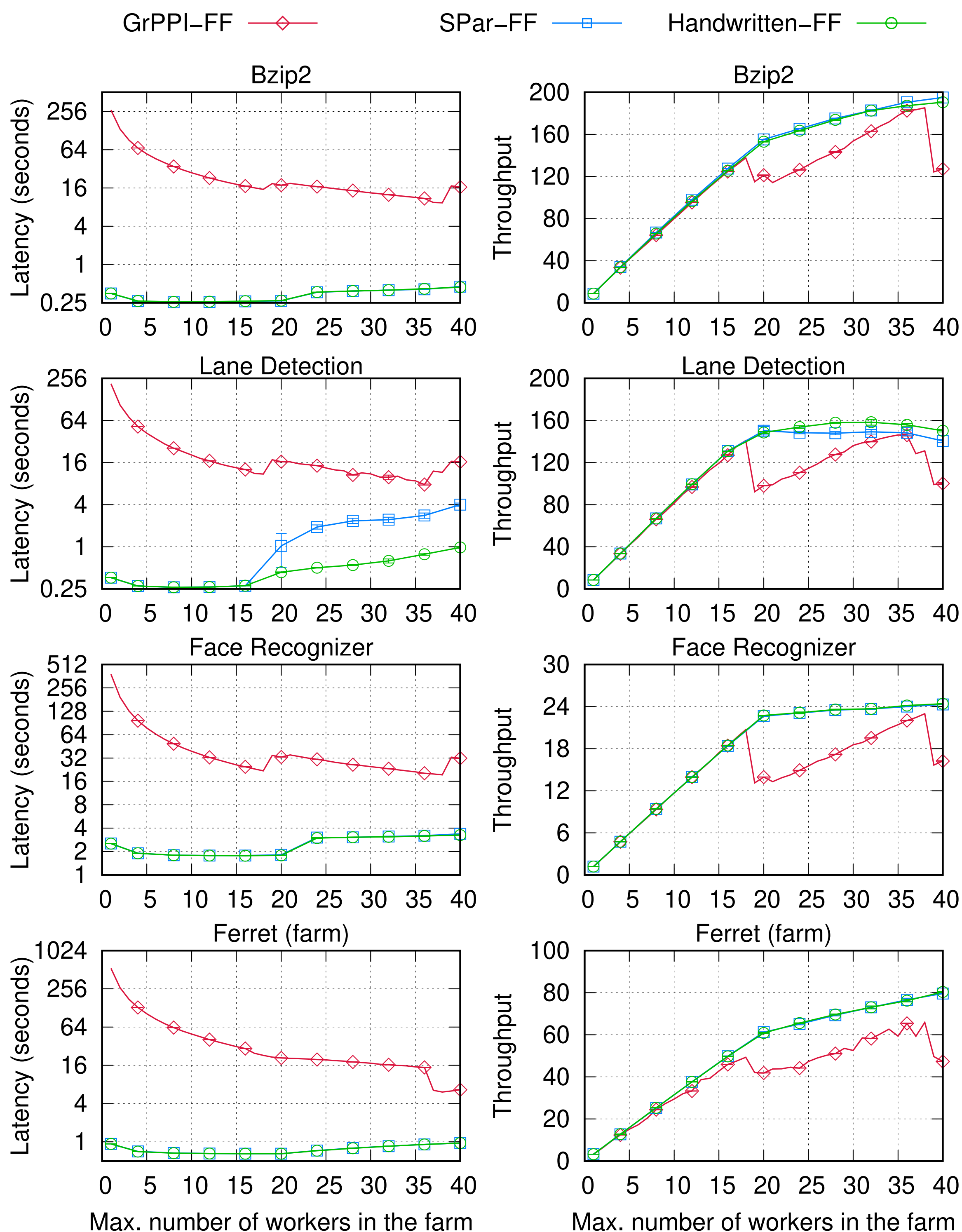


Figure 5.11: Performance of SPar-FastFlow, GrPPI-FastFlow, and handwritten FastFlow with single farm benchmarks.

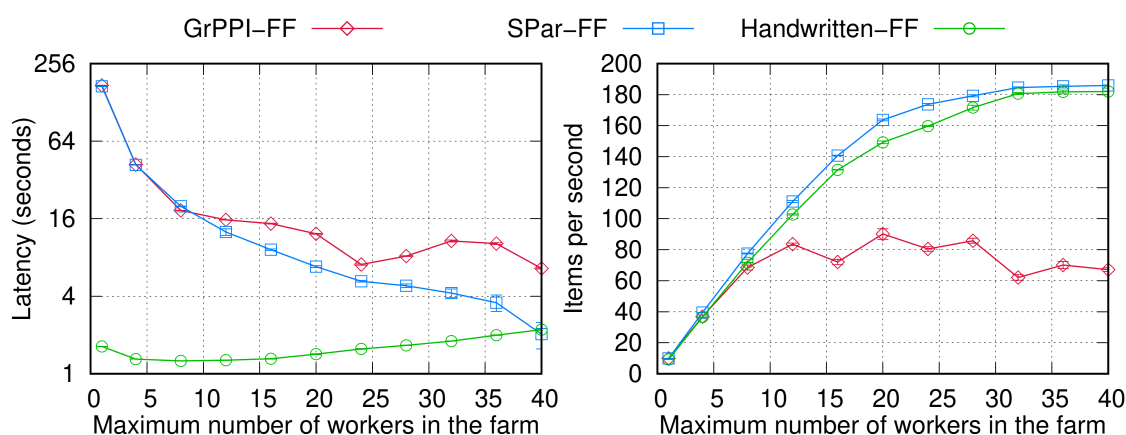


Figure 5.12: Performance of SPar-FastFlow, GrPPI-FastFlow, and handwritten FastFlow with a pipeline of farms benchmark.

exact structure of the handwritten FastFlow benchmark. Also, both use the same settings. However, while SPar uses FastFlow 3.0, the handwritten code uses the 3.0.1 version. Since FastFlow does not include any release notes on this, we are not sure if there are any improvements that may impact this specific case.

The results of the experiments with the pipeline of farms are presented in Figure 5.12. Here, as done in the GrPPI experiments in Section 5.5.3, we use the over-subscription methodology. That is, the pipeline of farms in Ferret has four farms. When 10 workers are assigned to each of them, the application will already be using more than 40 threads, which is higher than the number of threads of the architecture. Also, in a pipeline of farms, FastFlow adds several auxiliary stages between the farms. These stages will also run on dedicated threads.

For the over-subscription method to be worthwhile and the application to continue improving performance when all threads in the architecture are used, two main things should happen: 1) the application workload has to be highly unbalanced, and 2) threads should be in blocking mode. In FastFlow, it is also important to use mechanisms that reduce the number of auxiliary stages in the pipeline. This results in the farm pipeline illustrated in Figure 4.10, which is how handwritten FastFlow benchmarks are constructed in SPBench.

In the pipeline of farms benchmark, in Figure 5.12, the latency performance of SPar was more similar to the GrPPI-FF than to the handwritten FastFlow. Much different from what was observed in the single-farm results. We have a hypothesis about what is causing this behavior and it is related to SPar and FastFlow limitations. First of all, SPar is not generating an optimized FastFlow implementation. We extracted the FastFlow-generated code from SPar and observed that the code is not as optimized as our handwritten implementation. The pipe-farm Ferret-FastFlow handwritten benchmark available in SPBench required massive investigation, experimental analysis, and discussion to be implemented. It required the composition of multiple small and large pipelines and farms and manually

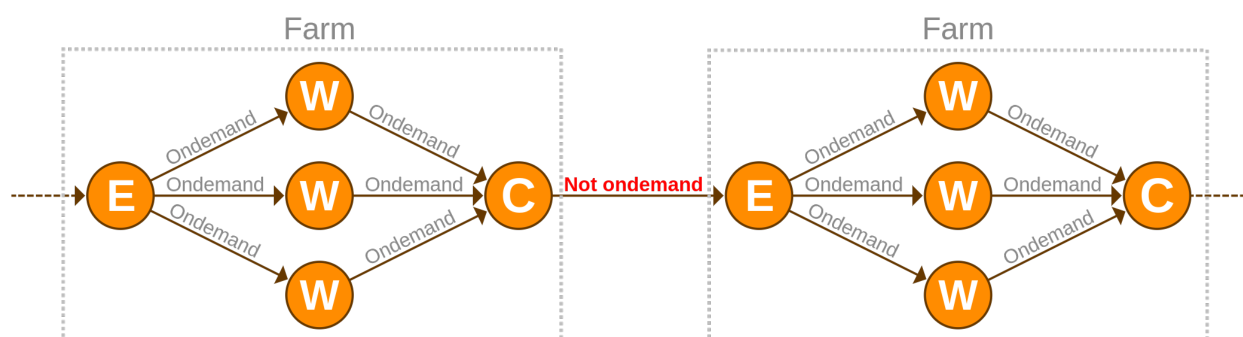


Figure 5.13: Usual on-demand behavior in a pipeline of farms in FastFlow.

removing unnecessary stages to get the structure presented in Figure 4.10. The Listing 11 shows an example of this implementation.

SPar generates a much more simple and more generic FastFlow code that has many additional stages. It would not be a problem for SPar if the `set_scheduling_ondemand()` method of the `ff::ff_Farm` FastFlow class worked adequately. This method enables an on-demand behavior only inside the farms but does not do the same to the inter-farm space, as shown in Figure 5.13. It means that between each pair of farms, there is at least one 512-sized queue, which is the default queue size in FastFlow. In the Ferret application, the most computationally costly operator is the Rank one (see Figure 5.3), which is at the end of the pipeline. Thus, in a pipeline of farms, the stages that precede the Rank process load all the data to these queues at a much faster rate than the Rank can consume them. Therefore, data items wait longer in queues in the first stages of the pipeline, increasing the average latency.

In our handwritten implementation, in addition to eliminating this inter-farm space, we compile the benchmark with the macro "`-DFF_BOUNDED_BUFFER`" to create bounded buffers and "`-DDEFAULT_BUFFER_CAPACITY=1`" to reduce their capacity size to one. These macros reduce the size of all FastFlow queues to 1 and simulate an on-demand behavior. Therefore, the SPar's inability to use such macros and generate such optimized code as the handwritten FastFlow implementation adds to the inability of FastFlow to apply a global on-demand policy through the `set_scheduling_ondemand()`, resulting in a poor latency performance of SPar-FF.

All the limitations that incur high latency do not affect the throughput, although. On the opposite, it shows how the strategy we used to reduce latency in the handwritten benchmark impacts throughput. With 20 workers per farm, the throughput of SPar is about 10% higher than the handwritten FastFlow. SPar also manages to achieve a lower latency when using 40 workers per farm.

In this over-subscription case, however, what most affects throughput is the inability to set a blocking policy for the FastFlow threads. SPar can enable the FastFlow block mode using the "-spar_blocking" compiler flag. However, this is still a limitation for

GrPPI-FF, as discussed in Section 5.5.3, leading to a throughput decrease when running more than 40 concurrent threads, which is the limit of the Xeon Silver 4210 computer. In this case, 8-worker farms already require over 40 threads because of the auxiliary stages created since GrPPI-FF also does not run an optimized FastFlow implementation. GrPPI throughput decreases precisely at this point.

5.5.5 Custom Parallel Compositions Results

In Subsection 5.5.3 and 5.5.4, we discussed the pipeline of farms' performance using the Ferret application. Experimental results, presented in Figure 5.12, showed that GrPPI and SPar struggle to achieve performance similar to the handwritten implementation in FastFlow. Besides, for FastFlow to achieve competitive performance with TBB in a pipeline of farms, it demanded a significantly higher programming effort. Ferret's unbalanced workload plays against static execution models like FastFlow's one. It is an even worse scenario for latency performance when computational lighter stages are at the beginning of the pipeline and heavier ones are at the end, as in Ferret's case. This makes date items pile up at the first nodes of the pipeline. Also, these stages process all the data first and may be in a busy waiting state after that, competing for computational resources that the heavier stages could use. To get around this, we apply the strategy of on-demand, so items don't pile up, plus blocking mode for idle threads to free up resources, plus an over-subscription strategy, so that threads that perform heavier tasks can actually use the maximum available resources. This strategy has been discussed at length previously throughout Section 5.5.

Another way to achieve more efficient hardware utilization and improve the performance of an unbalanced pipeline of farms is to assign a different number of workers to each farm. The more computationally costly a pipeline stage is, the more parallelism it gets. However, as the number of pipeline stages increases it gets more complex to find the optimal parallelism configuration. A possible approach in these cases could be to adapt the parallelism on the fly, and there is in the literature a research effort in this direction [VGS+18, VGDF22, VGF21]. Another approach could be to merge the pipeline stages. The ideal scenario is often to merge all intermediate stages into a single one. We already did that when evaluating Ferret's performance using a single farm in Section 5.5.2.

In this subsection, we will investigate an approach that consists in merging the first three lighter intermediate stages of Ferret, resulting in a pipeline with two middle farms, and applying different parallelism degrees to each farm. The goal is to achieve performance competitive with the previous strategies we used without using the over-subscription method. Also, we aim to investigate what the trade-offs are between latency and throughput with varying parallelism configurations. With this reduced number of farms,

the complexity of finding optimal parallel configurations is also reduced. This way, we used an approach of scaling the parallelism of the farms based on specific factors, manually changing each farm's parallel degree. This investigation only makes sense regarding PPIs that do not use a dynamic execution model. FastFlow is one of the most used PPIs in this context and provides structured parallel patterns as well as on-demand and blocking mechanisms. Therefore, we use only FastFlow for the experiments in this section.

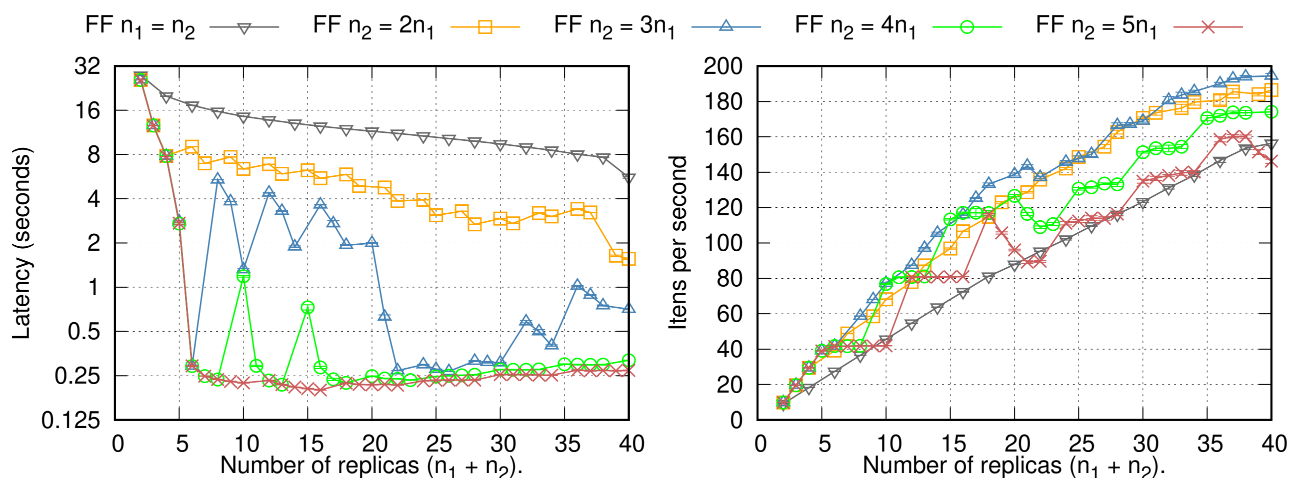
The Ferret application in PARSEC is originally parallelized using a similar pipeline of farms structure we used in SPBench. There, the four middle stages are thread pools that can run in parallel [BKSL08]. We can define this structure as `pipe(seq(source), Farm(seg, n), Farm(extract, n), Farm(vect, n), Farm(rank, n), seq(sink))`, where n is the number of workers in the farms. Here, we merge the farms that run the Seg., Extract., and Vect. operators, which, even combined, are less computing intensive than the Rank one, as shown in Figure 5.3. Thus, the resulting pipeline of farms can be represented as `pipe(seq(source), Farm((seg, extract, vect), n1), Farm(rank, n2), seq(sink))`. Building this custom pipeline of farms composition in SPBench is very simple and quick since it is just a matter of moving two lines of code from the operators into one of the farms and removing the code from two of the other farms.

Figure 5.14 shows the results of the custom pipe-farm benchmark. The x-axis in the graphs presents the sum of n_1 and n_2, which are the number the workers used in each Farm. We enabled the blocking mode and the on-demand policy in FastFlow. The experiments were performed on the Xeon Silver 4210 and Xeon E5-2620 v3 computers.

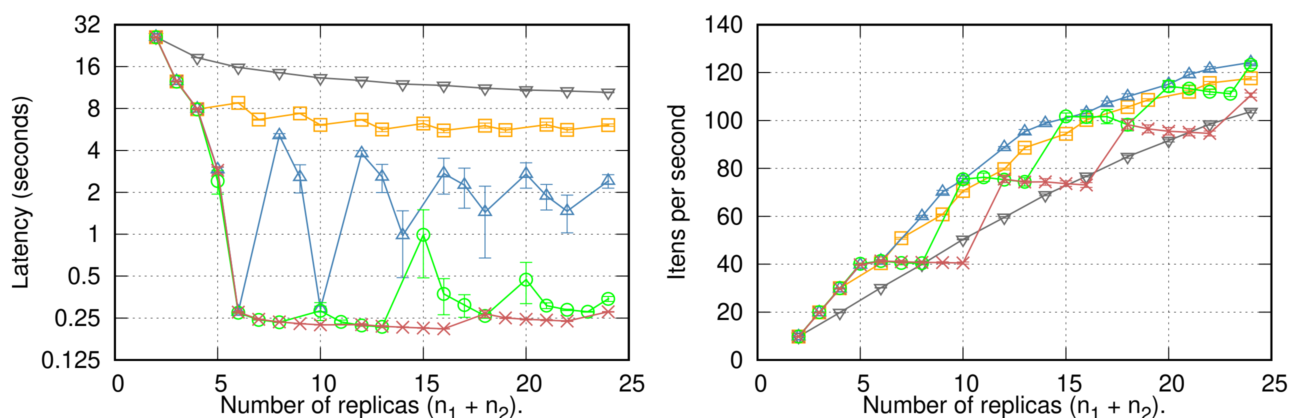
The custom pipeline-farm implementation consists of a four-stage pipeline with a farm in the second and a farm in the third stage. We implemented five variations, each representing a different ratio between n_1 (first Farm) and n_2 (second Farm): $1:1$, $1:2$, $1:3$, $1:4$, and $1:5$. For instance, for the $1:3$ ratio, when $n_1 + n_2$ equals 40, it means 10 replicas for the first farm plus 30 replicas for the second farm. Another example, considering the $1:2$ ratio, when $n_1 + n_2$ equals 24, it means 8 worker replicas in the first farm plus 16 workers in the second farm. We choose ratios only from $1:1$ to $1:5$ because for higher values of n_1, n_2 becomes a performance bottleneck, and vice-versa.

Although the results are somewhat different between the two tested computers, the conclusions we can draw from these results are the same for both scenarios. The $1:1$ parallel ratio presented the worst latency and throughput. Although $1:2$ and $1:3$ achieved good throughput, they did not achieve low latency. It means that for $1:1$, $1:2$, and $1:3$ ratios, n_2 value is still a bottleneck. in this case, the parallelism degree of the first farm is too high if compared to the second farm. So the first farm produces data faster than the second one can consume, which incurs items waiting longer in the queues, increasing the latency.

The $1:5$ ratio presented the lowest latency. However, with this ratio, the n_1 value becomes a bottleneck, where the first farm produces data slower than the second farm can



(a) Xeon Silver 4210



(b) Xeon E5-2620 v3

Figure 5.14: Latency and throughput for the custom `pipe-farm` Ferret benchmark. It defined as `pipe(seq(source), farm((seg,ext,vect),` n_1`), farm(rank,` n_2`), seq(sink))`, where n_1 and n_2 are the number of workers in the farms. Each line in the graphs represents a different ratio of the parallelism degree of the two farms. E.g., $n_2 = 3n_1$ means that every time n_2 is increased by 3, n_1 is increased by 1.

Source: [GGSF22b] ©20XX Springer Nature.

consume, decreasing the throughput. Therefore, we can observe that the 1 : 4 ratio is the most balanced for this custom Ferret implementation. This custom 1 : 4 ratio pipe-farm achieved a throughput close to the single farm and pipeline of farms implementations. However, the latency is about 10 times lower than the traditional four-farm pipeline and equivalent to the single-farm FastFlow implementation.

Therefore, we found a more balanced parallel configuration while still using a pipeline of farms and without over-subscribing tasks to the system. However, even reducing the number of parallel stages of the pipeline to achieve such a configuration was a burdening task. Finding a balanced configuration in the original Ferret's four-farm pipeline structure would require huge efforts. These experiments and results help to highlight

the importance of parallelism dynamicity in execution models and strategies that aim to alleviate this task from users, such as self-adaptive parallelism.

5.5.6 Data Stream Performance

Fraud Detection (FD) was the latest application added to the SPBench. For this reason, we have not yet investigated it in any other work prior to this thesis. So it still requires further analysis and polishing to achieve a more optimal implementation. However, we still decided to include an analysis of this application in this thesis because it can show that SPBench can be flexible to incorporate applications from other SP subdomains besides traditional stream processing, such as data stream processing. In Section 4.2.5, we discussed the challenges that FD posed to the SPBench and how we addressed them. However, an analysis of this application can help in another aspect: showing the performance impact after reprogramming the application to fit the SPBench API.

As discussed in Section 2.2.2, data stream applications usually process smaller data items than traditional stream processing applications. Classically, data stream applications process streams of string tuples, while traditional SP processes unstructured data, such as video frames, images, large blocks of bytes, etc. Each FD tuple totals 27 bytes in size, for instance, while this value is 900KB in Bzip2. Therefore, the time required to process these small tuplets is usually much shorter. So these data stream applications like Fraud Detection are capable of processing more than a million tuples per second. This is throughput thousands of times higher than we achieve with the other applications in the SPBench. This much faster throughput of these applications means that small overheads in processing a single item have a big impact on the average performance of the application.

The SPBench Fraud Detection application was built based on an existing implementation with WindFlow [MTG+19]. It is part of StreamBenchmarks[1], an extended version from the DSPBench benchmark suite [BGM+20] that includes WindFlow benchmarks. We first wrote a sequential version of FD-WindFlow and, from that, translated the sequential application code to SPBench style, as presented in Listing 9. After that, we used WindFlow to create a Fraud Detection benchmark in SPBench. We based our parallel implementation on the original FD-WindFlow available in the StreamBenchmarks repository.

In this section, we run Fraud Detection with WindFlow from SPBench and compare the performance with the original implementation, which also uses WindFlow PPI. The goal is to evaluate the impact of the SPBench API on the performance of this application and how the performance compares to a pure implementation with WindFlow. This application implements a three-stage pipeline, as illustrated in Figure 4.8. However, since the middle stage is not very computationally expensive, it processes items very fast, so as parallelism

[1] https://github.com/ParaGroup/StreamBenchmarks

increases, the Source and Sink operators become performance bottlenecks. Thus, achieving higher throughputs involves a balance of parallelism among the three stages. Therefore, we perform experiments varying the parallelism of the three stages.

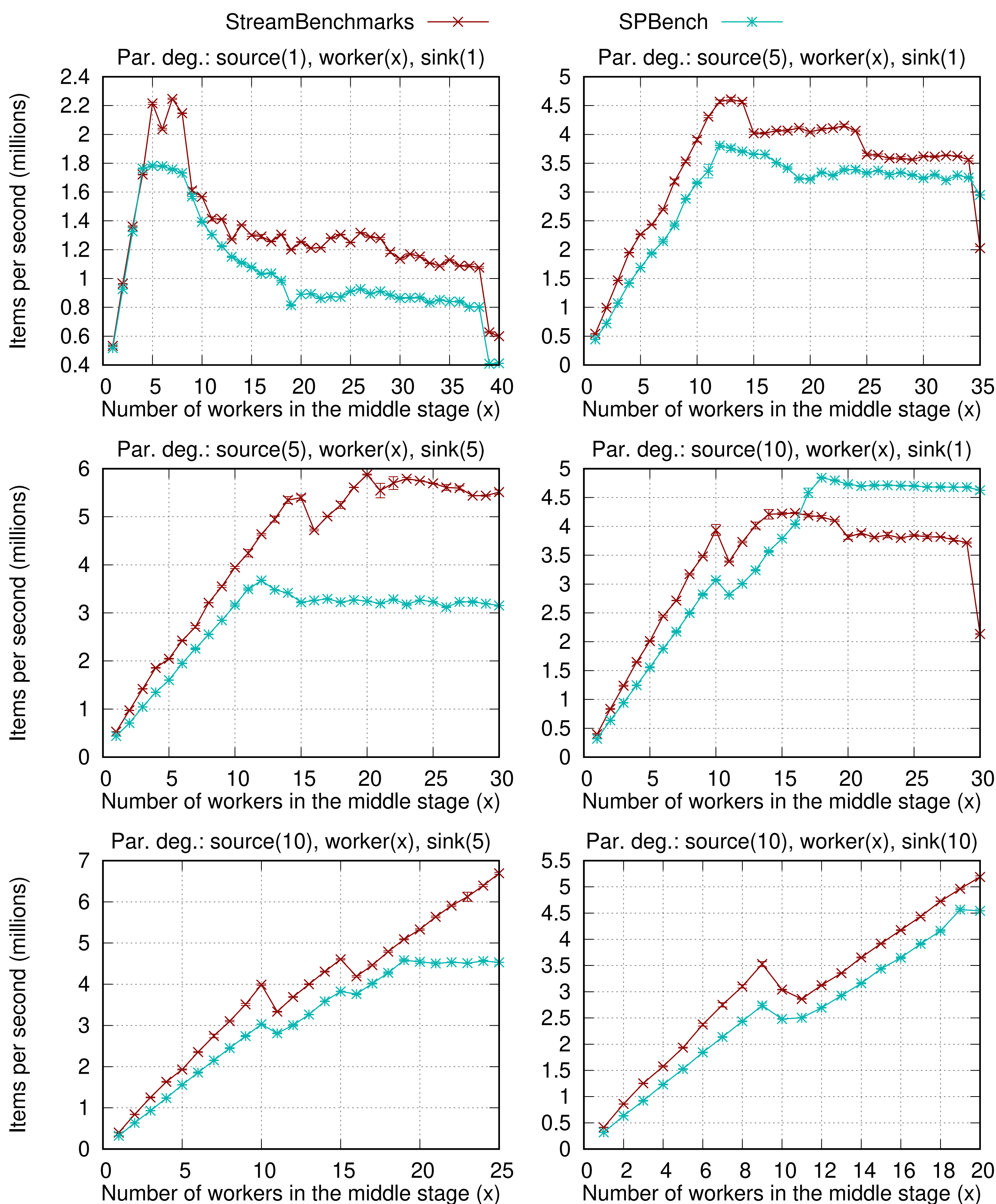


Figure 5.15: Fraud detection performance results.

Figure 5.15 shows the performance of the Fraud Detection benchmark using WindFlow. Each of the six graphs represents a different parallelism strategy we used for the experiments. In each strategy, we use a static parallelism degree for the Source and Sink operators and vary the parallelism degree of the middle stage. These strategies are described at the top of each graph, where the *x* value from “worker(*x*)” is the x-axis value. Thus, the parallelism degree of the workers is the only attribute that varies in each graph. The middle stage of Fraud Detection implements a filter operator. Only transactions that are potentially fraudulent go through it and are received by the Sink operator. This way, the Sink processes fewer items than the Source, and, in addition, it is less computationally intensive. Therefore, in all of our test cases, the parallelism of the Source operator is equal to or higher than the Sink one. In addition, as described in Section 4.2.5, in this application, we enabled the SPBench custom user item counter in the Source operator, which implies that throughput is computed over the items sent by the sources rather than received by the sink.

The graphs in Figure 5.15 show the throughput performance in millions of transactions processed per second. We compare the throughput of the StreamBenchmarks-FD benchmark against the SPBench-FD benchmark. We can notice that the first strategy (1,x,1) resulted in the lowest performance. It is a consequence of Source and Sink bottlenecks. The biggest performance difference between StreamBenchmarks and SPBench is with the (5,x,5) strategy, where StreamBenchmarks-FD almost doubled SPBench-FD performance with more threads. On the other hand, in the (10,x,1) strategy, SPBench achieved higher performance results than StreamBenchmarks. Our hypothesis, based on these two results, is that the SPBench Source operator is heavier than and the Sink is a lighter task than the ones in StreamBenchmarks. Therefore, SPBench benefits from more Source parallelism and require less Sink parallelism, while Streambenchmarks-FD benefits from more balanced configurations. The results of the strategies (10,x,5) and (10,x,10) help to confirm this hypothesis. StreamBenchmarks-FD achieved the highest throughput with (10,25,5) parallelism, increasing it almost linearly. SPBench, on the other hand, only presents such a linear throughput increase when up to 20 parallel workers are used in the middle stage, plus higher source parallelism.

The last strategy (10,x,10) was the one where SPBench-FD and StreamBenchmarks-FD performed the most similarly. In this case, SPBench throughput performance is around 0.5 million lower than StreamBenchmarks for higher values of *x*. In this strategy, the parallelism degree used for Source and Sink eliminates possible bottlenecks of these operators. Thus, it better shows how the original WindFlow Fraud Detection benchmark outperforms the SPBench benchmark.

It is expected for SPBench applications to present some performance overhead since it includes several additional workload management mechanisms, abstraction layers, and performance metrics. All of it adds many more conditional structures, memory

allocations, and extra instructions that can impact performance. However, as previously discussed, Fraud Detection was the last application added, and only recently, to the SPBench suite. Its structure is a lot different from the traditional stream processing applications. Also, it did not undergo extensive evaluation and polishing as the other SPBench benchmarks. Therefore, improved implementation regarding the application and parallelism exploration may reduce the performance overhead in the future.

The experiments and results presented in this section showed that we successfully incorporated Fraud Detection, a data stream application, into the SPBench suite. For that, in SPBench, we had to implement support for Source and Sink parallelism, allow key-by data partitioning, and handle stateful and filter operators. Also, we successfully added support for creating WindFlow benchmarks with SPBench. Besides the SPBench framework infrastructure, another thing that differentiates SPBench from Streambenchmarks is that we offer sequential implementations of the applications, which increases the portability of the benchmarks to other PPIs.

5.6 Memory Usage

The evaluation of memory usage is a crucial aspect of developing and optimizing stream processing applications. To meet real-time processing demands, SP applications typically require handling large amounts of data and running tasks concurrently in parallel to achieve high throughput and lower latency. By evaluating memory consumption, developers can identify potential bottlenecks and optimize their code for better performance. It is also a relevant factor when it comes to the scalability of stream processing applications. As the amount of data being processed increases, so does the memory required to handle that data. By evaluating memory consumption, developers can ensure that their code can scale effectively as the size of the input data increases.

In this section, we use the SPBench benchmarks to evaluate the memory usage of TBB, FastFlow, OpenMP, ISO C++ threads, SPar, and GrPPI in the context of stream parallelism. 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 results of the benchmarks that implement a single farm are shown in Figure 5.16. Note that the y-axis is in a logarithmic scale for better data visualization. In most cases, PPIs demanded similar amounts of memory in each application. The apparent exception was GrPPI-FF, 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. Another contribution to this effect is derived from the strategy

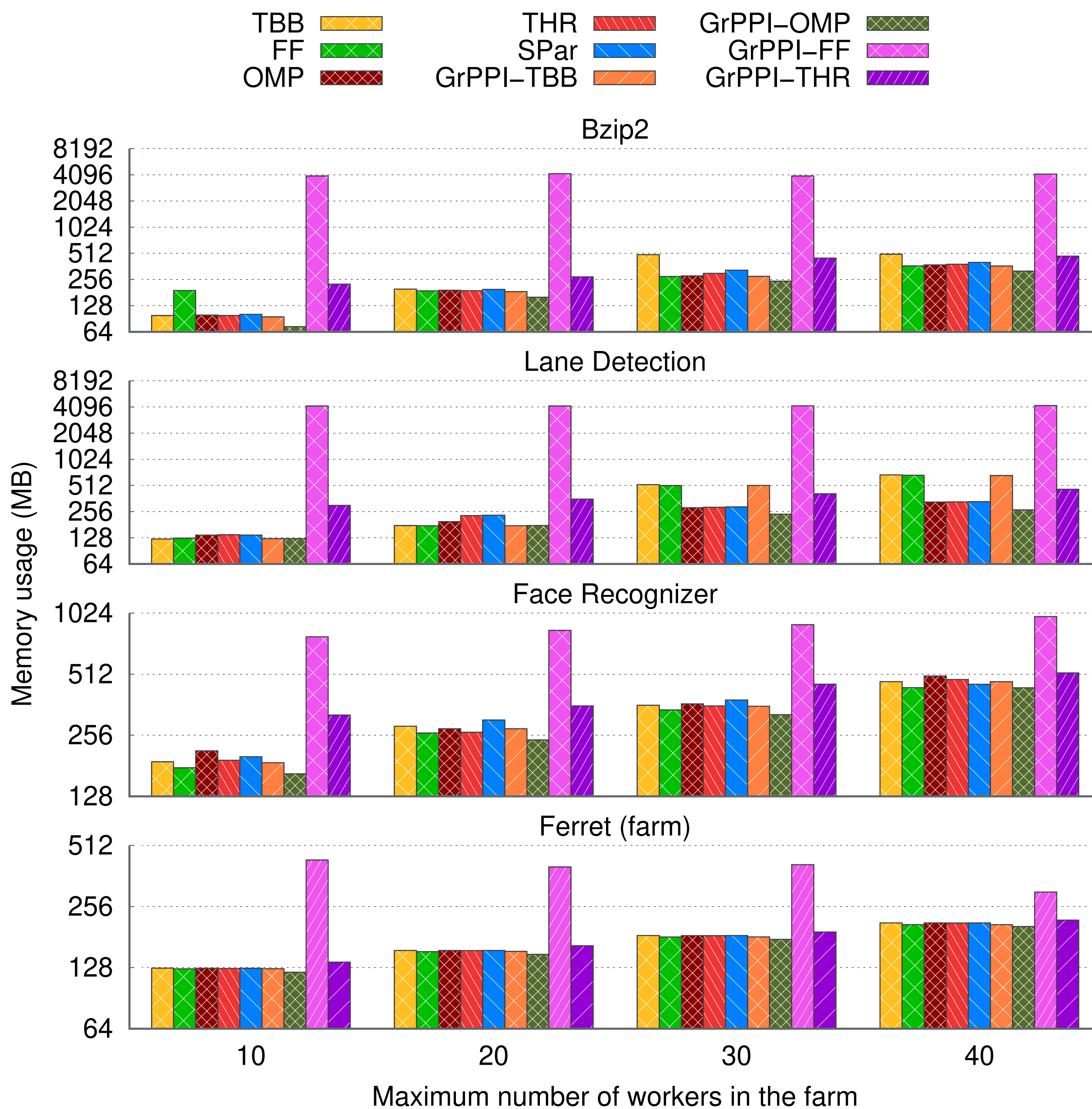


Figure 5.16: Total memory consumption of benchmarks with a single farm.

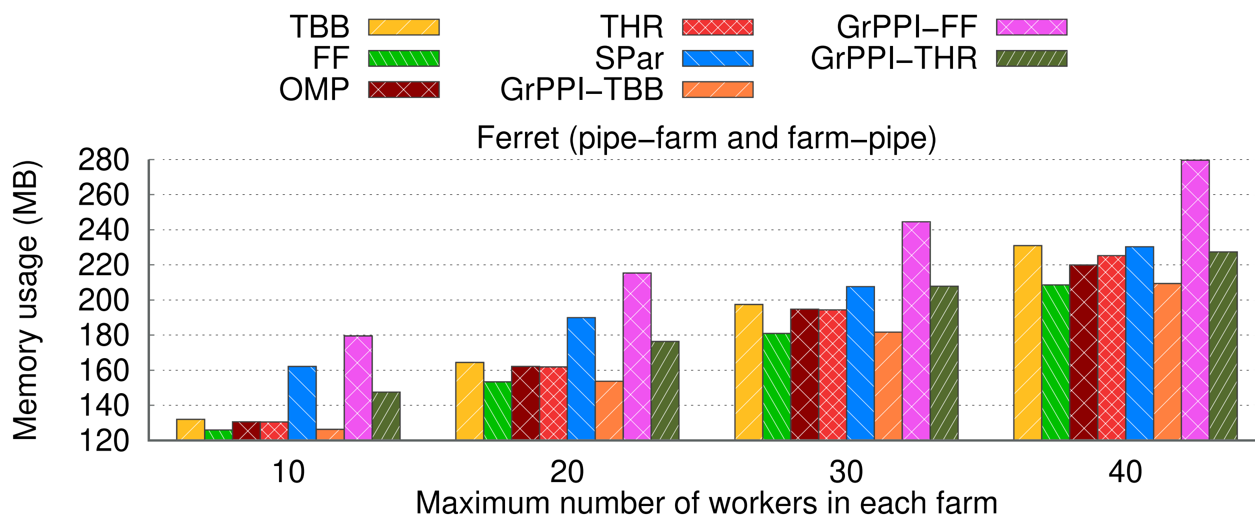


Figure 5.17: Total memory consumption of Ferret benchmarks using pipeline-farm (PF) and farm-pipeline (FP) compositions.

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.

GrPPI-THR (ISO C++ threads backend) was the second case that most used memory in all applications except Lane Detection. The difference from the other PPIs is more prominent when using fewer parallel workers on the farm. However, we can see that this result is directly linked to the high latencies that this backend presented in the performance evaluation, as discussed in Section 5.5.3. It may indicate the presence of lousy optimizations in GrPPI or the inability to enable the on-demand mode for this backend, as occurs with GrPPI-FastFlow. The PPI that used the least memory in the big picture was GrPPI-OpenMP. The other PPIs performed similarly in most cases.

The memory usage of the pipeline of farms composition is shown in Figure 5.17. GrPPI-FF again shows high memory usage. Again the GrPPI favors a higher memory usage due to the increased queue sizes and their strategy of avoiding extra allocation/deallocation. However, comparing GrPPI-FF to the other PPIs, the memory usage difference here is lower than that with a single farm. We believe this is because resources are tightly contested with blocking mode disabled, and more intensive stages (the last ones) 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. Therefore, to explain this would require further investigation to understand how GrPPI implements the communication mechanisms between the stages using ISO C++ threads.

The lowest memory usage with a pipeline of farms was achieved by handwritten FastFlow and GrPPI-TBB. In the single farm benchmarks, handwritten TBB and GrPPI-TBB got similar results. This is true not only for memory usage but for latency and throughput performance as well (Section 5.5.3). However, GrPPI-TBB presented such an unexpected

behavior regarding latency when using a pipeline of farms composition. We are unsure about what may cause that behavior, but we believe it also impacts the memory usage of this benchmark. On the other hand, our highly optimized handwritten FastFlow managed to use over 10% less memory than the handwritten TBB benchmark. The benefits of TBB's work-stealing execution model add some costs. Whenever a thread operates on an item, it creates a new object for the next stage, including its instance variables. Instantiating objects in a multi-thread environment can be slow and cause contention for the heap and the memory allocator data structures [Rei07]. We believe this contention may explain the extra memory TBB demanded compared to FastFlow.

In this section, we evaluated the memory usage of the PPIs with varying parallel compositions and parallelism degrees. The results helped to confirm some hypotheses from the previous sections, where we were able to draw connections between memory usage and latency in some cases. In addition, in patterns when using more complex parallel patterns, such as a pipeline of farms, FastFlow showed memory usage almost equivalent to its single farm implementation and less than TBB. This is only possible because of the low-level configuration and flexibility to compose parallel patterns that FastFlow provides. However, when such flexibility is limited, this static execution model pays the price, as we can observe in the SPar and GrPPI-FF results, the two PPIs that use FastFlow as backend.

5.7 Programmability Evaluation

Parallel programming is usually evaluated in terms of execution time and speedup. Other metrics in the stream processing domain are also considered, such as latency and throughput. [MRR12] says that a PPI should balance three properties: performance, portability, and programmability/productivity. However, programmer productivity is a critical factor that is not usually addressed in parallel programming. It is directly related to the lack of methods and tools that support parallel programming and the difficulty of performing experiments on humans [AGSF23, AGS+21]. Thus, most productivity evaluations of parallel programming are tied to code metrics such as lines of code (LOC) or cyclomatic complexity. Most related works use such metrics, as discussed in Section 5.2. However, these metrics may be inaccurate and lead users to wrong assumptions [AGS+22].

If disregarding parallel programming, there are in the literature well-known code-based methods for evaluating the programmability cost of applications. One of the most known is Halstead's method, a code-based metric used to measure the complexity of a program [Hal77]. Maurice Howard Halstead proposed it in 1977. The method is based on the observation that the complexity of a program is related to the number of unique operators and operands used in it. The Halstead method can be used to predict the effort required to develop or maintain a software program, as well as to estimate the number of

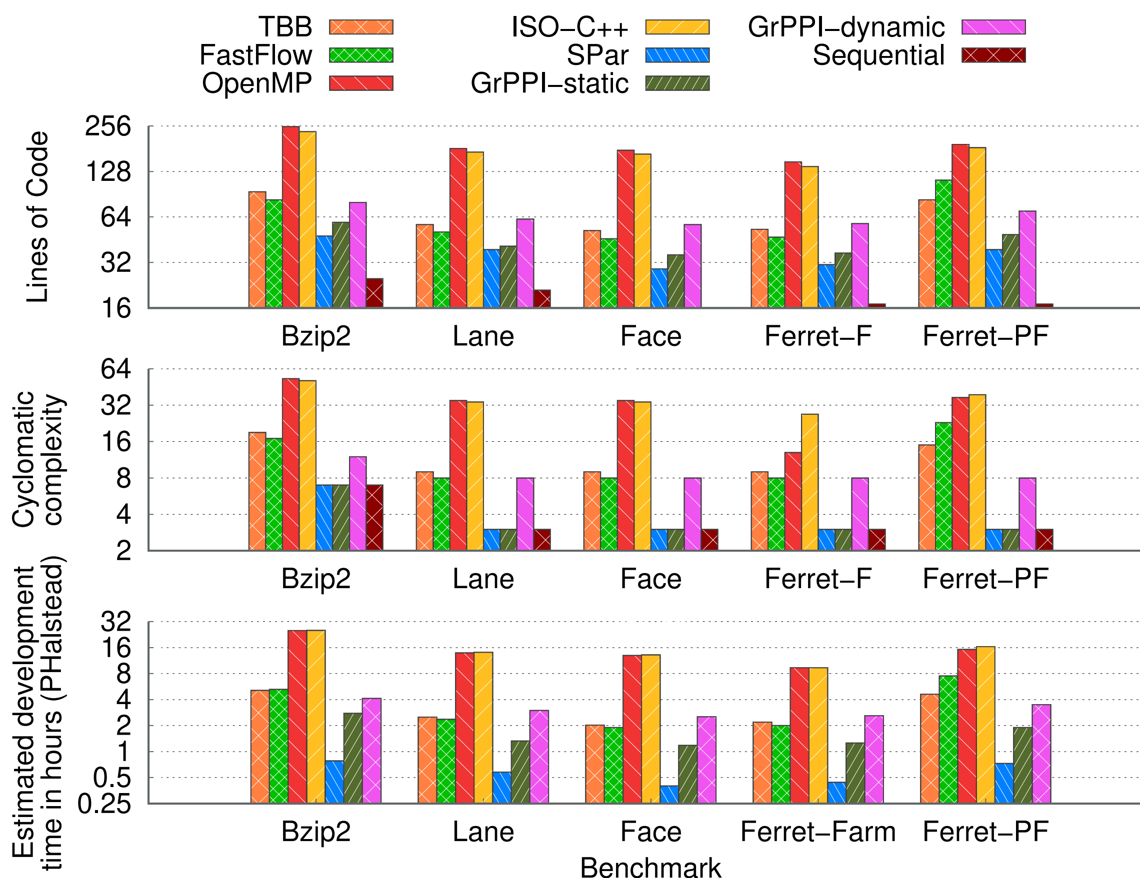


Figure 5.18: Number of lines of code, cyclomatic complexity, and estimated development time (PHalstead [AGS+22]) of the benchmarks implemented with FastFlow, TBB, OpenMP, ISO C++ Threads, SPar, GrPPI-static, and GrPPI-dynamic.

bugs that may be present in the program [AGS+22]. It can also be used to compare the complexity of different programs and to identify sections of code that may be particularly difficult to understand or maintain.

Halstead's method, however, does not correctly address parallel programming since it can not recognize specific code tokens from most of the PPIs. Andrade et al. [AGS+22] overcame this problem and adapted Halstead's method to support some PPIs, including the ones used in this work. This method is based on tokens of code, classified as operators or operands, and proposes a series of measures, including estimated development time. [AGS+22] added specific tokens from different PPIs and developed a tool called PHalstead[2] from this.

In this section, we evaluate the programmability/productivity of the PPIs using the most common metrics found in the literature and Halstead's method leveraged by the PHaltead tool. We measured LOC and CCN using the Lizard 1.17.10 tool[3]. Figure 5.18 shows the results of the single farm plus pipeline of farms (PF) implementations. For the

[2]https://github.com/GMAP/phalstead
[3]https://github.com/terryyin/lizard

LOC and cyclomatic complexity, we also added the results of the sequential applications for baseline comparison.

Regarding GrPPI, we consider two versions: “GrPPI-static” is a more straightforward implementation that invokes the executor of a specific backend statically within the code; “GrPPI-dynamic” is an implementation that allows switching between the four backends dynamically at execution time. This second version requires the addition of the backend selection mechanism represented in Listing 16. 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.

We can see in Figure 5.18 that SPar achieved the best results in all cases, followed by GrPPI-static. The SPar performed as expected since it uses code annotations that do not require code-refactoring and low-level implementation. It typically requires a few lines to enable parallelism.

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) implementations, where the complexity of programming with FastFlow increases considerably. Concerning cyclomatic complexity, SPar and GrPPI-static presented 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. The addition of the backend switching mechanism gives this extra cost. 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, this is due to the significant increase in implementation complexity with TBB and FastFlow.

The two PPIs that showed the worst programmability results were OpenMP and ISO C++ Threads (THR). These PPIs do not provide structured parallel patterns, nor do they abstract away the concurrency control mechanisms. Therefore, they require much more programming effort. Since both PPIs share some structures in the SPBench benchmarks, such as communication queues, they have similar results in all three programmability metrics.

Regarding the use of Halstead’s method, it brought interesting insights. If we look only at LOCs and CCNs, we might be misled into thinking that SPar and GrPPI-static have the same programming complexity. However, the Halstead method estimated a much lower development time for SPar. If we look at the implementation examples with SPar and GrPPI presented in Listings 6 and 16, it is somewhat noticeable that SPar presents a simpler and cleaner high-level abstraction interface. SPar does this while giving up on parallelism customizations. Therefore, we argue that metrics traditionally used in the literature, such as LOC and CCN, can be inaccurate and lead to erroneous conclusions. PHalstead is a metric that can be used to complement these analyses. It addresses specific aspects of PPIs and

proposes to be a more reliable and accurate alternative for programmability/productivity evaluation than traditional metrics.

5.8 Overview of the Results

Here we make an overview of the results from Section 5.5, Section 5.6, and Section 5.7. SPar presented the best programmability results and competitive throughput performance using a single farm. Although it can reach higher throughputs than handwritten FastFlow in a pipeline of farms, it is the consequence of its inability to enable on-demand mode in such a parallel composition. Therefore, it eliminates the on-demand overhead but pays a high price in latency for this. Also, in general, it cannot achieve latencies as low as TBB because it uses FastFlow as runtime, which in turn implements a static execution model. Besides the ease of programming, SPar is easy to use and configure. It already includes FastFlow and does not require users to install any other specific dependencies.

While the DSL SPar only generates FastFlow code, the GrPPI library allows users to run the benchmarks with four backends: FastFlow, TBB, OpenMP, and ISO C++ Threads. Unlike SPar, GrPPI does not follow the “all-inclusive” style and does not provide TBB and FastFlow within it. Thus, it is up to the users to build and configure the backend PPIs before installing GrPPI. Installing them and setting up the system environment so GrPPI can see TBB and FastFlow can be tricky for less experienced users who do not have admin access to the system.

The performance results of GrPPI varied significantly among the four backends. Since it provides no mechanism to configure FastFlow queues and scheduling policies, this greatly limited the performance of this backend. In addition, it only supports an older version of the FastFlow library (2.2.0), which also limits the users’ option to use newer features of FastFlow, such as the `BOUNDED_BUFFER` compiling macros. The second worst latency, in general, was achieved by GrPPI-THR (ISO C++ Threads). However, it did not present the throughput performance issue faced by the FastFlow backend.

GrPPI-TBB performed as well as the handwritten TBB in most cases. However, it presented unexpected increased latency when running Ferret benchmarks. Due to GrPPI’s internal implementations and lack of documentation, we could not find why it shows this behavior. For last, GrPPI-OMP presented a surprisingly good performance. In most cases, it achieved lower latency than handwritten OpenMP, FastFLow, and ISO C++ threads. In the Lane Detection benchmark, GrPPI-OMP was the PPI that presented the lowest latency when using hyper-threading. With the Ferret (farm) benchmark, it presented equivalent latency to handwritten TBB at high parallelism degrees.

Although GrPPI-OMP shows up as a good alternative, we could not run it with the pipeline of farms composition. In our experiments, we observe that it creates much more

and an arbitrary number of threads than what would be expected. For instance, it creates around 17 threads to run the Ferret pipe-farm with two workers per farm. Thus, with little parallelism, it already used all the available threads of the architecture it was running.

We could find most of the GrPPI limitations because this work is the first to evaluate it under a latency perspective, as we showed in the related work Section 5.2. All its backends presented performance or behavior issues in some of our experiments. The main issues are higher latency for GrPPI-FF and GrPPI-THR, throughput drop for GrPPI-FF, unexpected high latency for GrPPI-TBB in Ferret benchmarks, limited parallelism scaling with GrPPI-OMP in more complex compositions, and poor throughput performance and high memory usage of GrPPI-FF with a pipeline of farms implementation. Therefore, while GrPPI-THR can deliver high throughput and does not require installing additional libraries, it delivers poor latency performance. While GrPPI-OMP delivers decent latency and good throughput performance, it presents unpredictable behavior in complex parallel compositions. While GrPPI-TBB presents good latency and throughput performance, it presents unexpected performance behavior in some cases and may demand more effort to install. GrPPI-FF does not stand out from the other backends at any point in our evaluation.

TBB, FastFlow, OpenMP, and PThreads (or C++ threads) are the PPIs most used to validate other solutions in related work, as presented in Section 5.2. The handwritten benchmarks we implemented using these PPIs presented similar throughput and latency performance in most cases. Also, they showed equivalent resource usage in single-farm benchmarks. In the pipeline of farms parallel composition, TBB used slightly more memory than OpenMP and ISO C++, while FastFlow used over 10% less than the others.

Regarding programmability, OpenMP and ISO C++ threads demand the most programming effort, as expected, since they do not provide structured parallel patterns or abstract concurrency control mechanisms. In single-farm benchmarks, TBB and FastFlow require similar programming effort in our results. For the pipeline of farms composition, the three programmability metrics we used pointed out that FastFlow required less than twice the programming effort of TBB. While the productivity/programmability evaluation can somewhat address the effort of writing parallel code, it cannot address the difficulty it is to find parallelism configurations that perform decently. In the pipe-farm case, we argue that the effort required to implement a FastFlow version that achieves performance competitive with TBB demands much more effort than the metrics point out. And this goes beyond the complexity of the code itself but also encompasses the parallelism settings required to achieve the desired performance levels. In addition, code-based metrics also fail to address other aspects, such as the lack of documentation of PPIs, which would increase development time. In any case, we believe that the code-based metrics could at least point in the right direction when evaluating PPIs in our test cases. However, they may fail with respect to proportionality in more specific cases.

5.9 Chapter Summary

In this chapter, we evaluated parallel programming interfaces that support stream parallelism in C++ regarding performance, memory usage, and programmability/productivity. We first discussed the related work in Section 5.2. We looked for works that have similarly evaluated any of the currently supported PPIs in SPBench. The literature review showed that most related works evaluated performance regarding execution time/speedup only. Only [Gri16b] evaluated PPIs' latency, an increasingly relevant metric for real-time processing. Regarding memory usage, most studies do not consider different parallelism strategies and degrees of parallelism or different stream processing applications. We performed our experiments using four real-world application benchmarks, varying parallelism degrees, and different compositions of stream parallel patterns. Our work also differentiated from the others because we used larger input workloads than the studies that evaluated the same benchmark applications using the same PPIs Also, we run experiments on a newer and more robust multi-core architecture than most related work. With respect to programmability, the most used method to evaluate PPIs is through LOC and CCN.

In Section 5.4, we characterized the benchmarks' workload. It showed that SPBench applications have varying loads, which allows for representing multiple scenarios. The characterization results also help with the understanding of specific performance results. It will still be essential to explain some performance behaviors from the experiments in Chapters 6 and 7.

In Section 5.5, we evaluate the PPIs TBB, FastFlow, OpenMP, ISO C++ threads, SPar, GrPPI, and WindFlow. Section 5.5.2 evaluated the throughput and latency performance of handwritten TBB, FastFlow, OpenMP, and ISO C++ threads. The evaluation was carried out on three different computers. We used these four PPIs to create benchmarks based on the Lane Detection, Bzip2, Face Recognizer, and Bzip2 applications from SPBench. Besides the lower latency achieved by TBB, all these four PPIs performed similarly across different computers and applications.

The GrPPI performance evaluation was presented in Section 5.5.3. We compared GrPPI's backends (TBB, FastFlow, OpenMP, and ISO C++ threads backends) latency and throughput performance against the handwritten benchmarks. Our experiments showed that the abstraction layer that GrPPI applies on top of the PPIs results in several limitations that affect all PPIs' performance in some way in our test cases. Similarly, in Section 5.5.4, we evaluated SPar performance. Since SPar generates FastFlow code, we compared its performance against handwritten FastFlow and GrPPI-FF. SPar presented equivalent performance to the handwritten benchmark in most cases. However, its abstraction layer also presented some limitations that prevent users from using specific FastFlow configurations for better latency performance.

Regarding memory usage, we evaluated the performance of PPIs with single-farm and pipeline of farms parallel compositions. PPIs performed similarly in most cases when implementing a single-farm structure. The most prominent exception was GrPPI-FF, which demanded more memory than all the other PPIs combined in some situations. We observed a strong bond between high latency and high memory usage. Besides GrPPI-FF, this bond is noticeably present in TBB and FastFlow using hyperthreading in Lane Detection benchmarks and in SPar at lower parallelism degrees when it implements a pipe-farm pattern.

We finished the evaluation of the PPIs in this chapter by analyzing how they perform concerning programmability/productivity. Half of the related work does not address the programmability aspect of PPIs. Of course, carrying on such an evaluation depends on the context of the study. The other half mostly considers lines of code or cyclomatic complexity. [HLLL22] and [AGSF23] are the ones who went beyond and performed programmability experiments using humans. However, while [HLLL22] evaluated only task parallelism and with a small group of five Ph.D. level C++ developers, [AGSF23] adopted a biased methodology that favored some PPIs over others. This highlights the difficulty of adequately evaluating programmability in parallel computing. On the one hand, experiments with humans are costly and require a lot of planning. If well-planned and using a representative group of users it can lead to more accurate results. On the other hand, code-based metrics are cheap and easy to do but can lead to wrong perceptions. In this work, we contributed in this respect by evaluating the programmability of the PPIs using Halstead's method [Hal77], a widespread method for evaluating applications programming efforts from outside the parallel programming scope. This method was adapted by [AGS+22] to support stream parallelism, enabling the evaluation of the PPIs we used in this work. Although it is also a code-based metric with several limitations, it better emphasizes the differences between the high-level abstraction interfaces of GrPPI and SPar.

In this chapter, we addressed the following research problem: **"Performance analysis of PPIs for C++ stream processing is usually incomplete"**. The literature review showed that studies comparing PPIs in the stream parallelism context most evaluate it regarding performance, resource usage, and programmability/productivity. However, the related work presented several limitations in terms of representative metrics, parallelism exploration, evaluation methodology, and adequate benchmarks. Our work presents a latency and throughput analysis of PPIs, the two most important performance metrics in stream processing. However, if compared to throughput, latency is more susceptible to optimization problems and requires fine-tuning to keep it low without incurring large throughput losses. Since latency evaluation is the most lacking analysis in related work, our extensive analysis in this regard brought a new perspective on these PPIs. The programmability evaluation helped to highlight the benefits of using PPIs that offer structured parallel patterns since some of them can offer similar or better performance with less programming effort.

In general, the experimental results presented in this chapter helped show the impact that the limitations imposed by high-level parallelism abstractions can have on the performance of stream processing applications. In Chapter 6 and 7, we continue addressing the same research problem we investigated in this chapter. There, we analyze TBB and Fast-Flow under specific circumstances, such as the impact of varying data stream frequencies and micro-batching on the performance of traditional stream processing applications.

6. DATA STREAM FREQUENCY

Stream processing aims to process data as it arrives, in near real-time. Therefore, applications in this domain are susceptible to unexpected spikes, bursty phases, and other abrupt changes in the input streams. It can cause undesirable effects that negatively impact the throughput and latency or even lead to a system failure or data loss [HSS+14]. Therefore, testing stream processing systems under varying data stream arrival frequency can help to evaluate the system's performance and behavior under different load conditions. It can help to identify bottlenecks or limitations in the system that may affect its ability to handle high volumes of data or sudden bursts of activity.

In this chapter, we propose a series of algorithms to generate the literature's most used data stream frequency patterns for evaluating stream processing. We implemented these algorithms in SPBench together with mechanisms that allow users dynamically change the frequency of the input streams and combine different frequency patterns. Then, we use it to evaluate the impact of data stream frequency on the performance of stream processing benchmarks implemented with TBB and FastFlow. This chapter includes part of the study published in the papers [GGSF22a, GGSF23] and has been reproduced here in accordance with the signed copyright agreement and the copyright holder.

Section 6.1 shows the motivational scenario. Then, Section 6.2 presents this chapter's related work. In previous work [GGSF22a], we designed some preliminary strategies for generating data stream frequency patterns and then evaluated the performance of FastFlow and TBB in this scenario. We briefly present this first design and analysis in Section 6.4.1. In a more recent work [GGSF23], we redesigned the old strategy by implementing algorithms and mechanisms that allow for generating frequency patterns more precisely, predictably, controllably, and dynamically. Section 6.4.2 presents this new design with the current algorithms and mechanisms that enables it in SPBench. In Section 6.5, we use this SPBench feature to evaluate the impact of data stream frequency on stream processing PPIs and applications.

Contents

6.1 MOTIVATION **155**

6.2 RELATED WORK **156**

6.3 DATA STREAM FREQUENCY MANAGER **159**

6.4 FREQUENCY PATTERNS **160**

6.4.1 FIRST PROPOSED SOLUTION 160

6.4.2 CURRENT SOLUTION 163

6.5 EXPERIMENTAL EVALUATION **167**

6.5.1 EXPERIMENTAL METHODOLOGY 167

6.5.2 EXPERIMENTAL RESULTS 168

6.5.3 DISCUSSION OF THE RESULTS 176

6.6 CHAPTER SUMMARY **177**

6.1 Motivation

It is pretty standard for data not to arrive at constant speeds throughout the execution of a stream processing application. Fluctuations can occur due to workload characteristics, transient network issues, garbage collection in JVM-based engines, etc [KRK+18]. However, when the data frequency is significantly higher than the system's processing capacity, it can cause a buffer overflow or memory exhaustion, leading to crashes and data loss. Examples of loads that can present huge and often predicted fluctuations are data from network monitoring, traffic control, GPS, social media, etc. These fluctuations link to the times people use these services the most throughout the day and draw a waveform. However, many works in the literature need to do tests with abrupt fluctuations and shorter periods [BTO13, KRK+18, SCS17, PHUK20, AMDA20]. The result is input streams that present spikes and binary frequency patterns.

Sometimes, when the data frequency exceeds the sustainable throughput [IPV17, KRK+18] of the application, this can result in an increased allocation of computational resources and trigger amortization techniques such as batching or back pressure mechanisms [HSS+14, KRK+18]. These mechanisms are activated to avoid data loss (e.g., load shedding), and frameworks may require several minutes of resource reconfiguration to increase data throughput. There is significant research and development effort toward mitigating the impact of these fluctuations in input streams and increasing fault tolerance. Scientists are constantly developing solutions for applications in the SP domain, both for design applications using new technologies and for adaptive systems [ZSRS16, TCN+16, AMDA20, SRG+20, APTE21, LZS+22, VGDF22]. Therefore, it is important to have benchmarks that help simulate specific data overload scenarios so that these research solutions and SP systems can be properly evaluated. For example, a benchmark that simulates increasing frequency variations to analyze if the system can sustain a target throughput or latency.

Data frequency can have different meanings in stream processing. It can mean the frequency with which the application receives items of the same type. The literature uses several synonyms, such as data arrival/item/stream rate or frequency, stream pressure, input frequency, etc. Here, we define *data stream frequency* as the number of items available for the source operators to read per unit of time. In practice, in this work, we add a time delay at the beginning of the source's main loop. Decreasing the time delay entails increasing the data frequency and vice versa.

In some cases, data stream frequency merges with the concept of data intensity or complexity [HH21a]. In these cases, the workload defines its behavior by the size or computational cost of the items. In [DSTD16], for example, the authors use an image processing application and cut the resolution of the input images at specific points by half.

They used it to simulate a binary pattern. [SRG+20] used real and custom workloads for data compression with stretches of higher and lower processing costs. In this work, we do not artificially modify the input workloads because they naturally exhibit intensity patterns similar to a wave, binary, and other patterns (Section 5.4).

To our knowledge, no one has compared Threading Building Blocks [VAR19] with FastFlow [ADKT17a] in stream processing applications concerning throughput and latency under these circumstances. Works from the literature that assess these PPIs mainly use fixed data frequency. Analyzing data frequency in stream processing with multiple applications is complex and challenging. In the related work (Section 6.2), we have not found available support tools or frameworks that allow users to create custom stream processing benchmarks with native support for latency and throughput analysis under dynamic data frequency and frequency patterns.

6.2 Related Work

In this chapter, we propose a series of algorithms for generating the most used data frequency strategies for SP evaluation in the literature. We integrate them into the SPBench benchmarking framework, which already includes mechanisms for dynamically changing the data frequency on its benchmarks. Then, we use SPBench to investigate the impact of data stream frequency on the performance of traditional stream processing applications. Therefore, as related work, we considered tools and techniques for generating data streams with manageable frequency and research works that deliberately evaluated SP systems under specific data frequencies.

Das et al. [DZSS14] propose a self-adaptive algorithm to reduce the latency of distributed batched streaming systems through dynamic batching resizing. They used Apache Spark and tested the algorithm by varying the input data rate with waveform and binary (sudden low-high frequency changes) strategies. [ZSRS16] targeted the same problem with a different approach, but they also tested their algorithm using a binary strategy for data stream frequency. [SRG+20] also have the same goal, but they target compression algorithms and graphics processing units (GPUs). Here the authors tested the algorithm with four workloads presenting different complexity patterns across the dataset to vary the data intensity. Abdelhamid et al. [AMDA20] introduce an algorithm for self-adaptive parallelism for micro-batch stream processing and test it with several data stream frequency strategies.

In [DSTD16] the authors propose a reconfiguration algorithm for power-aware parallel applications. They implemented the algorithm in the PARSEC [BKSL08] suite using FastFlow and tested it by changing the size of the items to simulate drops and rises in data intensity. In [HH21b], the authors evaluated the scalability of benchmarks implemented

with Apache Flink and Apache Kafka Streams by varying the data stream frequency. The strategy they used to increase data frequency was to increase the number of data sources rather than increase the rate of item generation. ESPBench [HMP+21] is a benchmark for the enterprise stream processing domain. It implements an abstraction layer using Apache Bean, which allows running the benchmark using different DSPSs. It has a configuration file where users can adjust the input stream frequency via Apache Kafka, which is the message broker used by ESPBench. Although it is a benchmark that aims to make it simple and easy to evaluate and compare different stream processing systems, it is still limited in terms of workload, metrics (it only measures latency), and supported architecture.

RIoTBench [SCS17] is a benchmark suite for IoT stream processing. They evaluated Storm's performance under real workloads exhibiting increasing, decreasing, wave, and binary data stream frequency strategies. [vDVdP20] evaluated the performance of micro-batching DSPSs using two data intensity strategies: workload with a burst at startup and workload with periodic burst spikes. Wang et al. [WFM+19] presented a Storm-based framework for auto-elasticity and tested it using data size and complexity to tune the data intensity. In [LPDTP+12] is proposed a framework for generating data to evaluate different engines for Linked Stream Data (LSD). This framework allows different parameters to be adjusted, such as data size, the number of sources, and data stream frequency. Karimov et al. [KRK+18] propose a benchmarking framework that generates data at a configurable rate and acts as a distributed in-memory data generator for evaluating DSPSs. The authors evaluated Apache Flink, Apache Storm, and Apache Spark with different workloads. They first ran these systems at maximum intensity and then decreased it until they found the point of sustainable throughput (no back-pressure) for each. Based on this point, They varied the data frequency with a binary strategy to see how well these DSPSs could handle and adapt to spikes and rapid changes in stream speed.

NAMB [PHUK20] is a framework that automatically generates micro-benchmarks to evaluate DSPSs. It includes a Kafka synthetic workload generator that can be configured to generate data streams at different frequencies. They evaluated the micro-benchmarks using a binary strategy for data stream frequency. Balkesen et al [BTO13] proposed a framework for adaptive input admission and data management in distributed stream processing. The authors used the distributed engine Borealis and modeled synthetic and real GPS data to meet several specific data stream frequency strategies to test the framework. In [DMM16], some strategies are presented for proactive elasticity and energy awareness in data stream processing. This work targets multi-core architectures and uses FastFlow [ADKT17a] to perform the experiments. In addition to the original workload, it implements a strategy where the data frequency increases or decreases in small random steps.

BenchBase (formerly OLTP-Bench) [DPCCM13] is an open-source multi-threaded streaming load generator for benchmarking parallel database management systems

(DBMSs). It is able to generate streams with variable data frequency. The authors evaluated several DBMS benchmarks using BenchBase for generating streams with binary, spike, and increasing data frequency patterns. [TCN+16] proposed a self-adaptive batch-based streaming middleware for efficient transfers of events between cloud data centers. They test the performance and adaptability of their solution using a binary data stream frequency.

Although many of these works evaluated different stream processing PPIs under different data stream frequencies, most aim for distributed engines [LPDTP+12, DPCCM13, TCN+16, PHUK20]. Other works have evaluated PPIs that support multi-core architectures, but either the focus was on stream processing on GPUs [SRG+20], or they did not compare different PPIs [DMM16, DSTD16]. Regarding performance evaluation, they are commonly either latency-, or throughput-aware [LPDTP+12, DZSS14, ZSRS16, SRG+20, AMDA20]. Those that allow modifying the frequency of the input stream, usually through Apache Kafka, only allow static configurations or do not provide simplified ways for the users to manage it and create specific frequency patterns. Therefore, none of the related work we found compares the impact of data stream frequency on the performance of different PPIs for stream processing on multi-cores. To our knowledge, there are no similar approaches for benchmarking stream processing with data frequency configurations as SPBench does.

Data frequency pattern	**Related work**
Increasing	[LPDTP+12, DPCCM13, IPV17, IPV18, SCS17] [AMDA20, HH21b, CYH21, PGMPG21, SCW+22]
Wave	[BTO13, DZSS14, GSHW14, IPV17, RNCLP18] [IPV18, AMDA20, SCS17, APTE21]
Binary	[BTO13, DPCCM13, DZSS14, ZSRS16, TCN+16, KRK+18] [PHUK20, AMDA20, SCS17, vDVdP20]
Spike	[DPCCM13, HJHF14, IPV18] [RNCLP18, MCT+20, vDVdP20, LALC+22]

Table 6.1: Data stream frequency patterns found in the literature.

This literature review helped us identify the most used stream frequency patterns and strategies for evaluating stream processing in the literature. Table 6.1 presents the data stream frequency (or data intensity) patterns that have been used at least twice in related work. It shows that *wave*, *binary*, *increasing*, and *spike* are the most commonly used patterns. Therefore, we developed algorithms to generate data streams at such frequency patterns. Then, we extended SPBench by supporting dynamic data frequency selection and custom frequency patterns. Therefore, through SPBench, users can create custom benchmarks, run tests with specific data frequency configurations, use pre-defined data frequency patterns, or create custom strategies.

6.3 Data Stream Frequency Manager

SPBench does not yet support external data generators. All the input workload data is stored on the disk, and the Source operators are responsible for reading this data and generating the data streams. Therefore, the sources have complete control over the incoming data because all the data is already there. Data frequency can be as fast as it takes for the sources to read it from disk or memory (when running in-memory). In this model, there are three main factors that limit the maximum frequency in which sources can generate data items: (1) the cost of composing a data item, which involves adding timestamps and other auxiliary data; (2) delays introduced in the stream by subsequent stages, such as the unavailability of free workers to send the data; and (3) the lack of computing resources in an over-subscription scenario.

The data stream frequency in SPBench is controlled by adding a time delay at the beginning of the source operator. A longer time delay implies a lower data frequency and vice versa. For example, a frequency of 100 means that at most 100 items will be available to be consumed by the source per second. It does not imply that the source will generate and send 100 items per second to the workers because the source of each application has its own delay, which can be mainly caused by some of the three factors we discussed earlier or others. We used a strategy to attenuate the impact of these delays on the precision of the resulting data frequency.

Algorithm 6.1 SPBench's algorithm for frequency management

1: **function** freqManager(*lastItemTimestamp*)
2:     *frequency* $\leftarrow$ frequencyPatternGenerator()
3:     *expectedDelay* $\leftarrow$ 1000000/*frequency* ▷ In microseconds
4:     *ProcTimeOfLastItem* $\leftarrow$ currentTime() - *lastItemTimestamp*
5:     *actualDelay* $\leftarrow$ *expectedDelay* $-$ *ProcTimeOfLastItem*
6:     **if** *actualDelay* > 0 **then**
7:         *usleep*(*actualDelay*)
8:     **end if**
9: **end function**

Algorithm 6.1 presents the methodology used to manage the frequency and compute the inter-item delay. The function `freqManager` is called at the beginning of the source operator. It takes the source's last processed item's timestamp as an argument. The timestamp of each item is also assigned at the beginning of the source. The first thing `freqManager` does is adjust the frequency according to the set frequency pattern. If no pattern has been set, `frequencyPatternGenerator` returns the frequency itself. Then, on line 3, the expected time delay in microseconds is computed according to the predefined frequency. Next, it computes the time elapsed since the source received the last item. This elapsed time is then subtracted from the expected delay (line 5). If the result of

the subtraction is higher than zero, the source still has to wait to receive the next item. Otherwise, the item is released immediately. So, for example, if the user has set the frequency to 2 items per second (500 ms delay), and the source took 150 ms to process and send the last item, then `freqManager` will add a delay of 350 ms to release the next item.

Users can statically or dynamically change the data stream frequency of a benchmark in SPBench. To dynamically change it, users can use the `SPBench::setFrequency()` method in the benchmark's source code at any time during the execution. To set a static frequency throughout the execution, users can simply use the `-frequency` parameter in the exec command, as below:

```
./spbench exec ... -frequency <number of items per second>
```

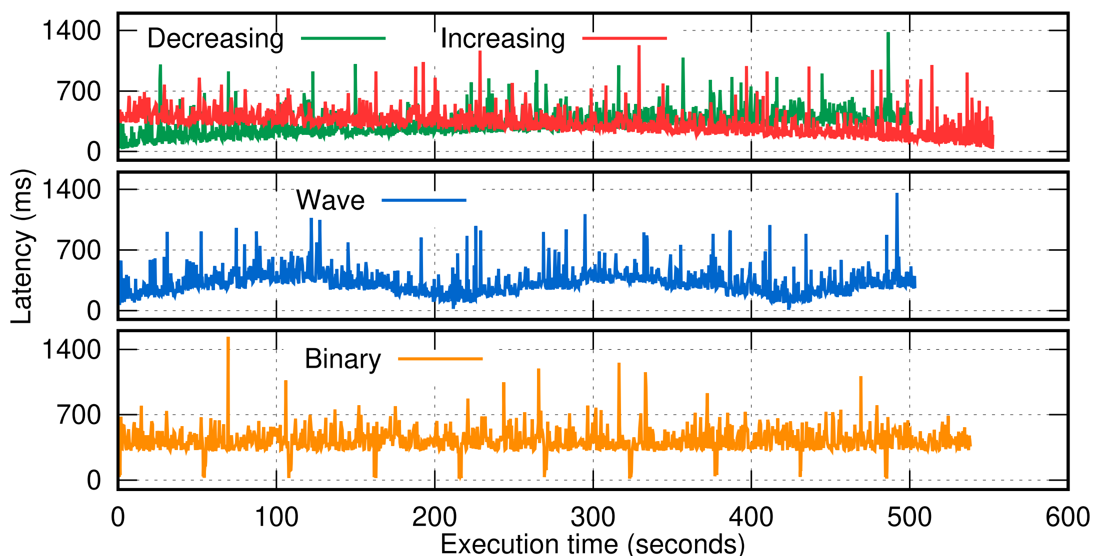
6.4 Frequency Patterns

The frequency patterns we refer to here are simply strategies for generating an x-value to be used with the `SPBench::setFrequency()` method. The strategy must compute the value of x so that the frequency data generated by `SPBench::setFrequency(`x`)` follows a specific pattern, like linear increasing or smoothly increasing/decreasing like a sine wave, spikes, etc. At first, we followed a simpler, static strategy to create these patterns. Analysis has been published using these strategies in [GGSF22a]. After that, we improved our methodology and developed better algorithms to generate the patterns. This second stage, published in [GGSF23], is the current strategy included in SPBench, and the performance analyses presented in Section 6.5 use these new strategies. Next, in Subsection 6.4.1, we briefly discuss the first strategy and some results. In Subsection 6.4.2, we present the current strategies for generating data frequency patterns.

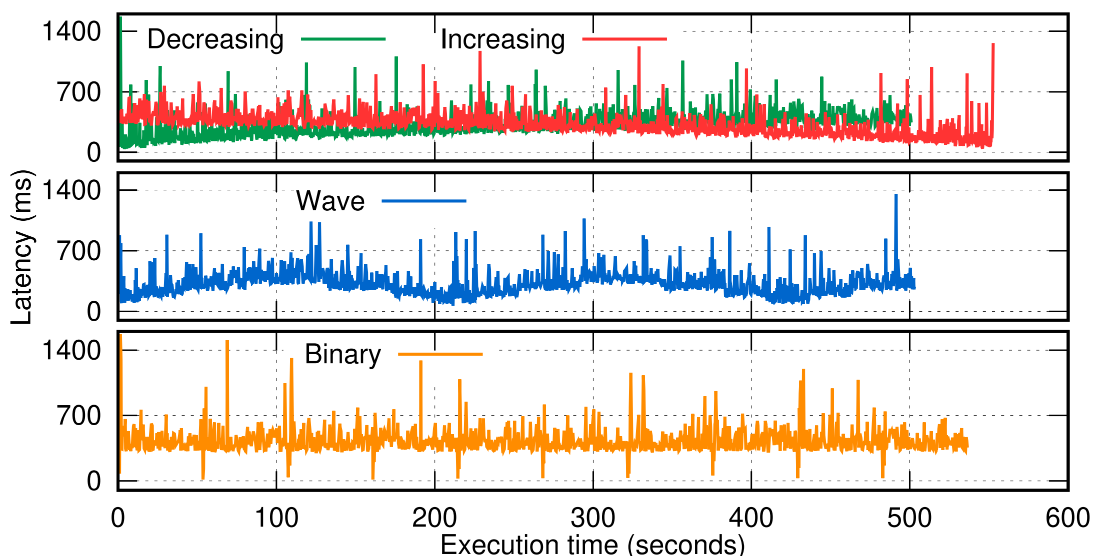
6.4.1 First Proposed Solution

In the first version of the frequency manager, instead of the user selecting a target source throughput, as it is now, the user explicitly indicated the inter-item time delay. Also, computing this time delay was done statically and required prior knowledge about the stream size. The input workload was then sliced into 20 slices, and each slice was executed with a certain frequency. Since 20 slices is a very high granularity, the result of a linear increasing pattern, for example, looked much more like a staircase than a line. The same applies to the other frequency patterns.

Thus, in each application, the amount of data items these slices contained varied. 1/20 part of Ferret's huge input workload class, for example, represents 225 items against



(a) Threading Building Blocks



(b) FastFlow

Figure 6.1: Latency of Ferret TBB and FastFlow benchmarks (farm) under different data stream frequency strategies.

Source: [GGSF22a] ©2022 IEEE.

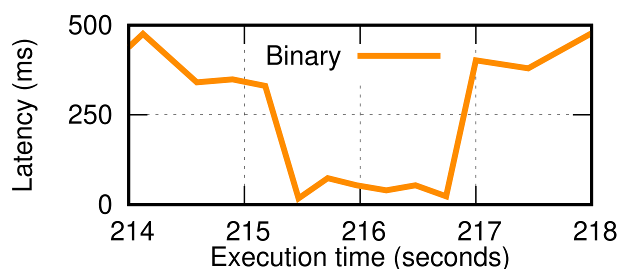


Figure 6.2: Snapshot from Figure 6.1 of a frequency switching cycle from Ferret using the binary pattern.

Source: [GGSF22a] ©2022 IEEE.

22 items in Person Recognition. Besides the problem of having to know the size of the stream in advance to compute the size of the slices, running slices at high frequency and others at low frequency mischaracterized the resulting frequency pattern. For example, in a binary pattern, is expected the frequency to interchange from time to time between a low and high-frequency state. However, if the switching of states is done over a pre-set number of items, the high-frequency state is processed and finished in moments, looking more like a spike pattern than a binary one. This behavior can be observed in the latency results in Figure 6.1, taken from [GGSF22a]. Figure 6.2 shows a snapshot of a high-frequency state of the binary pattern in Ferret. It takes less about 1.5 seconds for Ferret to process the 225 items of the high-frequency slices and about 50 seconds for the low-frequency ones. This way, the application under this binary pattern runs less than 15 seconds in a high-frequency state and more than 500 seconds in a low-frequency state. Such behavior is more a spike characteristic than a binary one. The same problem with the binary pattern can be generalized to the other patterns in Figure 6.1. In all of them, the application stays very little in the maximum frequency states (when the latency is low in the graphs).

Therefore the first strategy we used to draw the data frequency patterns did not perform as intended. The output we would expect from the fluctuating frequency patterns, such as wave and binary, is that it balances high and low-frequency states throughout the run. This behavior is nearly impossible to achieve with unstable input workloads and the static item-based slicing strategy we used in the first version. The frequency computation should not vary according to the application's behavior but according to an external factor instead, such as the elapsed time. This way, we can imitate scenarios where the data frequency is affected by external agents.

In addition, the performance metrics we used were not adequate for this type of evaluation. We measured per-item latency and global average throughput. Per-item latency makes the granularity too fine that the computing load of individual data items makes the analysis more difficult. A global average for throughput dilutes the impact of the frequency fluctuations and also makes it more difficult to extract any relevant conclusions from it. We did not evaluate the throughput in [GGSF22a] because of this problem. Ideally, we should use instant latency and throughput instead, which is computing the average of these metrics over short periods of time. It attenuates the impact of item load in latency and allows for visualizing the throughput changes throughout the execution.

Despite our challenges with this first solution, we extracted valuable analyses from the experiments. However, many conclusions are shared with the results obtained in the improved version, currently used in SPBench and which we will present in Section 6.4.2 below. Therefore, we will not present and discuss the results obtained with this first solution in this chapter. The paper with the complete analyses based on this first strategy is available in the reference [GGSF22a].

6.4.2 Current Solution

In the first proposed solution, we implemented the frequency patterns in a static way. It was item-oriented, which means that building up the frequency patterns required previous knowledge about the number of items to be processed. Then it sliced this number into twenty steps, and the frequency would increase or decrease according to the chosen frequency pattern at each step. It led to unwanted behaviors, such as the inability to keep the input at a high-frequency state for more than a small fraction of time in high-throughput applications.

To solve the issues from the first proposed solution, we have changed our strategy and improved the frequency pattern generation algorithms. In this second and current solution, the computation is time-based rather than based on the number of input data items. This way, we can draw much more predictable and accurate frequency waveforms. We recently published the strategy we present in this section in [GGSF23].

The frequency patterns currently supported by the SPBench are sine wave, binary, increasing, decreasing, and spike. All these patterns can be set via four parameters: pattern type, cycle period, minimum frequency, and maximum frequency. The exception is the spike pattern that can also get an additional parameter to define the spike duration interval as a percentage of the period (default is 10%). Users can use the frequency patterns through the `-freq-patt` exec parameter:

```
./spbench exec ... -freq-patt <pattern,period,minFreq,maxFreq>
```

Another alternative is to use the `SPBench::setFrequencyPattern()` function available for use inside the benchmark. This function can change the patterns and their behavior anytime during the run. Therefore, the data frequency in the input stream is highly configurable in SPBench. The following subsections describe how we implemented these frequency patterns within SPBench.

Sine Wave Frequency Pattern

The SPBench implements the wave pattern based on a sine wave given by Equation 6.1 [FK15].

$$A \cdot sin(2\pi ft + \varphi) + k \tag{6.1}$$

Where:

- A is the amplitude, the peak deviation of the function from zero.
- f is the ordinary frequency, the number of cycles that occur each second of time.
- t is the elapsed time.
- φ is the phase shift, where in its cycle the wave is at $t = 0$.
- k is the vertical shift from zero.

Algorithm 6.2 Sine wave frequency pattern

```
1: A ← (maxFrequency − minFrequency)/2
2: f ← 1/period
3: k ← A + minFrequency
4: function setPattern(A, f, k)
5:     newFrequency ← A× sin(2 × π × f× elapsedTime()) + k
6:     setFrequency(newFrequency)
7: end function
```

So, based on the input parameters and the sine wave equation, we implement the input frequency as described in Algorithm 6.2. In this algorithm, lines 1-3 represent the computation of some presets for the sine wave function. These variables are computed only once, or every time the pattern is modified since users can dynamically change the parameters or select a different pattern during the execution. The `wavePatternGenerator` function receives these parameters and uses them and the application's execution time to compute the sine, updating the frequency.

Binary Frequency Pattern

The binary frequency pattern interchanges between the maximum and minimum frequency set by users with an instantaneous transition, unlike the sine wave, where the transition is smooth. Therefore, the frequency under this pattern draws a kind of square wave. This pattern can simulate scenarios where sudden changes occur and remain for some time in the input stream.

The frequency in the binary pattern has two possible states: minimum or maximum. These states alternate once in a period (it changes from minimum to maximum frequency) and then repeat this pattern cyclically. Algorithm 6.3 shows the methodology we used to implement it. It uses a variable to store the current state (lines 2, 10, and 13) and changes between the states each half cycle (line 6).

Increasing and Decreasing Frequency Patterns

The increasing and decreasing patterns are not cyclic. Here, the frequency increases or decreases linearly and once, from minimum to maximum, or vice versa, over

Algorithm 6.3 Binary frequency pattern

1: $halfCycle \leftarrow period/2$
2: $isAtMaxState \leftarrow$ false
3: $halfCycleStartTime \leftarrow$ getCurrentTime()
4: **function** binaryPatternGenerator(*spikeInterval*)
5: $\quad$ $halfCycleElapsedTime \leftarrow$ getCurrentTime() - $halfCycleStartTime$
6: $\quad$ **if** $halfCycleElapsedTime > halfCycle$ **then**
7: $\quad\quad$ $halfCycleStartTime \leftarrow$ getCurrentTime()
8: $\quad\quad$ **if** $isAtMaxState == false$ **then**
9: $\quad\quad\quad$ setFrequency(*maxFrequency*)
10: $\quad\quad\quad$ $isAtMaxState \leftarrow$ true
11: $\quad\quad$ **else**
12: $\quad\quad\quad$ setFrequency(*minFrequency*)
13: $\quad\quad\quad$ $isAtMaxState \leftarrow$ false
14: $\quad\quad$ **end if**
15: $\quad$ **end if**
16: **end function**

the user-defined period. After the period, the frequency no longer changes and remains constant at the user-defined maximum frequency (increasing case) or minimum frequency (decreasing case). These frequency patterns can help test adaptive resource provision scenarios over long intervals, for example.

Algorithm 6.4 Increasing frequency pattern

1: $incrementStep \leftarrow (maxFrequency - minFrequency)/period$
2: setFrequency(*minFrequency*)
3: $stepStartTime \leftarrow$ getCurrentTime()
4: **function** increasingPatternGenerator( )
5: $\quad$ **if** getFrequency() $< maxFrequency$ **then**
6: $\quad\quad$ $stepElapsedTime \leftarrow$ getCurrentTime() - $stepStartTime$
7: $\quad\quad$ $newFrequency \leftarrow$ getFrequency() + ($stepElapsedTime \times incrementStep$)
8: $\quad\quad$ setFrequency(*newFrequency*)
9: $\quad\quad$ $stepStartTime \leftarrow$ getCurrentTime()
10: $\quad$ **end if**
11: **end function**

Algorithm 6.4 presents our strategy to implement the increasing pattern in SPBench. Generating this pattern is not as simple as it may seem. First, we divide the range between maximum and minimum frequency by the period (line 1). This value represents how much we need to increase the frequency per second to draw a linear increase over the period. However, this algorithm does not run independently, in a separate thread, for example. It runs every time an item arrives at the source. However, items may arrive at irregular intervals because of the workload's intrinsic characteristics, among other factors. Therefore, SPBench always needs to recalculate the step size on an increasing frequency. To do this, we take the elapsed time in seconds from the last step and multiply the step

size that should be taken per second (*incrementStep*) by this value. After recalculating the step size, the algorithm adds this value to the current frequency (line 7 of the algorithm). Thus, if the source operator took three seconds to process an item, the SPBench will take three steps in incrementing the frequency for the next item. Similarly, if an item took 500 ms, the algorithm will only take half an increasing step for the following item. The strategy that generates the **decreasing** pattern is exactly the opposite logic.

Spike Frequency Pattern

The spike frequency pattern is cyclical and creates one spike per period. This pattern can represent scenarios where the frequency changes abruptly for short intervals. Here, in addition to the maximum and minimum frequency and the period, users can modify the spike's duration. This duration is defined as a percentage of the period, and the default value is 10%. In 10 seconds, during the last second, the SPBench would incrementally increase the frequency to the maximum and drop it to the minimum frequency again, creating a one-second spike with a sawtooth shape, for example.

Algorithm 6.5 Spike frequency pattern

```
1:  setFrequency(minFreq)
2:  spikeInterval ← period × (spikeSize/100)  ▷ Spike size is given as percentage of period
3:  spikeIncreFactor ← (maxFreq − minFreq)/spikeInterval
4:  lowFreqInterval ← period − spikeInterval
5:  periodStartTime ← currentTime()
6:  function spikePatternGenerator( )
7:      periodElapsedTime ← currentTime() - periodStartTime
8:      if periodElapsedTime > lowInterval then
9:          step ← (periodElapsedTime − lowFreqInterval) × spikeIncreFactor
10:         newFrequency ← minFreq + step
11:         setFrequency(newFrequency)
12:         if periodElapsedTime > period then
13:             setFrequency(minFreq)
14:             periodStartTime ← currentTime()
15:         end if
16:     end if
17: end function
```

The strategy used to create this pattern is in the Algorithm 6.5. Like the other algorithms, there is an initialization phase that runs only once. First, we set the frequency to the minimum. Then we compute the duration of the spike based on the percentage of the period the user has chosen. With this value, on line 3, we calculate the spike factor, which is how much we need to increase the frequency per second during the spike phase. Here we also calculate how long the low-frequency phase, the off-spike phase in each cycle, should last (line 4). Therefore, whenever the elapsed time of the cycle is longer than the low phase (line 8), the algorithm creates a spike. The logic we use to increase the frequency

in the spike in lines 9 and 10 is the same in the increasing pattern. For the last, every time the elapsed time of the cycle is greater than the user-defined period (line 12), the spike ends, and the cycle starts again.

6.5 Experimental Evaluation

In this section, we investigate the impact of data stream frequency on the performance of parallel programming interfaces. We run experiments with four benchmarks under five frequency patterns and evaluate two performance metrics (latency and throughput). Therefore, we are only evaluating and comparing two PPIs in this chapter: FastFlow and Threading Building Blocks (TBB). We chose TBB and FastFlow because they are the two most popular PPIs that provide structured parallel patterns for C++ stream parallelism in our related work (Section 5.2). Also, these two PPIs present several differences, mainly regarding low-level implementation and task-scheduling strategies.

6.5.1 Experimental Methodology

To model the frequency patterns through SPBench, we adopted the same criteria for all benchmarks. We first run the benchmark with the same parameters but without setting a specific frequency, i.e., we run it at maximum frequency, which is delimited by the underneath hardware. Then we measure this sample's average throughput and execution time and use this data to model the frequency patterns and re-run the benchmarks. We run all the benchmarks using the in-memory mode in SPBench, which allows for achieving higher frequencies since reading data from memory is faster than reading it from disk.

For all the patterns, we set the minimum frequency to be 10% of the average throughput of the sample. The maximum frequency was equal to 110% of the average throughput. For the periodic patterns (wave, binary, and spike), the size of each period was set to 1/3 of the sample execution time for at least three cycles to occur. The period was equal to the execution time for the non-cyclical patterns (increasing and decreasing). We also set each spike duration as 10% of the period for the spike pattern.

We measured performance by monitoring each benchmark throughout its execution. We used the routines of the SPBench, which allows for performance monitoring with microsecond precision. We measure the instantaneous latency and the instantaneous throughput, which is the average of these metrics over a short time interval. These are the same metrics we used for the workload characterization, presented in Section 5.4. There, we used a 5-second average interval since fewer items are processed per second in the sequential applications. However, the parallel benchmarks we evaluate here have higher

throughputs. Therefore, we have reduced the average interval to 250 ms. We did not use an interval lower than that to avoid interfering with the results.

The experiments in this section were conducted on the Xeon Silver 4210 computer, described in Section 5.3. It is a 40-threads multicore computer. We run TBB with 40 threads and FastFlow with 40 farm workers, following the oversubscription strategy discussed in Section 5.5.1.

6.5.2 Experimental Results

This section presents the experimental results using data stream frequency patterns. We use frequency patterns widely used in related work, such as: *increasing*, *wave*, *binary*, and *spike*. We also run the benchmarks with the *decreasing* pattern provided by SPBench.

The combinations of frequency patterns and applications we tested resulted in many data. Therefore, we present the most representative results for each frequency pattern. The graphs show the frequency set for the input stream as dashed lines. Since the frequency patterns were built based on data given by pre-runs of the benchmarks, the resulting frequency patterns for TBB and FastFlow may vary. Although the throughput performance of these interfaces is very similar, minor differences can accumulate and change the target frequency as the run progresses. For this reason, graphs show the frequency set for each of the PPIs along the execution. The top graph of each figure presents the instantaneous throughput, and the bottom graph shows the instantaneous latency.

Figure 6.3 shows the latency and throughput of Bzip2, Lane Detection, and Ferret benchmarks running under a wave frequency pattern. In Bzip2 and Lane Detection, TBB benchmarks take a little longer to execute than FastFlow. Based on that, we can assume that the throughput with FastFlow is slightly higher on average since it processes the same number of items in a shorter time. On the other hand, at high frequencies, FastFlow could not sustain a low latency, presenting high latency spikes in Bzip2 and Lane Detection. If we look at Ferret's results in Figure 6.3c, however, we can see that the difference in latency between TBB and FastFlow decreases. Also, FastFlow's execution time is longer in these cases, which implies lower throughput. It was expected that FastFlow would not perform as well as TBB in this case. After all, Ferret implements the pipeline-farm parallel pattern, and load unbalancing is a big issue for Ferret. So FastFlow is at a considerable disadvantage in these scenarios. On the other hand, in applications that do not require ordering, as with Ferret, as long as there is computational resource available, a TBB task can process an item from the beginning to the end of the pipeline without interruption. So this scenario is where TBB has a significant advantage, as opposed to FastFlow.

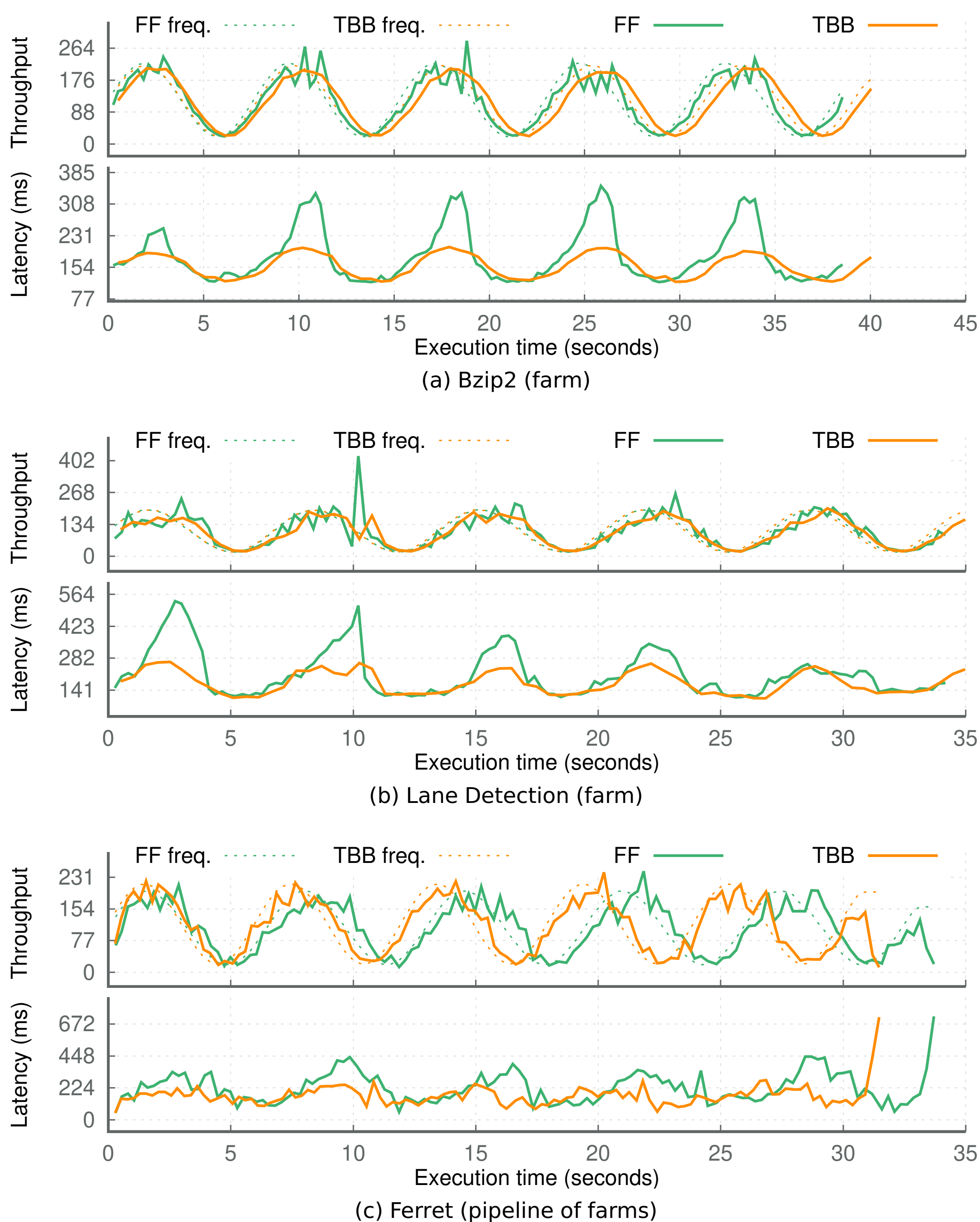


Figure 6.3: Wave frequency pattern with Bzip2, Lane Detection, and Ferret.

Source: [GGSF23] ©20XX Springer Nature.

Figure 6.4 presents the results for the binary pattern. For example, a binary pattern in a Lane Detection application can represent scenarios where there are sensor cameras with dynamic FPS, and the frame rate rapidly decreases or increases if the vehicle stops or starts moving. Comparing the results for Bzip2 with the wave pattern, it is possible to observe that the latency is higher at the maximum state in the binary case. It is an expected difference since the frequency stays at a high state in a sinusoidal wave for a smaller time interval. In Lane Detection, the latency results with wave and binary are

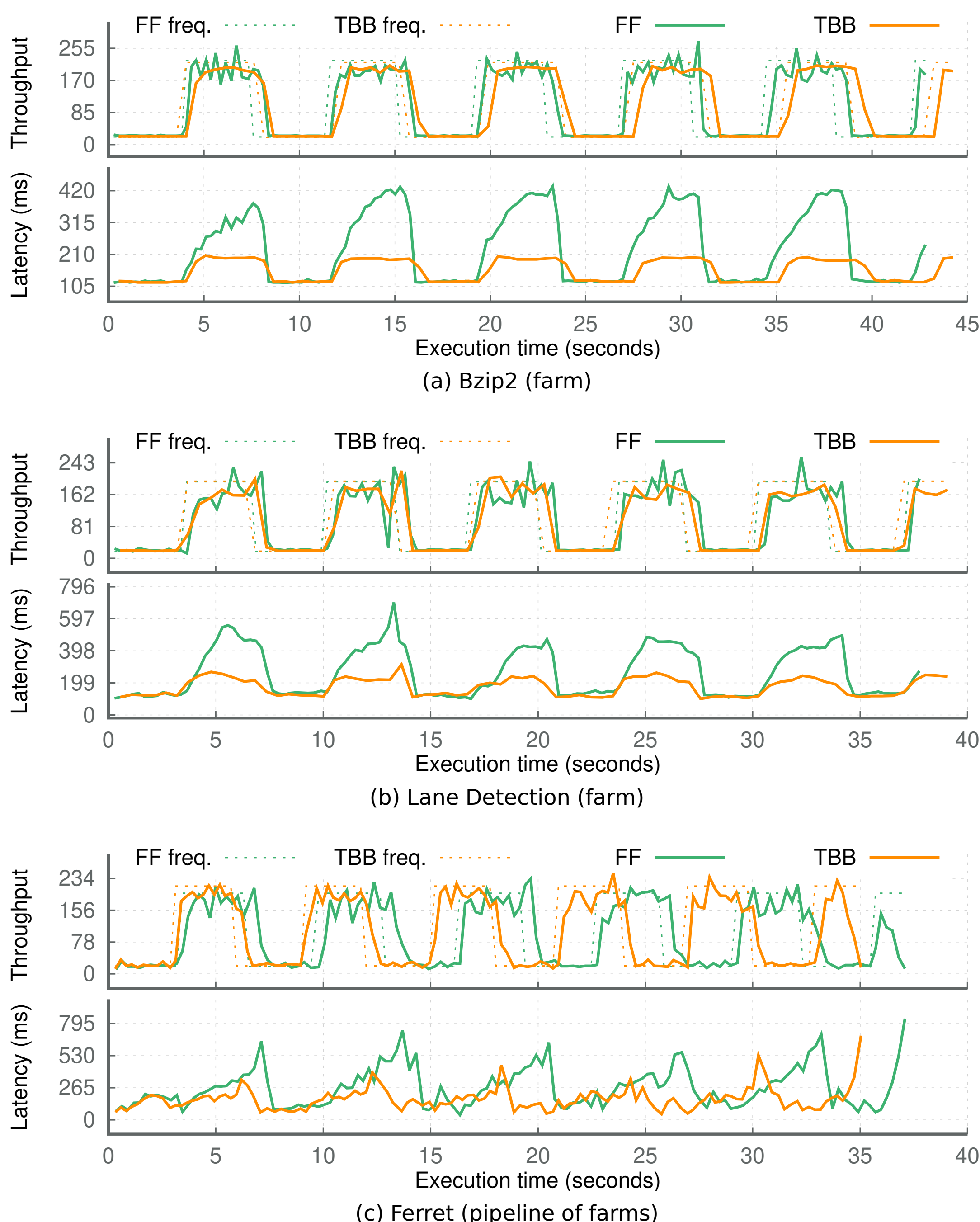


Figure 6.4: Binary frequency pattern with Bzip2, Lane Detection, and Ferret.

Source: [GGSF23] ©20XX Springer Nature.

a bit different. With the wave pattern, FastFlow reduces the latency in the latter part of the execution, even at high-frequency times. However, this behavior is not observed in the binary pattern. We hypothesize that this is related to the natural frequency of the workload in this application. In Figure 5.2, we can see that the latency already shows a kind of wave pattern itself in the final part. So we assume that this wave pattern from the workload matches the wave pattern generated by the SPBench. It probably occurs in the binary pattern as well. After all, the latency peaks are also smaller in the final part of the

execution, but the effects are less noticeable due to the sudden state changes. Besides the higher latency presented by FastFlow, in Lane Detection, its execution time was almost 5% lower than TBB. Regarding Ferret with binary frequency pattern, the behavior was very similar to the wave pattern, and the same considerations apply here.

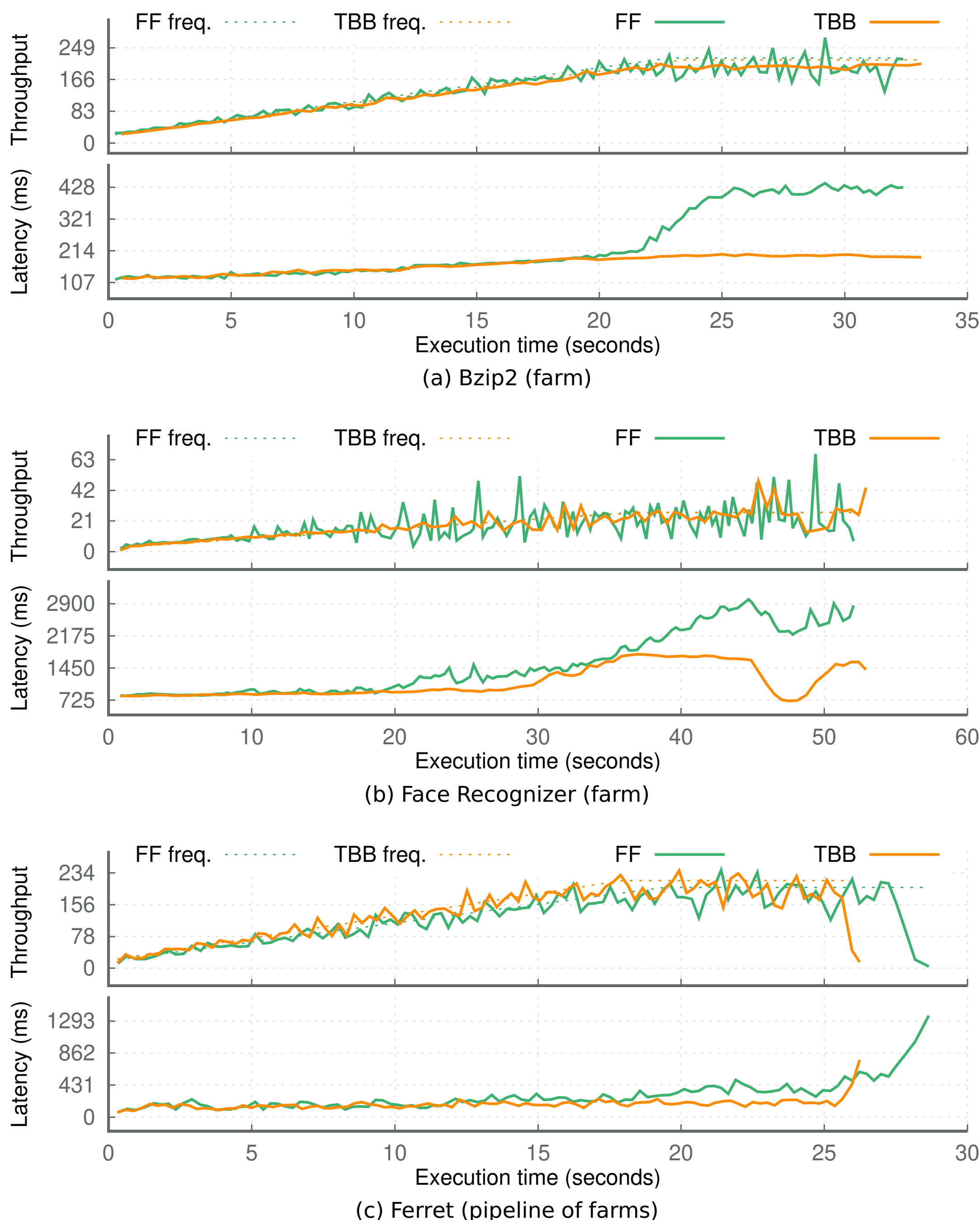


Figure 6.5: Increasing frequency pattern with Bzip2, Face Recognizer, and Ferret.

Source: [GGSF23] ©20XX Springer Nature.

The results for the increasing frequency pattern are in Figure 6.5. Here we evaluate the benchmarks for Bzip2, Face Recognizer, and Ferret applications. Bzip2 and Face Recognizer presented a distinct behavior. As previously discussed, the maximum

target frequency in this experiment is 110% of the average throughput the application can sustain at maximum frequency. Since Bzip2 has a more stable input, presenting minor fluctuations along with the execution (see Section 5.4), once the target frequency reaches the threshold given by the sustainable frequency, the throughput stabilizes, and there is a latency jump with FastFlow (Figure 6.5a). We believe that the main reason for this is that, in FastFlow, the source reads an item from the input stream even if there is no free worker to process it or free space in their queues [ADKT17a]. So, this item remains enqueued, waiting to be processed, increasing its processing time, thus the latency. In the TBB state-based pipeline, when an item (token) is assigned to a thread, this thread can run this item through all the stages of the pipeline until it is no more possible (when it encounters an ordered stage and the item is not in the correct order, for instance) [MSS04]. However, cluttering is less present in more stable workloads, mainly when items are more computationally costly (Bzip's case). Therefore, this method can avoid queuing items between stages, implying reduced latency. The steady throughput increase of TBB can confirm it.

On the other hand, the natural latency of the Face Recognizer workload is not as stable as that of Bzip2. Therefore, items are processed at different speeds by each worker. So as we increase the frequency, the item clutter gets worse. It dramatically impacts FastFlow, increasing latency before the frequency reaches the sustainable throughput level (Figure 6.5b). However, it also impacts TBB, which shows a rapid increase in latency at the moment of maximum frequency. In this scenario, a single TBB thread cannot process an item from end to end of the pipeline. If the item arrives in a disordered state at the last stage, the thread puts that item in a buffer and takes another task to execute, increasing the latency for that item.

Regarding Ferret (Figure 6.5c), FastFlow's latency is almost equal to TBB when the frequency is slightly below the maximum sustainable throughput. So, in scenarios like increasing, where most of the time the frequency is below this limit, FastFlow can keep the latency closer to TBB for longer. That is because, under low frequencies, there are more chances of having resources to process the items that the source takes from the input stream. With TBB and its task scheduler [VAR19], in our experiments, there will be resources to keep processing further an item taken from the input stream most of the time. It contributes to keeping the latency low. However, like in the other frequency patterns, Ferret has this throughput drop and latency spike in the last few seconds, which occurs in both FastFlow and TBB. We do not precisely know what causes this behavior, but we suspect it is a consequence of load unbalancing.

Figure 6.6 shows the behavior of Face Recognizer, Lane Detection, and Ferret applications when run under the decreasing frequency pattern. Unlike the increasing pattern, at the end of the down-ramp, the decreasing pattern remains at the minimum frequency (10% of maximum sustainable throughput), so it is expected that the applications would take longer to run after the ramp. It could be different if we set a more extended

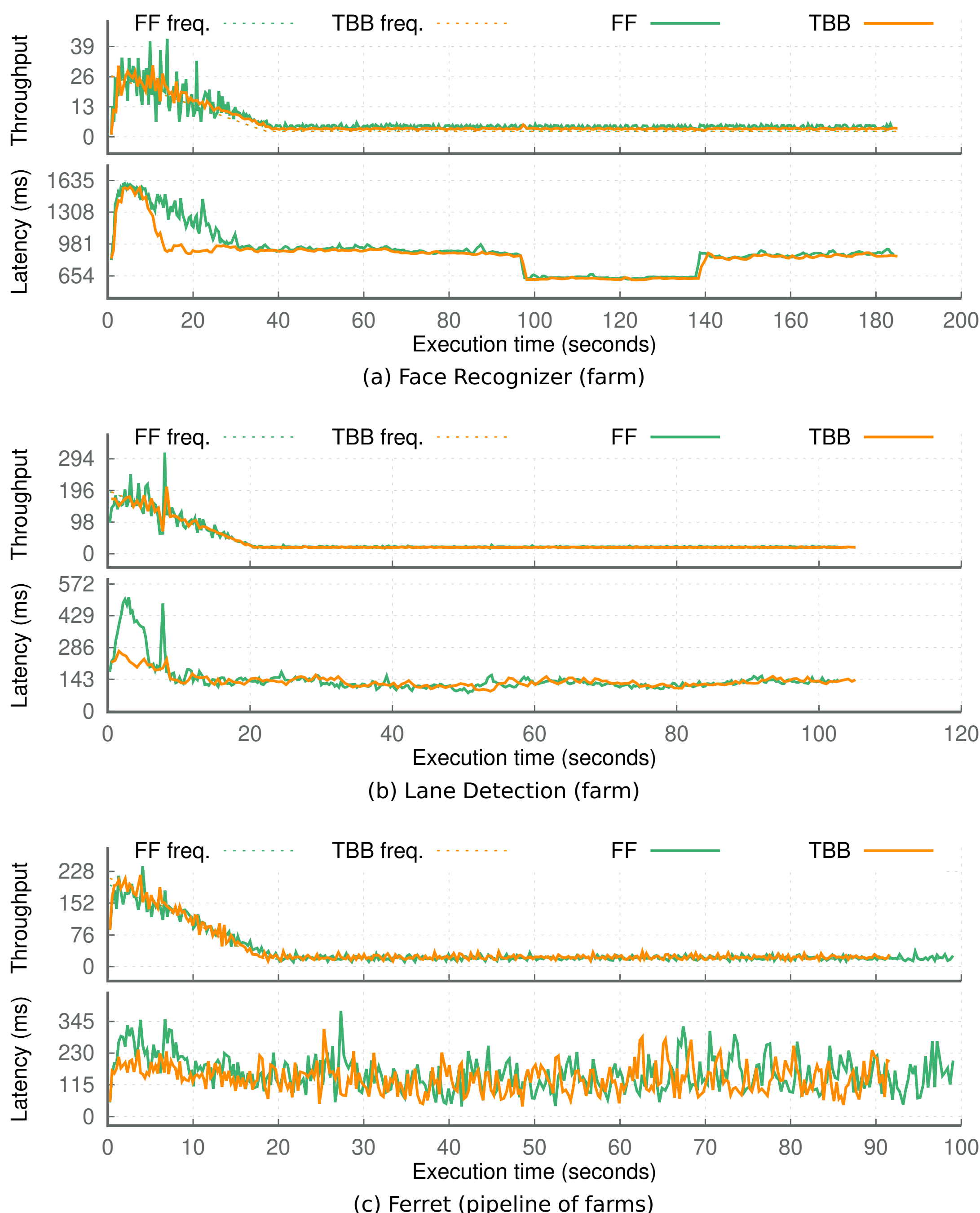


Figure 6.6: Decreasing frequency pattern with Face Recognizer, Lane Detection, and Ferret. Source: [GGSF23] ©20XX Springer Nature.

ramp period, but we prefer to use the same period we used in the increasing pattern for comparison purposes. Even though the throughput remains stable at the minimum, in Face Recognizer, there is still a low latency moment after 100 seconds that represents the same pattern seen in Figure 5.2. Since the frequency manager only limits the maximum throughput and discounts the processing time of the item at the source from the added delay, it is expected that it will not affect the native low-latency moments of the workload.

In this decreasing pattern, the TBB Face Recognizer exhibits the opposite behavior of FastFlow in the increasing pattern with Bzip2 (Figure 6.5a). Here the TBB can bring down the latency as soon as the frequency falls below sustainable throughput. Regarding FastFlow, Face Recognizer has a pretty steady throughput, and the hump present in the workload (Figure 5.2) makes the average throughput of this application higher. Therefore, our frequency strategy overestimated a little the sustainable throughput of this application. According to the previous experiments, FastFlow latency seems more sensitive concerning sustainable throughput. Maybe this is why it takes longer to keep up with the TBB latency. A similar effect occurs with Lane Detection, where the natural throughput of the workload at the beginning of the execution is lower than the average. So it is another scenario of overestimated sustainable throughput, but here it occurs only during a short interval. This way, we believe this can contribute to FastFlow reducing latency to a minimum before the end of the decreasing ramp.

In Ferret, TBB and FastFlow showed similar behavior, although FastFlow has a longer execution time. In this application, the workload varies a lot and quite frequently. So overestimation of sustainable throughput is not a problem in this case. Regarding Ferret's large latency fluctuation (Figure 6.6c), Figure 5.2 shows that Ferret is the application that has the most extensive variation in latency caused by the input workload itself. This behavior has a direct impact on the results presented here.

Figure 6.7 displays the experimental results using the spike pattern. As expected, applications take longer to run with the spike frequency strategy. After all, in spike, the frequency is at a minimum most of the time. It explains the increase in the number of cycles that occur (one spike per cycle) compared to the other cyclic patterns. In Figure 6.7a are the results of the Bzip2 benchmark. TBB and FastFlow showed similar values regarding throughput, latency, and execution time. It is noticeable from the results with the other frequency patterns that FastFlow increases latency over TBB when the frequency approaches the maximum. However, if we look at Figure 6.4a, we can see that it takes a while after the frequency increase for the FastFlow's latency to rise if compared to TBB. This way, with the spike pattern, it does not happen in Bzip2, probably because of the short spike duration (about 0.8 seconds). Also, FastFlow has higher throughput spikes than TBB since it ran faster, as in most cases implementing the farm pattern as expected. Although the latency spikes of the TBB are somewhat smaller, it is possible to observe some moments of higher latency in the off-spike intervals.

Face Recognizer does not have as stable a workload in terms of latency and throughput as Bzip2, as seen in Figure 5.2. Therefore, in this application, the spikes in the graphics (Figure 6.7b) appear more shapeless than in Bzip2. Moreover, its sequential throughput is almost ten times lower, just over 1 item per second in the sequential version in our experiments. In addition, the application takes longer to execute, implying more extended periods and, consequently, slightly longer spikes. Although the application

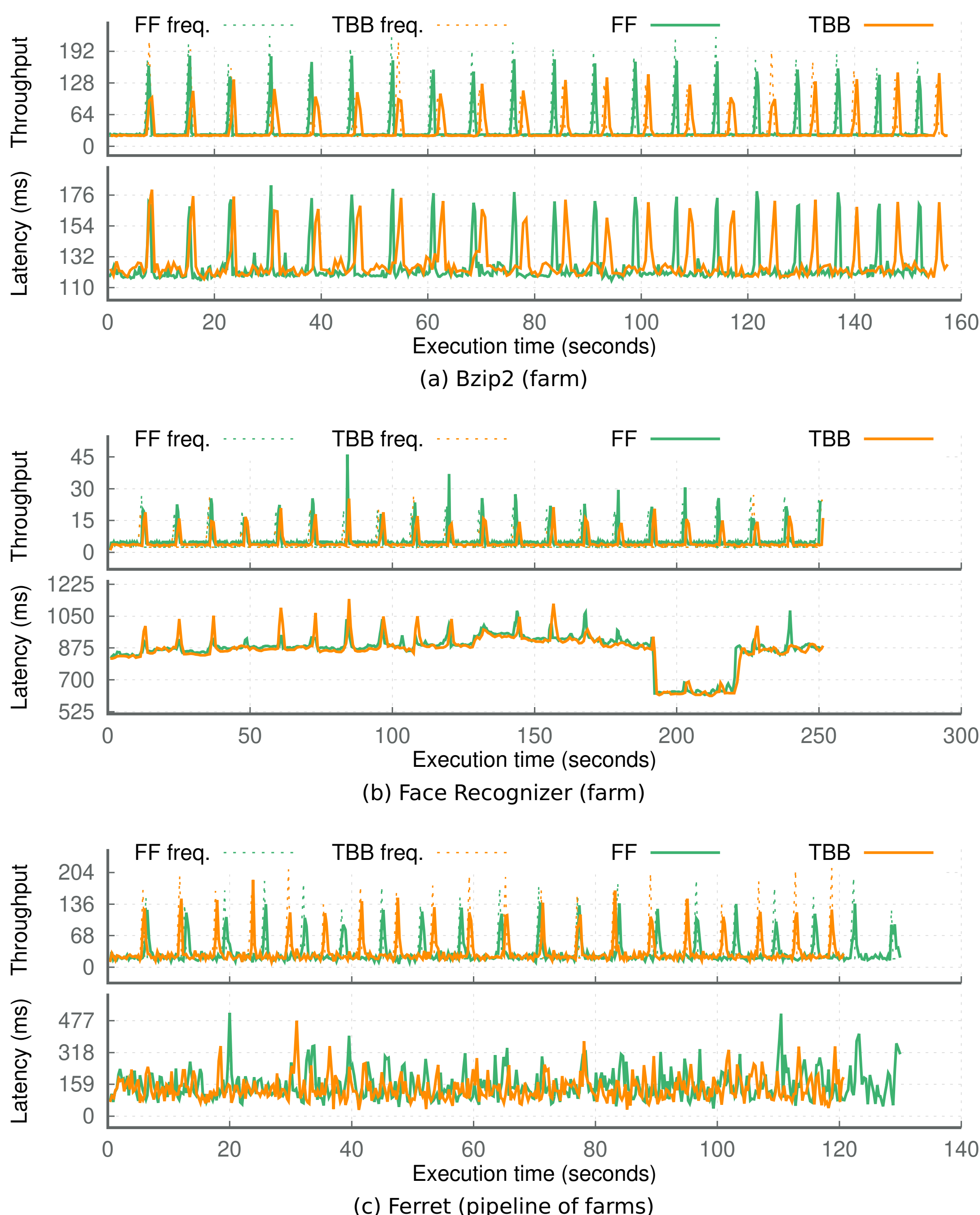


Figure 6.7: Spike frequency pattern with Bzip2, Face Recognizer, and Ferret.

Source: [GGSF23] ©20XX Springer Nature.

behaves similarly to Bzip2 in throughput, the latency results change quite a bit. Since the latency changes a lot in this application and many items are processed during spikes, a slight shift in execution time causes the TBB and FastFlow to end up processing distinct parts during the spikes of the input stream. This way, its natural load-induced latency mixes with the latency caused by the spike pattern, causing large latency spikes sometimes with TBB and others with FastFlow. A similar effect occurs with Ferret (Figure 6.7c), where spikes

occur more erratically because of the input stream's significant variations in throughput and latency.

6.5.3 Discussion of the Results

In this section, we evaluated the impact of data frequency on the FastFlow and TBB benchmarks. FastFlow showed more erratic behavior than TBB in all cases during high-frequency states. We believe that the differences in the implementation of the buffers/queues may cause this behavior. In FastFlow, even using queues of size one (on-demand policy), each worker has its input and output queue. So even with short queues, many items are in the pipeline throughout the execution. Each 40-worker farm can hold up to 80 items in FastFlow (40 queued plus 40 with the workers). Therefore, those items already in the pipeline are no longer under frequency constraint, and performance fluctuates more. In TBB, this could also occur in specific highly cluttered input scenarios.

On the other hand, FastFlow showed slightly higher throughput in most farm-parallel benchmarks. Some factors may prevent TBB from running faster than FastFlow in most farm implementations we tested. The benefits of TBB's work-stealing scheduler add some costs. Whenever a thread operates on an item, it creates a new object for the next stage, including its instance variables. Instantiating objects in a multi-thread environment can be slow and cause contention for the heap and the memory allocator data structures, inhibiting concurrency [Rei07, VAR19]. Also, we used the same value to set the number of replicas of FastFlow workers and the maximum degree of parallelism in the TBB parallel stages. However, in TBB's task model, this degree of parallelism means the maximum number of threads that will execute. The TBB "workers" must share the work with the source (emitter) and sink (collector) stages. In FastFlow, the workers each run in their dedicated thread. Therefore, the number of threads used in a FastFlow farm is always greater than the number of worker replicas because it creates one more thread for the source and one more for the sink. Still, these two stages in our benchmarks consume fewer resources than the worker stages. So, if running FastFlow in blocking mode (the case in this paper), these two extra threads that FastFlow creates may give it a short performance advantage over TBB in the parallel farm pattern.

FastFlow's performance is inferior to TBB in the Ferret application with varying data stream frequencies. Part of this can be explained by the maximum frequency of FastFlow being significantly lower than that of TBB in the experiments. However, this is the methodology that was chosen and that we explain in Section 6.5.1. besides, in FastFlow, we could exploit pipeline and stream parallelism in several other ways. For example, with FastFlow on Ferret, we could implement combined nodes or a farm of pipelines, possibly resulting in better performance, as we show in Chapter 5. But we preferred to keep the

original Ferret structure which is a pipeline of farms. The other benchmarks we used already address a similar structure to the combined nodes in FastFlow. Also, reducing the degree of parallelism of the less intensive stages would use fewer resources, which could be allocated to the more intensive ones. Fewer queues would be available to hold items, implying lower latency in this case. However, optimal configurations in stream processing applications take work to achieve. It depends on the type of application, the workload, the architecture, and the available resources, among other factors [LZS+22]. Also, the optimizations can benefit the throughput or latency more, as shown in previous work [GGSF22b]. On the other hand, programmers have more freedom to fine-tune FastFlow applications compared to TBB. Therefore, TBB and FastFlow have advantages and disadvantages in different aspects, from performance to the possibility of optimization, fine-tuning, and parallelism modeling.

6.6 Chapter Summary

In this chapter, we investigate the impact of data stream frequency on the performance of parallel programming interfaces. For that, we reviewed the literature and identified the most used frequency configurations for evaluating stream processing systems. We created algorithms to generate the most used patterns and extended SPBench to support these patterns and provide them for users dynamically. Then, we used SPBench to create benchmarks using two of the most used PPIs that provide structured parallel patterns for stream parallelism in C++. We focused on using structured parallel patterns to reduce the scope of the analysis since in low-abstraction level PPIs, such as C++ threads and OpenMP, the parallelism can be explored in more unbounded ways, and the performance of low-level structures is in the programmers' hands. Then, we ran the benchmarks under the five data frequency patterns and measured their performance regarding latency and throughput.

The experimental results showed that FastFlow and TBB perform differently under varying frequencies. Regarding throughput, FastFlow outperformed the TBB single-farm benchmarks by up to 5% in all of our test cases. With respect to latency, FastFlow and TBB present similar latency when data frequency is reduced. However, TBB also manages to keep a low latency even when using frequencies higher than the sustainable throughput of the benchmarks. In the Ferret benchmarks with a pipeline of farms parallelism, TBB performed better than FastFlow in all the cases. However, the initial conditions for setting up the frequencies (Section 6.5.1) may have hindered the possible throughput gains of FastFlow under varying frequencies in pipe-farm implementations.

Therefore, the contributions of this chapter were a series of algorithms that allow generating data frequency patterns for stream processing applications, mechanisms that allow users to easily benchmark stream processing using such patterns, and analysis of

FastFlow and TBB performance under varying frequency scenarios. We have not found in the literature any similar analysis of FastFlow and TBB from this perspective.

7. MICRO-BATCHING

Micro-batch (or mini-batch) processing is a variant of traditional batch processing. Micro-batching systems process data in small groups at a higher frequency. Stream processing systems can use micro-batching as an optimization technique that trades throughput for latency [HSS+14, HOL22]. It also may enable efficient resource utilization, high throughput, fault tolerance, and data consistency. In stream processing, micro-batching is often used to improve systems' adaptability to fluctuations in input streams. The best benefits of using batching apply in heterogeneous or distributed environments, where there is a cost of communicating data, and batching may reduce the total cost. However, there are many possible benefits of using micro-batching in shared-memory environments, which we will discuss in the next section. Even so, we found little related research investigating the impact of using micro-batches on the performance stream processing applications on multi-cores.

In this chapter, we investigate how micro-batching size impacts the throughput and latency performance of SP applications using FastFlow and TBB PPIs. In Section 7.1, we further discuss our motivational context. This chapter's related work is addressed in Section 7.2. The mechanisms we implemented in SPBench for batching support are presented in Section 7.3 In Section 7.4, we run experiments with the TBB and FastFlow benchmarks with varying parallelism degrees and micro-batch size. Then, Section 7.5 summarizes and adds final remarks on this chapter.

Contents

7.1 MOTIVATION **180**
7.2 RELATED WORK **181**
7.3 PROPOSED SOLUTION **183**
7.4 EXPERIMENTAL EVALUATION **183**
7.4.1 EXPERIMENTAL METHODOLOGY 183
7.4.2 EXPERIMENTAL RESULTS 184
7.5 CHAPTER SUMMARY **188**

7.1 Motivation

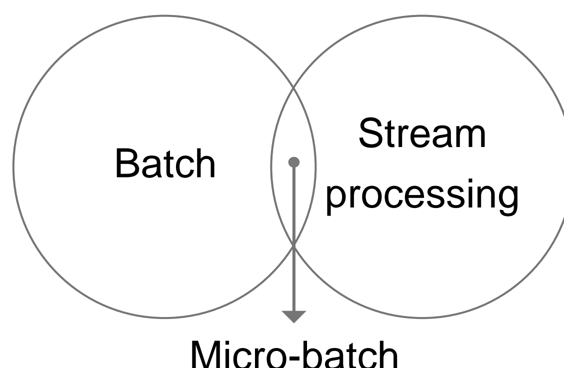


Figure 7.1: Micro-batching.

Micro batching is a middle-ground between batch processing and stream processing that balances latency and throughput, as illustrated in Figure 7.1. It is a technique that is often used to tune the performance of stream processing applications. It involves grouping small amounts of data into batches and then processing the batches as if they were a single stream. The size of a micro-batch may vary according to predefined criteria, such as time intervals (e.g., each 2-second interval forms a new micro-batch), data size (e.g., micro-batches with 5 MB of data), number of items, and others [DZSS14]. In general, the optimal size is the one that achieves the desired trade-off between throughput and latency [SRG+20]. However, this is not a static value since fluctuations in the data frequency and processing cost of each item is very common in stream processing. Therefore, the batch size is considered one of the most important tuning parameters in stream processing systems [DZSS14, HOL22].

In DSPSs and heterogeneous systems, micro-batching can add clear benefits. For example, stream processing with GPUs requires batching input elements for efficient resource utilization [SRG+20]. Since most benefits are tied to communicating data, the potential advantages of using it on multi-core systems are not so obvious. However, besides the disadvantage of increased latency, micro-batches may add many advantages to multi-core stream processing. Enabling batching support in an application implies adding loops. Therefore, the compiler may optimize these loops with unrolling techniques using software pipelining. It can also enable vectorization. Besides, micro-batching may improve throughput by amortizing operator-firing and communication costs. Such amortizable costs can include deeply nested calls, warm-up costs (e.g., for the instruction cache), and scheduling costs, possibly involving a context switch [HSS+14]. Also, as data items tend to be added in order inside each batch, it can potentially reduce data cluttering in applications with ordering constraints. In Addition, micro-batching can ensure system stability and lower latency for a wide range of auto-adaptive algorithms workloads despite significant

variations in data rates and operating conditions [DZSS14]. Such algorithms can achieve performance levels without demanding extra resources or leading to data losses.

The primary control variable in micro-batching is batch size. Ideally, it may be either statically or dynamically set [HSS+14]. StreamIt [TA10], for example, has a static batching algorithm that aims to trade the data-cache cost of requiring larger buffers for the benefits of using instruction-cache when processing micro-batches. However, systems that statically set micro-batch sizes may exhibit high latency under lower loads or may not cope with bursts in data frequency or item processing cost [DZSS14]. On the other hand, self-adaptive methods commonly use dynamic batching for reacting to load changes and maintaining system stability. Many works focus on developing algorithms that exploit dynamic batching to improve performance or resource utilization [WCB01, CcR+03, DZSS14, ZSRS16, SRG+20, AMDA20]. These works require the researcher to allocate extra time to implement benchmarking support, diverting from the research scope. Therefore, we argue that there is a demand for tools like SPBench with batching support, which can be helpful for researchers in this area.

7.2 Related Work

As related work, we mainly considered research towards investigating the impact of micro-batch sizing on the performance of stream processing applications. We also searched for tools that share the SPBench goal of enabling easy benchmarking of stream processing with micro-batches.

Das et al. [DZSS14] propose a self-adaptive algorithm to reduce the latency of distributed batched streaming systems through dynamic batching resizing. They used Apache Spark and tested the algorithm by varying the input data rate with waveform and binary (sudden low-high frequency changes) strategies. Zhang et al. [ZSRS16] targeted the same problem with a different approach, but they also tested their algorithm using a binary strategy for data stream frequency. Stein et al. [SRG+20] also have the same goal, but they target compression algorithms and graphics processing units (GPUs). Here the authors tested the algorithm with four workloads presenting different complexity patterns across the dataset to vary the data intensity. Abdelhamid et al. [AMDA20] introduce an algorithm for self-adaptive parallelism for micro-batch stream processing and test it with several data stream frequency strategies. [DSMV+20] proposed mechanisms for improving the latency and throughput of DSPSs by automatically adjusting the size of micro-batches using a feedback loop to collect metrics and make decisions.

TS-BatPro [YWVS17] is a framework that integrates time- and space-based batching techniques to optimize the energy consumption of latency-constrained applications in multi-core data centers. The technique consists of batching items to keep processors in a

low-power state for longer. The authors evaluated the framework using different stream processing benchmarks, including Ferret from the PARSEC suite. [TTMD22] proposes FastFlow library extensions to execute applications in a distributed-memory environment. They evaluated their solution by running a word count benchmark in a distributed environment and varying micro-bach size from 8 to 128. They manage to improve throughput performance when increasing the micro-batch size up to 32 messages per batch.

[Poh17] shows the influence of parallelism on the instruction level with vectorization and multithreading. They argue that SIMD effects improve performance when data is stored in a cache-friendly way within contiguous memory. Since tuples cannot always be processed one after another, they propose a batching mechanism with cache-friend formation of batches. It is a good approach since, without careful reordering, any vectorization speedup would be lost through expensive scattered memory accesses [PRR19]. [PSTF12] also relied on combining batching and vectorization techniques to improve the performance of stream processing applications. They achieve 40% speedup improvement on a network checksum application, but no performance improvement FM Radio, a more realistic and representative SP benchmark. The authors argue that the performance gains with the first benchmark came from two strategies: (1) sending an amount of data in a multiple of the cache line size and (2) segregating producer and consumer in different cache lines. Therefore, they only achieved this results because they reduced the data sharing involved in the communication and avoided excessive coherency, such as false sharing, thus avoiding the costs associated with memory cache coherency.

Thus, most work investigating micro-batching in stream processing targets only distributed platforms with DSPSs. In this context, the benchmarks that allow exploring micro-batching transfer this responsibility to the DSPSs (Spark usually) [DZSS14, SCS17, KRK+18]. While both [Poh17] and [PSTF12] have taken advantage of batching with SP on multi-cores, they have created very specific test scenarios that are difficult to occur naturally in the real world. In this chapter, we aim to see if any of these factors can occasionally impact the performance of SP real-world applications. Although [TTMD22] managed to improve FastFlow's throughput when increasing the size of micro-batches, they ran the experiments in a distributed platform and used a word count benchmark. This benchmark has small messages as a data item. Therefore, aggregating this data up to a certain point compensates for the overhead of communicating single small messages through the network. Therefore, none of the related work we found compares the impact of micro-batching size on the performance of different PPIs for parallel stream processing on multi-cores.

7.3 Proposed Solution

To implement batch support in SPBench, instead of each item containing a single piece of data, it contains an array of data and carries information about its batch size. We have added loops to all the operators in the application to process these arrays. This way, the size of the batches can be changed statically or dynamically. The following command is an example of how to use batch statically:

```
./spbench exec ... -batch-size 5 -batch-interval 2
```

The command above will run the application creating micro-batches with 10 items or a 2-second interval, whichever occurs first. Therefore users can set a single batch size limiting parameter or both combined. To change batch sizes dynamically during execution, users can use the methods `SPBench::setBatchSize()` or `SPBench::setBatchInterval()` inside the code. All these commands will manage the size of the batches in the source operator when generating the data items.

The strategy we defined to define the size of the batches is illustrated in Figure 7.2. As long as there is data to be processed, the source operator will try to apply the logic of the flowchart. The algorithm will try to decide whether to close and send the item or not based on the closing criteria, which can be a specific size, a time interval, or both. In the case of both, the condition that occurs first prevails. That is, the batch is closed if it fills up with the specified maximum number of items or if the batch interval times out. When there is no more data in the input stream, the batch is closed anyway and sent to subsequent operators. When there is no more input data, the algorithm stops the source.

7.4 Experimental Evaluation

In this section, we run performance experiments with the SPBench benchmarks under varying micro-batching sizes. We choose to evaluate only TBB and FastFlow PPIs because they are the two most popular PPIs in the literature that provide structured parallel patterns for stream processing in C++.

7.4.1 Experimental Methodology

For micro-batch experiments, we first evaluate it under multiple parallelism degrees. Thus, we used static micro-batch sizes for these experiments. We also evaluate it by dynamically changing the size of the batches during the execution of the benchmarks.

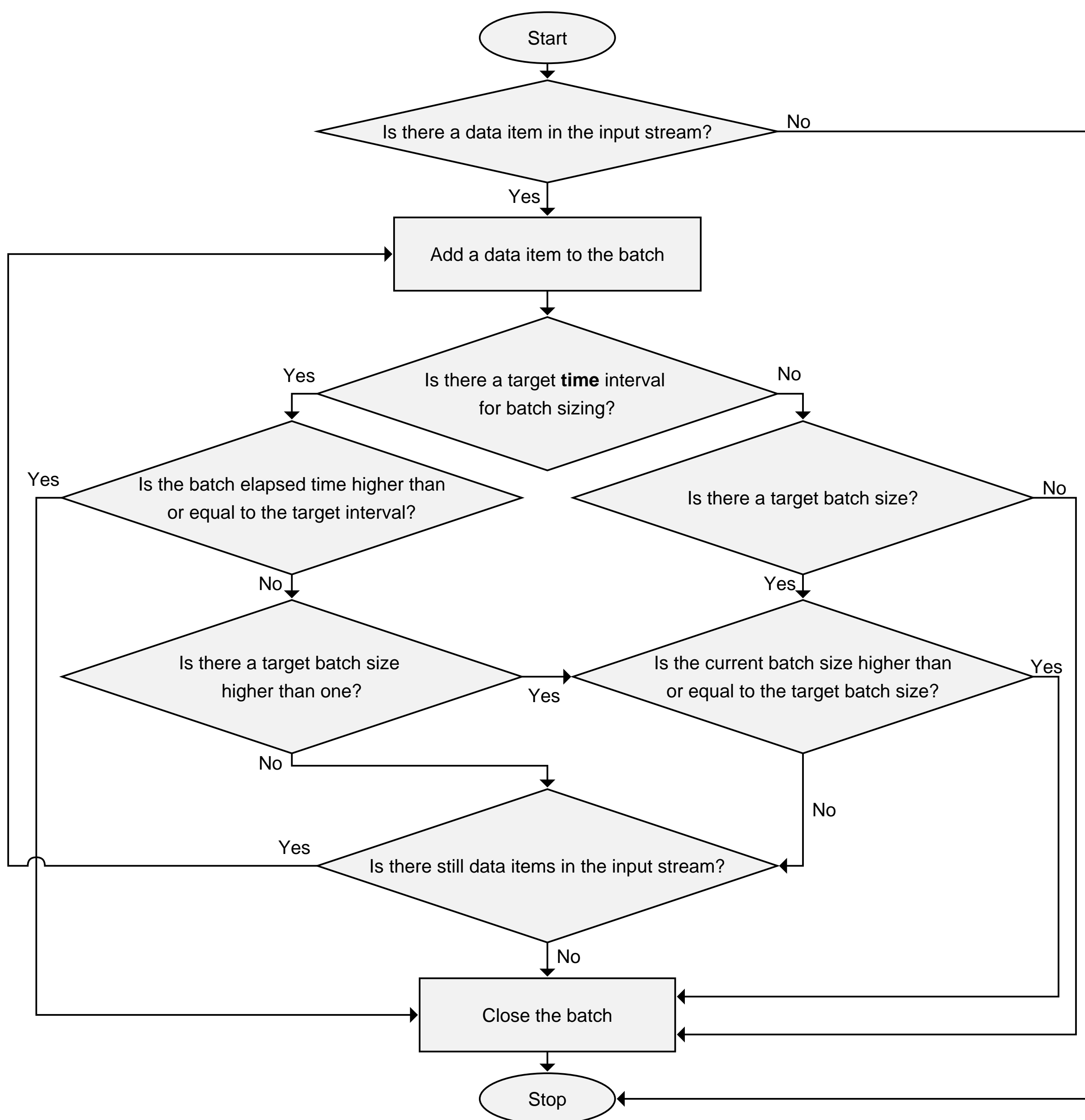


Figure 7.2: SPBench's batch sizing flowchart.

We increased the batch size from 1 to 10 in this case. SPBench also supports limiting the size of batches by the number of items or time windows. But analyzing micro-batches by time window adds too many other variables and would open up the scope of this work. Therefore, we use micro-batches limited by the number of items.

7.4.2 Experimental Results

In [GGSF22a], we evaluated the impact of micro-batch size on the performance of stream processing applications on multi-core systems. In the experiments we performed in

that work, increasing the batch size could incur throughput increases, besides the expected increase in latency. However, we later identified inconsistencies between the SPBench metrics system and the new batching features that were added to that work. This problem was causing the throughput to be overestimated. Later, in [GGSF23], we fixed the metrics problem for batching. Therefore, we performed more experiments to check to what extent the advantages of batching discussed in this Chapter's motivation apply to real-world C++ SP applications on multi-core systems.

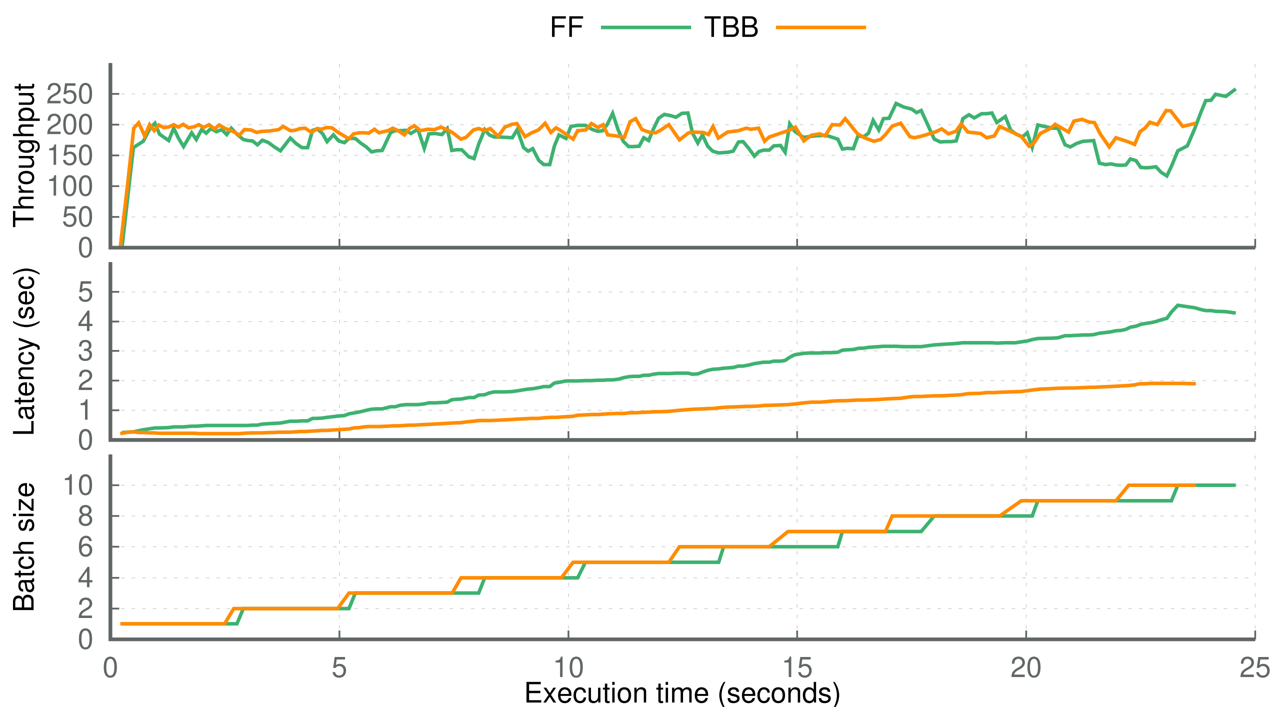


Figure 7.3: Throughput and latency results of Bzip2 benchmark implemented as a farm (40 workers) with TBB and FastFlow, increasing the batch size dynamically from 1 to 10 along the execution.

In the micro-batch experiments, we used two strategies. In the first one, we vary the batch size dynamically at execution time from 1 to 10. The goal was to validate this feature we added to SPBench and observe the impact of the batch size on the performance of the benchmarks at execution time. Figure 7.3 shows the throughput and latency results of the Bzip2 application on Intel Xeon Silver 4210. We can see that the FastFlow implementation took longer to run. Also, FastFlow has a less stable instantaneous throughput than TBB, and the increase in batch size expels the latency difference between PPIs. The increase in batch size apparently did not influence the application's throughput.

Figure 7.4 presents the results of the Ferret benchmark. The same discussion made in the case of Bzip2 applies here. FastFlow has a more unstable throughput and increases latency at a higher rate than TBB as the batch size increases. In the third graph in Figure 7.4, regarding batch size, there is a difference from Bzip2. Ferret is an application that does not require item sorting. The batch size monitoring in SPBench is performed in the last stage of the pipeline. Therefore, the batch size spikes represent items arriving

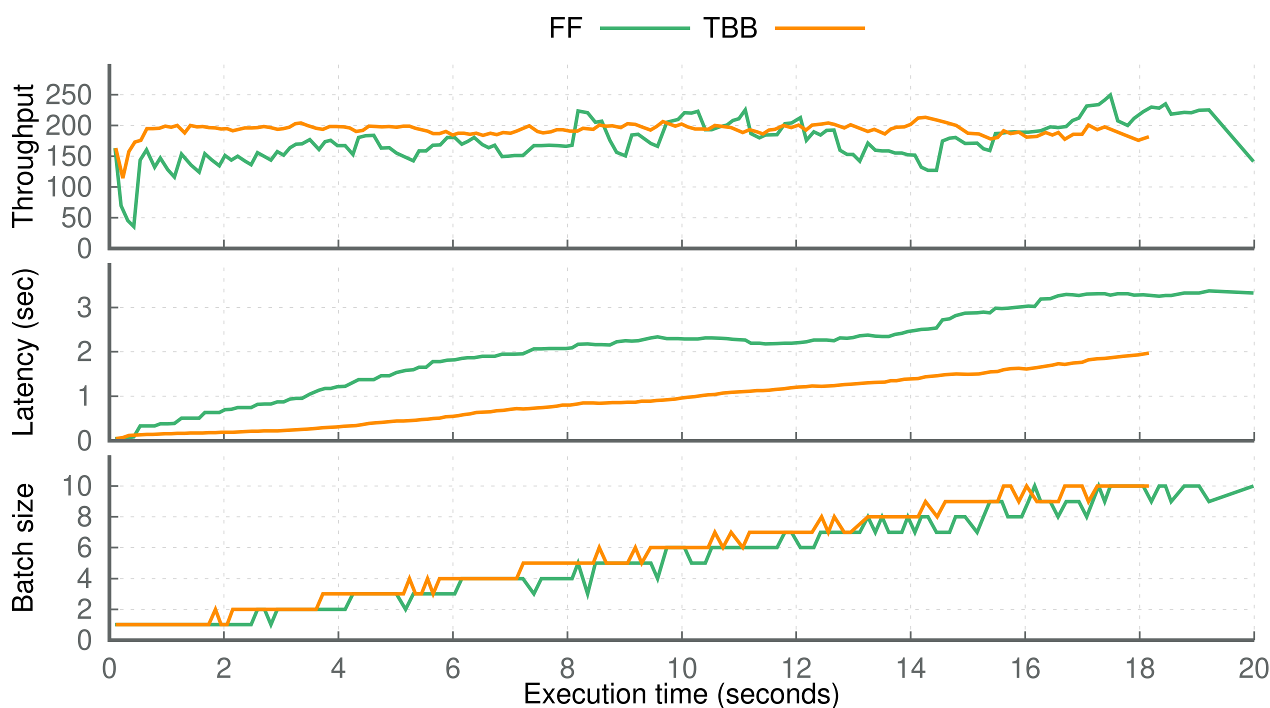
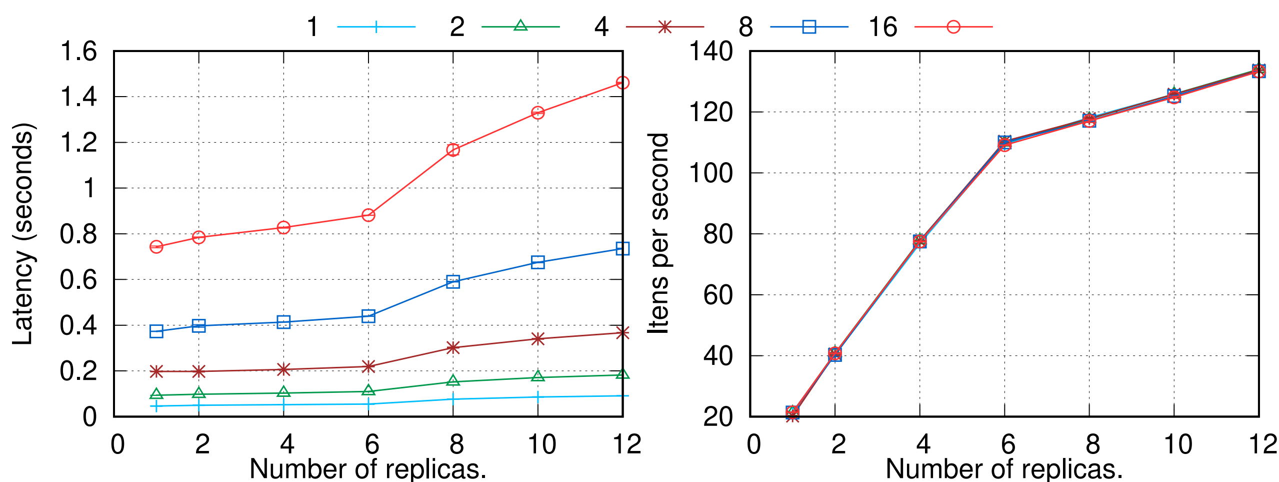


Figure 7.4: Throughput and latency results of Ferret implemented as a pipeline of farms (maximum of 40 workers per farm) with TBB and FastFlow, increasing the batch size dynamically from 1 to 10 along the execution.

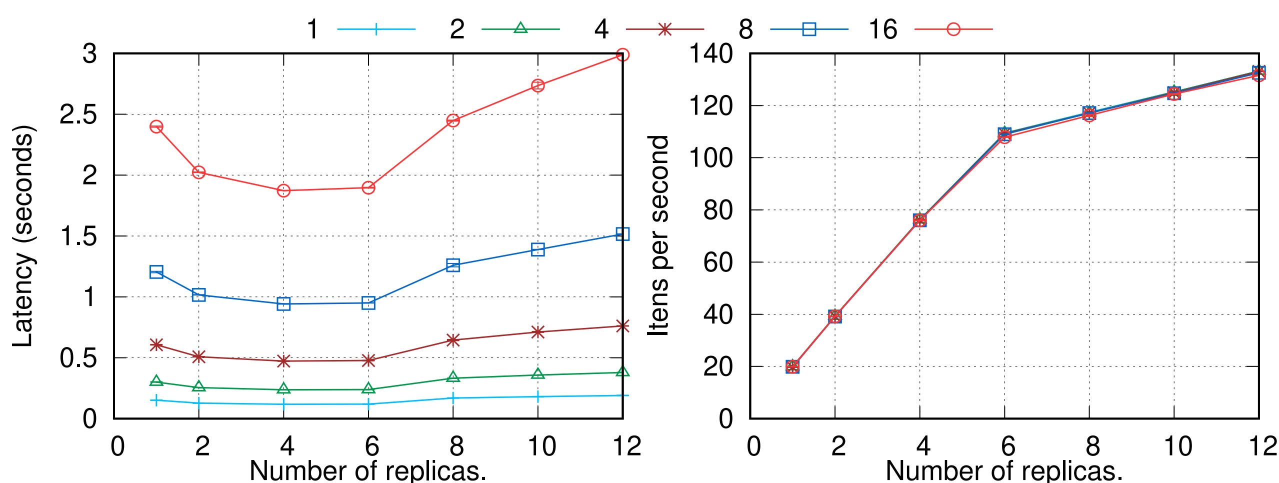
out of order in the sink. It shows that our FastFlow implementation causes more items to arrive unordered at the sink than TBB. This factor can also impact the latency of FastFlow in applications that require sorting.

We also investigate how batch size impacts application performance when varying degrees of parallelism. Here, instead of changing the batch size dynamically, we set it statically at the beginning of the execution. Therefore, besides the number of farm workers, we also vary the micro-batch size from 1 (no batch) to 16 items per batch. One of our goals with these experiments is to see if using batches could alleviate the cost of item ordering in applications with ordering constraints. After all, it is known that this cost can impact latency and throughput [GHDF18b, PRR19]. Therefore, if we decrease the number of items in the stream using batches of ordered items, there may be some performance gain in applications with issues with item cluttering.

Figure 7.5 presents the results of the micro-batch experiments varying the degree of parallelism in the Lane Detection benchmarks using the AMD computer. Lane Detection is a good test case for us to evaluate since it has a high throughput (more chances of items arriving at sink out of order), and the computational cost of each frame varies greater than Bzip2 or Face Recognizer, as seen in Figure 5.2. This further increases the clutter of items in the stream, and this application requires the frames to be ordered in the output video. The results show that for both TBB and FastFlow, the latency increases proportionally to



(a) Threading Building Blocks

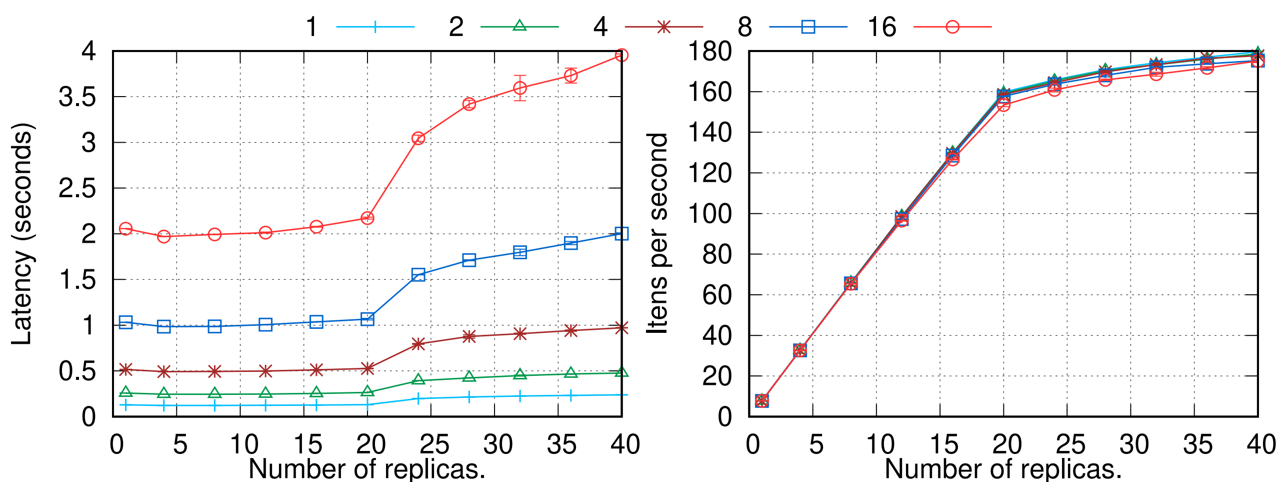


(b) FastFlow

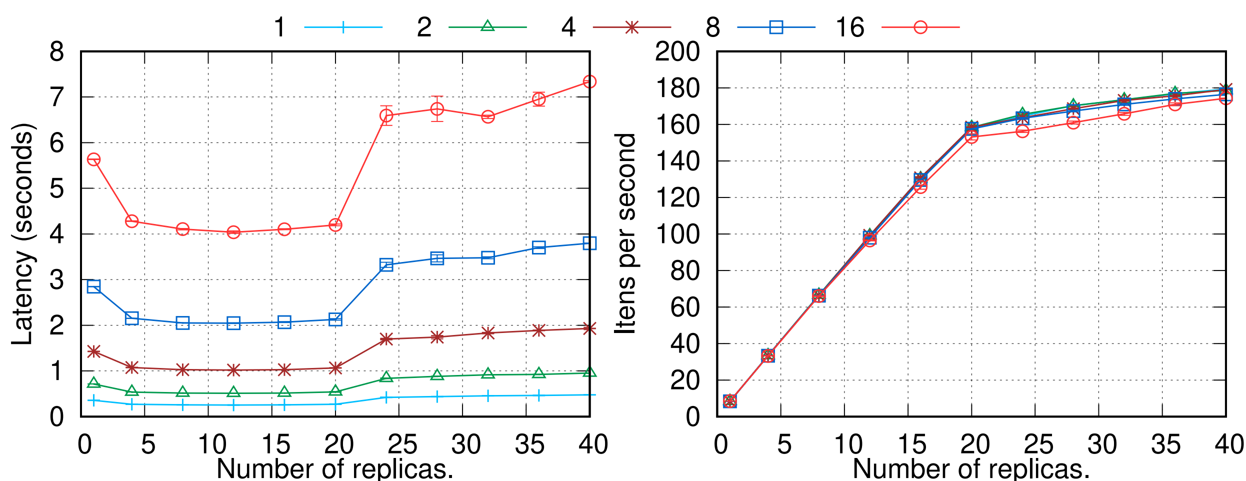
Figure 7.5: Latency and throughput results for Lane Detection with multiple parallelism degrees and statically set micro-batch sizes (AMD Ryzen 5 5600X).

the batch size. However, there is no significant throughput gain at all. We can conclude that batch played no role here.

We repeated the experiment as shown in Figure 7.5, but we used the computer with the Intel Xeon Silver 4210 this time. This computer has a total of 40 cores, and we can use more workers. The increase in workers causes more items to clutter because more concurrent threads are writing to the output queue. These results are shown in Figure 7.6. The latency behavior remained the same in this case. However, the throughput got worse when increasing the batch size at a higher degree of parallelism. The time required to fill each batch may prevent the source from meeting the demand of the subsequent stages. There is a minimal increase in FastFlow throughput with batch size 2, but it is minimal and should be disregarded. We performed the same experiments with the other benchmarks. However, the results were similar and did not lead to different conclusions, so we omitted them. So we can conclude that despite the possible advantages of using micro-batch in multi-core (Section 7.1), we could not identify this in our experiments. In the future,



(a) Threading Building Blocks



(b) FastFlow

Figure 7.6: Latency and throughput results for Lane Detection with multiple parallelism degrees and statically set micro-batch sizes (Intel Xeon Silver 4210).

we intend to test the SPBench in scenarios where batching is more impactful, such as distributed and heterogeneous systems or with specific benchmark applications.

7.5 Chapter Summary

In this chapter, we extended the SPBench benchmarking framework to support micro-batching. With the help of the extended framework, we analyzed the impact of micro-batch on real-world stream processing applications with different PPIs. We were able to create several workloads with some strategies that could change dynamically at execution time. We also tested micro-batching configurations under different levels of parallelism.

The micro-batching has the potential to improve performance in multi-cores by enabling software pipelining, vectorization, and amortizing costs such as operator-firing,

communication, data sorting, warm-up (e.g., for the instruction cache), data scheduling, and others. As discussed in Section 7.2, some related work managed to extract performance improvement in very specific scenarios, with synthetic benchmarks However, our experiments showed no benefit from micro-batches on the performance of the real-world benchmarks we used. Latency increased as expected, but it had no positive impact on throughput. In future work, we may assess the impact of micro-batching in the resource usage of SP applications. Also, another future work goes towards extending SPBench with support for heterogeneous and distributed architectures, where batching mechanisms may be more advantageous.

8. CONCLUSION

This work discusses the challenges and limitations of benchmarking C++ stream processing and analyzing parallel programming interfaces (PPIs) that leverage stream parallelism. Therefore, we provide a framework to ease the benchmark creation and benchmarking process in stream processing and a comprehensive analysis of the PPIs in this context.

We understand that benchmarking in C++ stream processing is a relevant research area. C++ stream processing is expanding due to the growing demand for low-latency applications and the increasing prevalence of powerful multi-core processors. This growth is also being driven by the Internet of Things (IoT), where Java-based languages and frameworks used in the industry add performance and resource usage overheads due to the JVM. New solutions and technologies are being developed to improve the available PPIs for exploring stream parallelism. These solutions aim to make PPIs more user-friendly, dynamic, domain-comprehensive, extensible, and portable. However, designing, evaluating, and validating such solutions is often complex.

To address these challenges, we developed SPBench, a framework that addresses the entire benchmarking process. Instead of following the classical approach of implementing a benchmark suite, we developed SPBench, a framework to facilitate the creation of benchmarks for stream processing. Using SPBench, we built a parallel benchmark suite using the leading state-of-the-art PPIs in this context. We identified that a more comprehensive analysis of these PPIs was lacking in the literature, considering more modern requirements such as latency, resource usage, and programmability. Thus, we evaluated all SPBench benchmarks considering these and other aspects, such as performing the PPIs under different data frequency levels and using micro-batching size.

Although SPBench mainly concerns C++ stream processing applications and PPIs that leverage stream parallelism, the framework concept is not limited by programming language or specific application domain. Below, we summarize the research goals of this thesis, discuss how we addressed them, and discuss the resulting technical and scientific contributions to the community.

The first research goal of this work involves easing the benchmarking of C++ parallel stream processing. As discussed in Section 3.2, we argue that only putting together a set of parallel stream processing applications, which would be the classical approach, is insufficient to achieve this goal. As we showed in Sections 3.3 and 5.2, most research in this area uses benchmarks that tend to be poorly parameterizable, hard to use and to extend to new PPIs, and often not publicly available.

In our work, we use another approach for building benchmarks. We created an API (Subsection 3.2.1) that establishes a way of developing stream processing applications

in a standard, synthesized, modular, reconfigurable, and assessable way. Then, we take a set of SP applications already used for benchmarking purposes in the literature and integrate them into this API. After this, we developed a command-line interface (CLI) (Subsection 3.2.6) to enhance the usability of the benchmarks. What we described to this point represents the SPBench framework's kernel: a set of sequential benchmarks from the stream processing domain implemented with a high-level abstraction API and a CLI. While the framework does not provide means to explore parallelism, it is designed to be used with PPIs enabling stream parallelism. This way, we selected some of the most popular PPIs that support C++ stream parallelism and used them to create a benchmark suite.

Pursuing the first research goal resulted in technical and scientific contributions to the community. As a technical contribution is the SPBench benchmarking framework, a tool that facilitates the creation of benchmarks for stream processing plus a parallel benchmark suite. SPBench benefits the scientific community by offering a range of highly parameterizable and reconfigurable real-world C++ parallel stream processing benchmarks. The scientific contribution is the framework conceptual idea, which adds a new perspective on building SP benchmarks in a more user-friendly way (Figure 3.3) if compared to the traditional benchmarks (Figure 3.2). This new perspective may help leverage the development of more accessible benchmarks in the future.

Our second research goal was to speed up and simplify the research for parallel stream processing by providing highly parameterizable benchmarks with self-built representative mechanisms, such as batching, data stream frequency management, and real-time performance metrics. This research goal is more specific than the first one. This one involves helping to speed up research in parallel stream processing by providing benchmarks with mechanisms and metrics that can be difficult to implement, error-prone, and time-consuming for users. It resulted in several technical contributions. Below we summarize the main ones:

- The API that standardizes the source code of the benchmarks and makes the parallel code highly portable among them, helping to speed up the creation of benchmarks with different parallelism strategies and other solutions.
- A CLI that provides several key features which allow users to manipulate and execute multiple benchmarks and commands that can be combined to enable extra functionality, dispensing the need to run scripts and parse the results.
- A time-window-based metrics system to measure average latency and throughput over short intervals. These metrics can be dynamically obtained from within the code, a valuable feature for self-adaptive performance algorithms that require accurate real-time performance feedback.

- Benchmarks with mechanisms that allow users to control the frequency of the data stream and algorithms that allow it to be changed according to the frequency patterns most commonly used for benchmarking SP systems in the literature.
- SP benchmarks with native batching support. While not advantageous in our experiments with multi-cores, it can be better leveraged on heterogeneous and distributed architectures in future work.
- A thorough user documentation with detailed information about the benchmarks, workloads, and tips on how to get the best out of SPBench.

As a scientific contribution, one could mention the algorithms behind the mechanisms, such as those that generate frequency patterns. However, the most important scientific contribution was to enable the analyses presented in Chapters 5, 6, and 7. It brings us to the third research goal: Conduct a comprehensive evaluation and comparison of the state-of-art PPIs that support parallel stream processing in C++.

The lack of comprehensive evaluation of C++ stream processing PPIs motivated this third research goal. Despite the increasing use of C++ for SP, given mainly by the demand for low-latency and resource-optimizing applications, we observed that most of the analysis in the literature did not address these aspects. Therefore, we did a more comprehensive analysis encompassing throughput, latency, memory usage, and programmability/productivity of the main PPIs that leverage C++ stream parallelism. Pursuing this research goal also motivated developing and improving the mechanisms listed above. The main technical contribution was the suite of parallel benchmarks, including low abstraction PPIs such as ISO C++ threads and OpenMP, structured parallel programming PPIs such as FastFlow and TBB, and high abstraction PPIs such as SPar and GrPPI.

The main scientific contribution of the third research goal was the analysis itself, which brought new insights over the PPIs. Overall, TBB showed similar throughput as FastFlow when using a single farm. Nevertheless, TBB's more dynamic task scheduler fits very well with the kind of applications we tested, so it was able to show better latency in most cases. Regarding a pipeline of farms with the Ferret benchmark, although FastFlow can slightly outperform TBB throughput with our over-subscription strategy, its latency was about ten times higher than TBB. However, FastFlow allows for specific parallelism configurations that deliver latency similar to TBB with low impact on throughput and without oversubscribing threads to the system, as presented in Subsection 5.5.5.

In GrPPI, all backends underperformed at some point in the test scenarios. Although GrPPI has a policy of not prioritizing fine-tuning to improve programmability [dRADFG17], some poor performance resulted from GrPPI's internal implementation rather than a lack of fine-tuning mechanisms. The memory usage results showed the impact of the lack of fine-tuning on system resource utilization and that high memory usage is commonly associated with high latency. The exception is TBB, which, despite low latency, exhibited a

memory usage overhead in some cases, likely due to the larger number of object allocations that its dynamic execution model can cause. The performance Programmability results showed that high-level parallelism abstractions come at the expense of performance in more complex parallel patterns, leading to limitations in addressing specific fine-tuning mechanisms. The data frequency results showed that FastFlow outperformed TBB throughput in frequency-varying scenarios and simpler parallelism compositions like single farms. Our analysis with micro-batching showed no performance benefit of using it in multicore with the kind of workloads we tested.

In this way, we achieve our research goals of improving the C++ parallel stream processing benchmarking space, providing a means to facilitate and accelerate research in this area, and conducting a comprehensive evaluation of PPIs in this context. We hope the results and our analysis can guide improvements and future development for the PPIs we evaluated. Especially FastFlow, GrPPI, and SPar, which are still relatively new PPIs used most in academia, unlike TBB and OpenMP, which are already widely used in industry.

8.1 Limitations and Future Work

Our work focused more on the framework design and development and less on developing the application set. We argue that investing our effort in the framework brought more relevant contributions to this work than a large application set would have since adding new applications is manual programming work that would result mainly in technical contributions. However, now that the framework has reached a certain maturity, it would be important to invest time in supporting new applications in the future. New applications in the suite could bring new insights. For instance, although Fraud Detection has a stateful operator, we did not include any traditional stream processing application with stateful operators. It would be valuable to evaluate how each PPI handles states and how this would impact their performance, resource usage, and programmability.

Although the main goal of our work is to facilitate the whole process of benchmarking in parallel stream processing, we have yet to be able to evaluate this aspect directly. It would be necessary to perform experiments with programmers to evaluate how much easier it is to use SPBench compared to traditional benchmarking methods. Although challenging, this aspect can be better investigated in the future.

We have not done a detailed study of the performance impact caused by the API, the workload management mechanisms, and the use of performance metrics. Such evaluation is challenging, as it is expected that the use of mechanisms such as batching and data frequency will indeed perform differently. To evaluate the impact of API abstractions, one would need to implement the applications without using SPBench and add similar metrics for a fair comparison and evaluate the difference in performance. Of course, using very

fine-grained monitoring metrics will cause some overhead. Future work may investigate these issues, which may be related to the difference in performance between SPBench Fraud Detection and StreamBenchmarks Fraud Detection, as discussed in Subsection 5.5.6.

As discussed in Subsection 2.1.3, [MRR12, Tor19] claim that PPIs ideally should balance three properties: programmability/productivity, portability, and performance. Although SPBench has as one of its primary goals to evaluate PPIs, we were not able to thoroughly evaluate the portability of PPIs because we have not yet evaluated SPBench's ability to address heterogeneous and distributed architectures. Also, in these architectures, the use of micro-batches could add performance advantages by decreasing the data communication overhead [TTMD22]. Moreover, disregarding the experiments in Subsection 5.5.2, where we evaluated the most popular PPIs on three architectures, all other experiments were performed on the same computer. We also do not address embedded architectures, where there is great motivation for using C++ for parallel stream processing.

We created very detailed documentation for the users[1]. However, we still need to improve the documentation for developers who want to collaborate with the SPBench, like an API manual and a guide on adding support for new applications. Although SPBench allows instantiating multiple sources in the benchmarks, this feature was not evaluated. We need to find use cases worth exploring in the current framework configuration. Use cases may arise with the addition of new applications in future work.

This work has focused on C++, where we found the most significant need for benchmarking solutions. The framework concept, however, can be extended to other languages. For that, it would be necessary to translate the mechanisms to a new language, implement the apps in the new language according to the SPBench API, and make minor adjustments in the CLI. Although we have designed the framework to have the possibility of sending/receiving data streams over the network, we did not implement this feature due to its complexity and the scope of this work. We envisioned it as an independent system that could run on another thread or a different computer. This system would have to incorporate data frequency mechanisms, and communication could be done using sockets in C++.

Regarding running SPBench on heterogeneous or distributed architectures, although some functionality becomes limited, adding support for these architectures is possible. One of the biggest problems with distributed processing is data serialization. Since the SPBench adds more complex structures on top of the data in the stream, it is challenging to serialize this data optimally, such as using MPI-specific serialization methods. However, SPBench could provide ready-made serialization/deserialization methods for users. It could be done using libraries like Cereal [GV13], as suggested in a study that proposes a FastFlow extension for distributed architectures [TTMD22]. It would also be interesting to evaluate stream processing on emergent ARM architectures for HPC and also on RISC-V architectures.

[1] https://spbench-doc.rtfd.io/

REFERENCES

[ABD+16] Agrawal, D.; Butt, A.; Doshi, K.; Larriba-Pey, J.-L.; Li, M.; Reiss, F. R.; Raab, F.; Schiefer, B.; Suzumura, T.; Xia, Y. “Sparkbench – a spark performance testing suite”. In: Proceedings of the International Conference on Performance Evaluation and Benchmarking: Traditional to Big Data to Internet of Things, Nambiar, R.; Poess, M. (Editors), 2016, pp. 26–44.

[ADKT17a] Aldinucci, M.; Danelutto, M.; Kilpatrick, P.; Torquati, M. “Fastflow: high-level and efficient streaming on multicore”. Hoboken, USA: John Wiley & Sons, Ltd, 2017, chap. 13, pp. 261–280.

[ADKT17b] Aldinucci, M.; Danelutto, M.; Kilpatrick, P.; Torquati, M. “Fastflow: high-level and efficient streaming on multicore”. John Wiley & Sons, Ltd, 2017, chap. 13, pp. 261–280, https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781119332015.ch13.

[AGS+21] Andrade, G.; Griebler, D.; Santos, R.; Danelutto, M.; Fernandes, L. G. “Assessing coding metrics for parallel programming of stream processing programs on multi-cores”. In: Proceedings of the International Conference on Software Engineering and Advanced Applications, 2021, pp. 291–295.

[AGS+22] Andrade, G.; Griebler, D.; Santos, R.; Kessler, C.; Ernstsson, A.; Fernandes, L. G. “Analyzing programming effort model accuracy of high-level parallel programs for stream processing”. In: Proceedings of the International Conference on Software Engineering and Advanced Applications, 2022, pp. 229–232.

[AGSF23] Andrade, G.; Griebler, D.; Santos, R.; Fernandes, L. G. “A parallel programming assessment for stream processing applications on multi-core systems”, *Computer Standards & Interfaces*, vol. 84, mar 2023, pp. 103691.

[AGT14] Andrade, H. C.; Gedik, B.; Turaga, D. S. “Fundamentals of stream processing: application design, systems, and analytics”. Cambridge University Press, 2014, 558p.

[AHN+20] Amanullah, M. A.; Habeeb, R. A. A.; Nasaruddin, F. H.; Gani, A.; Ahmed, E.; Nainar, A. S. M.; Akim, N. M.; Imran, M. “Deep learning and big data technologies for IoT security”, *Computer Communications*, vol. 151, feb 2020, pp. 495–517.

[Amd67] Amdahl, G. M. "Validity of the single processor approach to achieving large scale computing capabilities". In: Proceedings of the International Spring Joint Computer Conference, 1967, pp. 483–485.

[AMDA20] Abdelhamid, A. S.; Mahmood, A. R.; Daghistani, A.; Aref, W. G. "Prompt: Dynamic data-partitioning for distributed micro-batch stream processing systems". In: Proceedings of the International Conference on Management of Data, 2020, pp. 2455–2469.

[APTE21] Arkian, H.; Pierre, G.; Tordsson, J.; Elmroth, E. "Model-based stream processing auto-scaling in geo-distributed environments". In: Proceedings of the International Conference on Computer Communications and Networks (ICCCN), 2021, pp. 1–10.

[Aru13] Arubas, E. "Face detection and recognition (theory and practice)". Source: http://eyalarubas.com/face-detection-and-recognition.html, May 2021.

[ASAP17] Alevizos, E.; Skarlatidis, A.; Artikis, A.; Paliouras, G. "Probabilistic complex event recognition: A survey", *ACM Comput. Surv.*, vol. 50–5, Sep. 2017.

[ATM09] Aldinucci, M.; Torquati, M.; Meneghin, M. "Fastflow: Efficient parallel streaming applications on multi-core", Technical Report, Universita di Pisa, Dipartimento di Informatica, 2009, 25p.

[BAJ+16] Bingmann, T.; Axtmann, M.; Jöbstl, E.; Lamm, S.; Nguyen, H. C.; Noe, A.; Schlag, S.; Stumpp, M.; Sturm, T.; Sanders, P. "Thrill: High-performance algorithmic distributed batch data processing with C++". In: Proceedings of the International Conference on Big Data, 2016, pp. 172–183.

[BBB+91] Bailey, D. H.; Barszcz, E.; Barton, J. T.; Browning, D. S.; Carter, R. L.; Dagum, L.; Fatoohi, R. A.; Frederickson, P. O.; Lasinski, T. A.; Schreiber, R. S.; et al.. "The NAS Parallel Benchmarks", *International Journal of High Performance Computing Applications*, vol. 5–3, sep 1991, pp. 63–73.

[BBD+14] Ballard, C.; Brandt, O.; Devaraju, B.; Farrell, D.; Foster, K.; Howard, C.; Nicholls, P.; Pasricha, A.; Rea, R.; Schulz, N.; et al.. "IBM infosphere streams: accelerating deployments with analytic accelerators". IBM Redbooks, 2014, 556p.

[BCC+13] Bainomugisha, E.; Carreton, A. L.; Cutsem, T. v.; Mostinckx, S.; Meuter, W. d. "A survey on reactive programming", *ACM Comput. Surv.*, vol. 45–4, Aug. 2013.

[BGM+20] Bordin, M. V.; Griebler, D.; Mencagli, G.; Geyer, C. F. R.; Fernandes, L. G. "DSPBench: a suite of benchmark applications for distributed data stream processing systems", *IEEE Access*, vol. 8–na, dec 2020, pp. 222900–222917.

[BJB+20] Brown, C.; Janjic, V.; Barwell, A. D.; Garcia, J. D.; MacKenzie, K. "Refactoring GrPPI: generic refactoring for generic parallelism in C++", *International Journal of Parallel Programming*, vol. 48–4, jul 2020, pp. 603–625.

[BKSL08] Bienia, C.; Kumar, S.; Singh, J. P.; Li, K. "The PARSEC benchmark suite: Characterization and architectural implications". In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, 2008, pp. 72–81.

[BTO13] Balkesen, C.; Tatbul, N.; Özsu, M. T. "Adaptive input admission and management for parallel stream processing". In: Proceedings of the International Conference on Distributed Event-Based Systems, 2013, pp. 15–26.

[Car14] Carkci, M. "Dataflow and reactive programming systems: a practical guide". CreateSpace Independent Publishing Platform, 2014, 150p.

[CBP+17] Cranmer, M. D.; Barsdell, B. R.; Price, D. C.; Dowell, J.; Garsden, H.; Dike, V.; Eftekhari, T.; Hegedus, A. M.; Malins, J.; Obenberger, K. S.; et al.. "Bifrost: A python/C++ framework for high-throughput stream processing in astronomy", *Journal of Astronomical Instrumentation*, vol. 6–04, jan 2017, pp. 1750007.

[CcR+03] Carney, D.; Çetintemel, U.; Rasin, A.; Zdonik, S.; Cherniack, M.; Stonebraker, M. "Operator scheduling in a data stream manager". In: Proceedings of the International Conference on Very Large Data Bases, Freytag, J.-C.; Lockemann, P.; Abiteboul, S.; Carey, M.; Selinger, P.; Heuer, A. (Editors), 2003, pp. 838–849.

[CHGL22] Chiu, C.-H.; Huang, T.-W.; Guo, Z.; Lin, Y. "Pipeflow: An efficient task-parallel pipeline programming framework using modern C++". Preprint, Source: https://arxiv.org/abs/2202.00717, 2022.

[CN06] Ching, W.-K.; Ng, M. K. "Markov chains". Springer, 2006, 400p.

[Col89] Cole, M. I. "Algorithmic skeletons: structured management of parallel computation". Pitman London, 1989, 137p.

[Col04] Cole, M. "Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming", *Parallel computing*, vol. 30–3, mar 2004, pp. 389–406.

[CYH21] Chu, Z.; Yu, J.; Hamdulla, A. “Throughput prediction based on extratree for stream processing tasks”, *Computer Science and Information Systems*, vol. 18–1, jan 2021, pp. 1–22.

[Dav18] Davis, A. L. “Reactive streams in Java: concurrency with RxJava, Reactor, and Akka Streams”. USA: Apress, 2018, 1st ed., 159p.

[DBL+19] Djenouri, Y.; Belhadi, A.; Lin, J. C.; Djenouri, D.; Cano, A. “A survey on urban traffic anomalies detection algorithms”, *IEEE Access*, vol. 7, 2019, pp. 12192–12205.

[dDFG18] del Rio Astorga, D.; Dolz, M. F.; Fernández, J.; García, J. D. “Paving the way towards high-level parallel pattern interfaces for data stream processing”, *Future Generation Computer Systems*, vol. 87, aug 2018, pp. 228–241.

[DDMMT15] Danelutto, M.; De Matteis, T.; Mencagli, G.; Torquati, M. “Parallelizing high-frequency trading applications by using C++11 attributes”. In: Proceedings of the International Conference on Trust, Security and Privacy in Computing and Communications, 2015, pp. 140–147.

[DDMMT18] Danelutto, M.; De Matteis, T.; Mencagli, G.; Torquati, M. “Data stream processing via code annotations”, *The Journal of Supercomputing*, vol. 74–11, jun 2018, pp. 5659–5673.

[Den74] Dennis, J. B. “First version of a data flow procedure language”. In: Proceedings of the International Programming Symposium, 1974, pp. 362–376.

[DL15] Dietrich, C.; Lohmann, D. “The dataref versuchung: Saving time through better internal repeatability”, *SIGOPS Oper. Syst. Rev.*, vol. 49–1, jan 2015, pp. 51–60.

[DLP03] Dongarra, J. J.; Luszczek, P.; Petitet, A. “The linpack benchmark: past, present and future”, *Concurrency and Computation: practice and experience*, vol. 15–9, mar 2003, pp. 803–820.

[DM98] Dagum, L.; Menon, R. “OpenMP: an industry standard api for shared-memory programming”, *IEEE Computational Science and Engineering*, vol. 5–1, mar 1998, pp. 46–55.

[DMM16] De Matteis, T.; Mencagli, G. “Keep calm and react with foresight: Strategies for low-latency and energy-efficient elastic data stream processing”, *SIGPLAN Not.*, vol. 51–8, Feb. 2016.

[DMM17] De Matteis, T.; Mencagli, G. “Parallel patterns for window-based stateful operators on data streams: an algorithmic skeleton approach”, *International Journal of Parallel Programming*, vol. 45–2, mar 2017, pp. 382–401.

[DPCCM13] Difallah, D. E.; Pavlo, A.; Curino, C.; Cudre-Mauroux, P. “OLTP-Bench: An extensible testbed for benchmarking relational databases”, *Proc. VLDB Endow.*, vol. 7–4, dec 2013, pp. 277–288.

[dRADFG17] del Rio Astorga, D.; Dolz, M. F.; Fernández, J.; García, J. D. “A generic parallel pattern interface for stream and data processing”, *Concurrency and Computation: Practice and Experience*, vol. 29–24, may 2017, pp. e4175.

[DSDMT+17] De Sensi, D.; De Matteis, T.; Torquati, M.; Mencagli, G.; Danelutto, M. “Bringing parallel patterns out of the corner: The p3arsec benchmark suite”, *ACM Trans. Archit. Code Optim.*, vol. 14–4, Oct. 2017.

[DSMV+20] De Souza, P. R. R.; Matteussi, K. J.; Veith, A. D. S.; Zanchetta, B. F.; Leithardt, V. R. Q.; Murciego, A. L.; De Freitas, E. P.; Anjos, J. C. S. D.; Geyer, C. F. R. “Boosting big data streaming applications in clouds with burstflow”, *IEEE Access*, vol. 8, dec 2020, pp. 219124–219136.

[DSTD16] De Sensi, D.; Torquati, M.; Danelutto, M. “A reconfiguration algorithm for power-aware parallel applications”, *ACM Trans. Archit. Code Optim.*, vol. 13–4, Dec. 2016.

[DZSS14] Das, T.; Zhong, Y.; Stoica, I.; Shenker, S. “Adaptive stream processing using dynamic batch sizing”. In: Proceedings of the International Symposium on Cloud Computing, 2014, pp. 1–13.

[Ell09] Elliott, C. M. “Push-pull functional reactive programming”. In: Proceedings of the International Symposium on Haskell, 2009, pp. 25–36.

[FK15] Fleisch, D.; Kinnaman, L. “A student’s guide to waves”. Cambridge, UK: Cambridge University Press, 2015, 230p.

[Fou20] Foundation, S. C. “C++11 standard library extensions”. Source: https://isocpp.org/wiki/faq/cpp11-library-concurrency, Jan 2021.

[FT16] Friedman, E.; Tzoumas, K. “Introduction to Apache Flink: stream processing for real time and beyond”. " O’Reilly Media, Inc.", 2016, 109p.

[GAA+20] Giatrakos, N.; Alevizos, E.; Artikis, A.; Deligiannakis, A.; Garofalakis, M. “Complex event recognition in the big data era: a survey”, *The VLDB Journal*, vol. 29–1, jul 2020, pp. 313–352.

[Gam95] Gamma, E. "Design patterns: elements of reusable object-oriented software". Pearson Education India, 1995, 417p.

[GBdRAGC19] Garcia-Blas, J.; del Rio Astorga, D.; García, J. D.; Carretero, J. "Exploiting stream parallelism of mri reconstruction using GrPPI over multiple backends". In: Proceedings of the International Symposium on Cluster, Cloud and Grid Computing (CCGRID), 2019, pp. 631–637.

[GdRA+20] Garcia, J. D.; del Rio, D.; Aldinucci, M.; Tordini, F.; Danelutto, M.; Mencagli, G.; Torquati, M. "Challenging the abstraction penalty in parallel patterns libraries", *The Journal of Supercomputing*, vol. 76–7, jul 2020, pp. 5139–5159.

[GDTF17] Griebler, D.; Danelutto, M.; Torquati, M.; Fernandes, L. G. "SPar: A DSL for high-level and productive stream parallelism", *Parallel Processing Letters*, vol. 27–01, mar 2017, pp. 17.

[GGSF21] Garcia, A. M.; Griebler, D.; Schepke, C.; Fernandes, L. G. "Introducing a stream processing framework for assessing parallel programming interfaces". In: Proceedings of the International Conference on Parallel, Distributed and Network-Based Processing, 2021, pp. 84–88.

[GGSF22a] Garcia, A. M.; Griebler, D.; Schepke, C.; Fernandes, L. G. "Evaluating microbatch and data frequency for stream processing applications on multi-cores". In: Proceedings of the International Conference on Parallel, Distributed and Network-Based Processing, 2022, pp. 10–17.

[GGSF22b] Garcia, A. M.; Griebler, D.; Schepke, C.; Fernandes, L. G. "SPBench: a framework for creating benchmarks of stream processing applications", *Computing*, vol. 1, jan 2022.

[GGSF23] Garcia, A. M.; Griebler, D.; Schepke, C.; Fernandes, L. G. "Micro-batch and data frequency for stream processing on multi-cores", *The Journal of Supercomputing*, vol. In press–In press, jan 2023, pp. 1–39.

[GHDF17] Griebler, D.; Hoffmann, R. B.; Danelutto, M.; Fernandes, L. G. "Higher-level parallelism abstractions for video applications with SPar". In: Proceedings of the International Conference on Parallel Computing, 2017, pp. 698–707.

[GHDF18a] Griebler, D.; Hoffmann, R. B.; Danelutto, M.; Fernandes, L. G. "High-level and productive stream parallelism for dedup, ferret, and bzip2", *International Journal of Parallel Programming*, vol. 47–1, feb 2018, pp. 253–271.

[GHDF18b] Griebler, D.; Hoffmann, R. B.; Danelutto, M.; Fernandes, L. G. "Stream parallelism with ordered data constraints on multi-core systems", *Journal of Supercomputing*, vol. 75–8, jul 2018, pp. 4042–4061.

[GJ21] Gordon, A.; Jones, S. P. "Lambda: The ultimate excel worksheet function". Source: https://www.microsoft.com/en-us/research/blog/lambda-the-ultimatae-excel-worksheet-function/, Jan 2021.

[GPRD19] Gomes, E. H.; Plentz, P. D.; Rolt, C. R. D.; Dantas, M. A. "A survey on data stream, big data and real-time", *International Journal of Networking and Virtual Organisations*, vol. 20–2, 2019, pp. 143–167.

[Gra92] Gray, J. "Benchmark handbook: for database and transaction processing systems". Morgan Kaufmann Publishers Inc., 1992, 334p.

[Gri16a] Griebler, D. "Domain-specific language & support tool for high-level stream parallelism", Ph.D. Thesis, Faculdade de Informática - PPGCC - PUCRS, Porto Alegre, Brazil, 2016, 243p.

[Gri16b] Griebler, D. "Domain-specific language & support tool for high-level stream parallelism", Ph.D. Thesis, Computer Science Department - University of Pisa, Pisa, Italy, 2016, 244p.

[GSG20a] Garcia, A. M.; Schepke, C.; Girardi, A. "PAMPAR: A new parallel benchmark for performance and energy consumption evaluation", *Concurrency and Computation: Practice and Experience*, vol. 32–20, oct 2020, pp. e5504, https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.5504.

[GSG+20b] Garcia, A. M.; Serpa, M.; Griebler, D.; Schepke, C.; Fernandes, L. G. L.; Navaux, P. O. A. "The impact of CPU frequency scaling on power consumption of computing infrastructures". In: International Conference on Computational Science and its Applications, 2020, pp. 142–157.

[GSHW14] Gedik, B.; Schneider, S.; Hirzel, M.; Wu, K.-L. "Elastic scaling for data stream processing", *IEEE Transactions on Parallel and Distributed Systems*, vol. 25–6, dec 2014, pp. 1447–1463.

[Gus88] Gustafson, J. L. "Reevaluating Amdahl's law", *Commun. ACM*, vol. 31–5, May. 1988, pp. 532–533.

[GV13] Grant, W.; Voorhies, R. "Cereal: A C++11 library for serialization". Source: https://uscilab.github.io/cereal, 2013.

[GVS+19] Griebler, D.; Vogel, A.; Sensi, D. D.; Danelutto, M.; Fernandes, L. G. "Simplifying and implementing service level objectives for stream parallelism", *Journal of Supercomputing*, vol. 76, jun 2019, pp. 4603–4628.

[Hal77] Halstead, M. H. "Elements of software science", *Elsevier*, vol. 36–1, may 1977, pp. 4–41.

[Haz20] Hazelcast, I. "Hazelcast in-memory computing platform". Source: https://hazelcast.com/products/in-memory-computing-platform/ #in-memory-solutions, Jan 2021.

[HGDF20] Hoffmann, R. B.; Griebler, D.; Danelutto, M.; Fernandes, L. G. "Stream parallelism annotations for multi-core frameworks". In: Proceedings of the Brazilian Symposium on Programming Languages (SBLP), 2020, pp. 48–55.

[HH21a] Henning, S.; Hasselbring, W. "How to measure scalability of distributed stream processing engines?" In: Proceedings of the International Conference on Performance Engineering, 2021, pp. 85–88.

[HH21b] Henning, S.; Hasselbring, W. "Theodolite: Scalability benchmarking of distributed stream processing engines in microservice architectures", *Big Data Research*, vol. 25, jul 2021, pp. 100209.

[HJHF14] Heinze, T.; Jerzak, Z.; Hackenbroich, G.; Fetzer, C. "Latency-aware elastic scaling for distributed data stream processing systems". In: Proceedings of the International Conference on Distributed Event-Based Systems, 2014, pp. 13–22.

[HK19] Hueske, F.; Kalavri, V. "Stream processing with Apache Flink: fundamentals, implementation, and operation of streaming applications". O'Reilly Media, 2019, 310p.

[HLGF22] Hoffmann, R. B.; Löff, J.; Griebler, D.; Fernandes, L. G. "OpenMP as runtime for providing high-level stream parallelism on multi-cores", *The Journal of Supercomputing*, vol. 1–1, jan 2022, pp. 7655–7676.

[HLLL22] Huang, T.-W.; Lin, D.-L.; Lin, C.-X.; Lin, Y. "Taskflow: A lightweight parallel and heterogeneous task graph computing system", *IEEE Transactions on Parallel and Distributed Systems*, vol. 33–6, jun 2022, pp. 1303–1320.

[HMP+21] Hesse, G.; Matthies, C.; Perscheid, M.; Uflacker, M.; Plattner, H. "ESPBench: The enterprise stream processing benchmark". In: Proceedings of the International Conference on Performance Engineering, 2021, pp. 201–212.

[HOL22] Herodotou, H.; Odysseos, L.; Lu, J. "Automatic performance tuning for distributed data stream processing systems". In: Proceedings of the International Conference on Data Engineering, 2022, pp. 1–4.

[HP85] Harel, D.; Pnueli, A. “On the development of reactive systems”. In: Proceedings of the International Conference on Logics and Models of Concurrent Systems, Apt, K. R. (Editor), 1985, pp. 477–498.

[HSS+14] Hirzel, M.; Soulé, R.; Schneider, S.; Gedik, B.; Grimm, R. “A catalog of stream processing optimizations”, *ACM Comput. Surv.*, vol. 46–4, Mar. 2014.

[Hup09] Huppler, K. “The art of building a good benchmark”. In: Proceedings of the International Conference on Performance Evaluation and Benchmarking, Nambiar, R.; Poess, M. (Editors), 2009, pp. 18–30.

[HZEF16] Hughes, J. N.; Zimmerman, M. D.; Eichelberger, C. N.; Fox, A. D. “A survey of techniques and open-source tools for processing streams of spatio-temporal events”. In: Proceedings of the International Workshop on GeoStreaming, 2016, pp. 1–4.

[ICDV15] Imran, M.; Castillo, C.; Diaz, F.; Vieweg, S. “Processing social media messages in mass emergency: A survey”, *ACM Computing Surveys*, vol. 47–4, Jun. 2015.

[Int20] Intel, C. “Breakthrough memory optimized for data-centric workloads”. Source: https://www.intel.com.br/content/www/br/pt/architecture-and-technology/optane-dc-persistent-memory.html, Jan 2021.

[IPV17] Imai, S.; Patterson, S.; Varela, C. A. “Maximum sustainable throughput prediction for data stream processing over public clouds”. In: Proceedings of the International Symposium on Cluster, Cloud and Grid Computing (CCGRID), 2017, pp. 504–513.

[IPV18] Imai, S.; Patterson, S.; Varela, C. A. “Uncertainty-aware elastic virtual machine scheduling for stream processing systems”. In: Proceedings International Symposium on Cluster, Cloud and Grid Computing (CCGRID), 2018, pp. 62–71.

[IT18] Ivie, P.; Thain, D. “Reproducibility in scientific computing”, *ACM Comput. Surv.*, vol. 51–3, jul 2018.

[Jai17] Jain, A. “Mastering apache storm: Real-time big data streaming using kafka, hbase and redis”. Packt Publishing Ltd, 2017, 552p.

[JBM+16] Janjic, V.; Brown, C.; Mackenzie, K.; Hammond, K.; Danelutto, M.; Aldinucci, M.; Garcia, J. D. “RPL: A domain-specific language for designing and implementing parallel C++ applications”. In: Proceedings of the International Conference on Parallel, Distributed, and Network-Based Processing, 2016, pp. 288–295.

[KBDM13] Kambona, K.; Boix, E. G.; De Meuter, W. “An evaluation of reactive programming and promises for structuring collaborative web applications”. In: Proceedings of the International Workshop on Dynamic Languages and Applications, 2013, pp. 1–9.

[KHAL+14] Kaiser, H.; Heller, T.; Adelstein-Lelbach, B.; Serio, A.; Fey, D. “HPX: A task based programming model in a global address space”. In: Proceedings of the International Conference on Partitioned Global Address Space Programming Models, 2014, pp. 1–11.

[KLLDS12] Khammassi, N.; Le Lann, J.-C.; Diguet, J.-P.; Skrzyniarz, A. “MHPM: Multi-scale hybrid programming model: A flexible parallelization methodology”. In: Proceedings of the International Conference on High Performance Computing and Communication, 2012, pp. 71–80.

[KLV61] Kelly, J. L.; Lochbaum, C.; Vyssotsky, V. A. “A block diagram compiler”, *The Bell System Technical Journal*, vol. 40–3, 1961, pp. 669–678.

[KRK+18] Karimov, J.; Rabl, T.; Katsifodimos, A.; Samarev, R.; Heiskanen, H.; Markl, V. “Benchmarking distributed stream data processing systems”. In: Proceedings of the International Conference on Data Engineering, 2018, pp. 1507–1518.

[KWCF+16] Koliousis, A.; Weidlich, M.; Castro Fernandez, R.; Wolf, A. L.; Costa, P.; Pietzuch, P. “SABER: Window-based hybrid stream processing for heterogeneous architectures”. In: Proceedings of the International Conference on Management of Data, 2016, pp. 555–569.

[LALC+22] Lobato, A. G. P.; Andreoni Lopez, M.; Cardenas, A. A.; Duarte, O. C. M. B.; Pujolle, G. “A fast and accurate threat detection and prevention architecture using stream processing”, *Concurrency and Computation: Practice and Experience*, vol. 34–3, 2022, pp. e6561.

[LGFd+19] López-Gómez, J.; Fernández Muñoz, J.; del Rio Astorga, D.; Dolz, M. F.; Garcia, J. D. “Exploring stream parallel patterns in distributed MPI environments”, *Parallel Computing*, vol. 84, may 2019, pp. 24–36.

[LHP+22] Löff, J.; Hoffmann, R. B.; Pieper, R.; Griebler, D.; Fernandes, L. G. “DSParLib: A C++ template library for distributed stream parallelism”, *International Journal of Parallel Programming*, vol. 50–50, oct 2022, pp. 1–32.

[LLG19] Liu, L.; Li, H.; Gruteser, M. “Edge assisted real-time object detection for mobile augmented reality”. In: Proceedings of the International Conference on Mobile Computing and Networking, 2019, pp. 1–16.

[LLS+15] Lee, I.-T. A.; Leiserson, C. E.; Schardl, T. B.; Zhang, Z.; Sukha, J. “On-the-fly pipeline parallelism”, *ACM Trans. Parallel Comput.*, vol. 2–3, sep 2015.

[LM87] Lee, E. A.; Messerschmitt, D. G. “Synchronous data flow”, *Proceedings of the IEEE*, vol. 75–9, sep 1987, pp. 1235–1245.

[LPDTP+12] Le-Phuoc, D.; Dao-Tran, M.; Pham, M.-D.; Boncz, P.; Eiter, T.; Fink, M. “Linked stream data processing engines: Facts and figures”. In: Proceedings of the International Conference on The Semantic Web, 2012, pp. 300–312.

[Luc71] Lucas, H. “Performance evaluation and monitoring”, *ACM Comput. Surv.*, vol. 3–3, Sep. 1971, pp. 79–91.

[LWXH14] Lu, R.; Wu, G.; Xie, B.; Hu, J. “Stream bench: Towards benchmarking modern distributed stream computing frameworks”. In: Proceedings of the International Conference on Utility and Cloud Computing, 2014, pp. 69–78.

[LZS+22] Li, W.; Zhang, Z.; Shu, Y.; Liu, H.; Liu, T. “Toward optimal operator parallelism for stream processing topology with limited buffers”, *J. Supercomput.*, vol. 78–11, jul 2022, pp. 13276–13297.

[Mar20] Martin, D. “Ampere’s new 128-core altra CPU targets intel, AMD in the cloud”. Source: https://www.crn.com/news/components-peripherals/ampere-s-new-128-core-altra-cpu-targets-intel-amd-in-the-cloud, Jan 2021.

[MBDTE17] Moßburger, A.; Beck, H.; Dao-Tran, M.; Eiter, T. “A benchmarking framework for stream processors”. In: Proceedings of the International Conference on Knowledge Engineering and Knowledge Management, Ciancarini, P.; Poggi, F.; Horridge, M.; Zhao, J.; Groza, T.; Suarez-Figueroa, M. C.; d’Aquin, M.; Presutti, V. (Editors), 2017, pp. 153–157.

[McC95] McCalpin, J. D. “Memory bandwidth and machine balance in current high performance computers”, *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, vol. 1–1, Dec. 1995, pp. 19–25.

[MCT+20] Mei, Y.; Cheng, L.; Talwar, V.; Levin, M. Y.; Jacques-Silva, G.; Simha, N.; Banerjee, A.; Smith, B.; Williamson, T.; Yilmaz, S.; Chen, W.; Chen, G. J. “Turbine: Facebook’s service management platform for stream processing”. In: Proceedings of the International Conference on Data Engineering, 2020, pp. 1591–1602.

[MDAT17] Misale, C.; Drocco, M.; Aldinucci, M.; Tremblay, G. “A comparison of big data frameworks on a layered dataflow model”, *Parallel Processing Letters*, vol. 27–01, apr 2017, pp. 1740003.

[MDT18] Mencagli, G.; Dazzi, P.; Tonci, N. “Spinstreams: A static optimization tool for data stream processing applications”. In: Proceedings of the International Middleware Conference, 2018, pp. 66–79.

[MJP+19] Miao, H.; Jeon, M.; Pekhimenko, G.; McKinley, K. S.; Lin, F. X. “StreamBox-HBM: Stream analytics on high bandwidth hybrid memory”. In: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, 2019, pp. 167–181.

[MnDdRA+18] Muñoz, J. F.; Dolz, M. F.; del Rio Astorga, D.; Cepeda, J. P.; García, J. D. “Supporting MPI-distributed stream parallel patterns in GrPPI”. In: Proceedings of the International European MPI Users’ Group Meeting, 2018, pp. 1–10.

[MPJ+17] Miao, H.; Park, H.; Jeon, M.; Pekhimenko, G.; McKinley, K. S.; Lin, F. X. “Streambox: Modern stream processing on a multicore machine”. In: Proceedings of the International Conference on Usenix Annual Technical Conference, 2017, pp. 617–629.

[MRR12] McCool, M.; Reinders, J.; Robison, A. “Structured parallel programming: patterns for efficient computation”. Elsevier, 2012, 446p.

[MSM04] Mattson, T. G.; Sanders, B.; Massingill, B. “Patterns for parallel programming”. Pearson Education, 2004, 118p.

[MSS04] MacDonald, S.; Szafron, D.; Schaeffer, J. “Rethinking the pipeline as object-oriented states with transformations”. In: Proceedings of the International Workshop on High-Level Parallel Programming Models and Supportive Environments, 2004, pp. 12–21.

[MTC+21] Mencagli, G.; Torquati, M.; Cardaci, A.; Fais, A.; Rinaldi, L.; Danelutto, M. “Windflow: High-speed continuous stream processing with parallel building blocks”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 32–11, apr 2021, pp. 2748–2763.

[MTG+19] Mencagli, G.; Torquati, M.; Griebler, D.; Danelutto, M.; Fernandes, L. G. L. “Raising the parallel abstraction level for streaming analytics applications”, *IEEE Access*, vol. 7, sep 2019, pp. 131944–131961.

[MVGF19] Maron, C. A. F.; Vogel, A.; Griebler, D.; Fernandes, L. G. “Should PARSEC benchmarks be more parametric? a case study with dedup”. In: Proceedings of the International Conference on Parallel, Distributed and Network-Based Processing, 2019, pp. 217–221.

[Nab16] Nabi, Z. “Pro Spark Streaming: the zen of real-time analytics Using Apache Spark”. Apress, 2016, 249p.

[NCP02] Nilsson, H.; Courtney, A.; Peterson, J. “Functional reactive programming, continued”. In: Proceedings of the International Workshop on Haskell, 2002, pp. 51–64.

[NNG18] Nasiri, H.; Nasehi, S.; Goudarzi, M. “A survey of distributed stream processing systems for smart city data analytics”. In: Proceedings of the International Conference on Smart Cities and Internet of Things, 2018, pp. 1–7.

[NQA+20] Nauman, A.; Qadri, Y. A.; Amjad, M.; Zikria, Y. B.; Afzal, M. K.; Kim, S. W. “Multimedia internet of things: A comprehensive survey”, *IEEE Access*, vol. 8, 2020, pp. 8202–8250.

[NXC19] Nikouei, S. Y.; Xu, R.; Chen, Y. “Smart surveillance video stream processing at the edge for real-time human objects tracking”. John Wiley & Sons, Ltd, 2019, chap. 13, pp. 319–346, https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781119525080.ch13.

[Pet04] Petitet, A. “HPL-a portable implementation of the high-performance Linpack benchmark for distributed-memory computers”. Source: www.netlib.org/benchmark, Jan 2021.

[PGMPG21] Palyvos-Giannas, D.; Mencagli, G.; Papatriantafilou, M.; Gulisano, V. “Lachesis: A middleware for customizing os scheduling of stream processing queries”. In: Proceedings of the International Middleware Conference, 2021, pp. 365–378.

[PHUK20] Pagliari, A.; Huet, F.; Urvoy-Keller, G. “NAMB: A quick and flexible stream processing application prototype generator”. In: Proceedings of the International Symposium on Cluster, Cloud and Internet Computing, 2020, pp. 61–70.

[PLH+21] Pieper, R.; Löff, J.; Hoffmann, R. B.; Griebler, D.; Fernandes, L. G. “High-level and efficient structured stream parallelism for rust on multi-cores”, *Journal of Computer Languages*, vol. 65, jan 2021, pp. 101054.

[Poh17] Pohl, C. “A hardware-oblivious optimizer for data stream processing”. In: Proceedings of the International Conference on Very Large Databases, Christen, P.; Kemme, B.; Rahm, E. (Editors), 2017, pp. 1–4.

[PRR19] Prasaad, G.; Ramalingam, G.; Rajan, K. “Scaling ordered stream processing on shared-memory multicores”. In: Proceedings of International Conference on Real-Time Business Intelligence and Analytics, 2019, pp. 1–10.

[PSTF12] Preud'Homme, T.; Sopena, J.; Thomas, G.; Folliot, B. "An improvement of OpenMP pipeline parallelism with the batchqueue algorithm". In: Proceedings of the International Conference on Parallel and Distributed Systems, 2012, pp. 348–355.

[Rei07] Reinders, J. "Intel threading building blocks: outfitting C++ for multi-core processor parallelism". Sebastopol, CA, USA: O'Reilly Media, Inc., 2007, 336p.

[RGDF19] Rockenbach, D. A.; Griebler, D.; Danelutto, M.; Fernandes, L. G. "High-level stream parallelism abstractions with spar targeting GPUs". In: Proceedings of the International Conference on Parallel Computing, 2019, pp. 543–552.

[RLA+22] Rockenbach, D. A.; Löff, J.; Araujo, G.; Griebler, D.; Fernandes, L. G. "High-level stream and data parallelism in C++ for GPUs". In: Proceedings of the XXVI Brazilian Symposium on Programming Languages, 2022, pp. 41–49.

[RM19] Röger, H.; Mayer, R. "A comprehensive survey on parallelization and elasticity in stream processing", *ACM Comput. Surv.*, vol. 52–2, Apr. 2019.

[RNCLP18] Russo, G.; Nardelli, M.; Cardellini, V.; Lo Presti, F. "Multi-level elasticity for wide-area data streaming systems: A reinforcement learning approach", *Algorithms*, vol. 11, Sep 2018.

[RSG+19] Rockenbach, D. A.; Stein, C. M.; Griebler, D.; Mencagli, G.; Torquati, M.; Danelutto, M.; Fernandes, L. G. "Stream processing on multi-cores with GPUs: Parallel programming models' challenges". In: Proceedings of the International Parallel and Distributed Processing Symposium Workshops, 2019, pp. 834–841.

[RTMD20] Rinaldi, L.; Torquati, M.; Mencagli, G.; Danelutto, M. "High-throughput stream processing with actors". In: Proceedings of the International Workshop on Programming Based on Actors, Agents, and Decentralized Control, 2020, pp. 1–10.

[Ryd22] Rydning, J. "Worldwide global datasphere and global storagesphere structured and unstructured data forecast, 2022–2026", Technical Report, International Data Corporation (IDC), Needham, MA, USA, 2022, 978p.

[SCS17] Shukla, A.; Chaturvedi, S.; Simmhan, Y. "Riotbench: An IoT benchmark for distributed stream processing systems", *Concurrency and Computation: Practice and Experience*, vol. 29–21, 2017, pp. e4257.

[SCW+22] Sun, D.; Cui, Y.; Wu, M.; Gao, S.; Buyya, R. “An energy efficient and runtime-aware framework for distributed stream computing systems”, *Future Gener. Comput. Syst.*, vol. 136–C, nov 2022, pp. 252–269.

[Sew17] Seward, Julian. “A program and library for data compression”. Source: http://www.bzip.org/1.0.5/bzip2-manual-1.0.5.html, Oct 2020.

[SHGW15] Schneider, S.; Hirzel, M.; Gedik, B.; Wu, K. “Safe data parallelism for general streaming”, *IEEE Transactions on Computers*, vol. 64–2, nov 2015, pp. 504–517.

[SKSM08] Srivastava, A.; Kundu, A.; Sural, S.; Majumdar, A. “Credit card fraud detection using hidden markov model”, *IEEE Transactions on Dependable and Secure Computing*, vol. 5–1, 2008, pp. 37–48.

[SRG+20] Stein, C. M.; Rockenbach, D. A.; Griebler, D.; Torquati, M.; Mencagli, G.; Danelutto, M.; Fernandes, L. G. “Latency-aware adaptive micro-batching techniques for streamed data compression on graphics processing units”, *Concurrency and Computation: Practice and Experience*, may 2020, pp. e5786.

[Ste97] Stephens, R. “A survey of stream processing”, *Acta Informatica*, vol. 34–7, jul 1997, pp. 491–541.

[Sut66] Sutherland, W. R. “The on-line graphical specification of computer procedures.”, Ph.D. Thesis, Massachusetts Institute of Technology, 1966, 127p.

[TA10] Thies, W.; Amarasinghe, S. “An empirical characterization of stream programs and its implications for language and compiler design”. In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, 2010, pp. 365–376.

[TCN+16] Tudoran, R.; Costan, A.; Nano, O.; Santos, I.; Soncu, H.; Antoniu, G. “Jetstream: Enabling high throughput live event streaming on multi-site clouds”, *Future Generation Computer Systems*, vol. 54, jan 2016, pp. 274–291.

[TDVMB17] Tommasini, R.; Della Valle, E.; Mauri, A.; Brambilla, M. “RSPLab: RDF stream processing benchmarking made easy”. In: Proceedings of the International Semantic Web Conference, 2017, pp. 202–209.

[Tec01] Technologies, A. “Advanced design system 1.5, agilent ptolemy simulation”. Source: http://literature.cdn.keysight.com/litweb/pdf/ads15/ptolemy/pt093.html, Jan 2021.

[THR+18] Tawsif, K.; Hossen, J.; Raja, J. E.; Jesmeen, M. Z. H.; Arif, E. M. H. "A review on complex event processing systems for big data". In: Proceedings of the International Conference on Information Retrieval and Knowledge Management, 2018, pp. 1–6.

[TKPP20] Theodorakis, G.; Koliousis, A.; Pietzuch, P.; Pirk, H. "Lightsaber: Efficient window aggregation on multi-core processors". In: Proceedings of the International Conference on Management of Data, 2020, pp. 2505–2521.

[TKPP22] Theodorakis, G.; Kounelis, F.; Pietzuch, P.; Pirk, H. "Scabbard: Single-node fault-tolerant stream processing", *Proc. VLDB Endow.*, vol. 15–2, feb 2022, pp. 361–374.

[TMM+16] Tzelepis, C.; Ma, Z.; Mezaris, V.; Ionescu, B.; Kompatsiaris, I.; Boato, G.; Sebe, N.; Yan, S. "Event-based media processing and analysis: A survey of the literature", *Image and Vision Computing*, vol. 53, sep 2016, pp. 3–19, event-based Media Processing and Analysis.

[Tor19] Torquati, M. "Harnessing parallelism in multi/many-cores with streams and parallel patterns", Ph.D. Thesis, Computer Science Department - University of Pisa, Pisa, Italy, 2019, 378p.

[TSR20] Tantalaki, N.; Souravlas, S.; Roumeliotis, M. "A review on big data real-time stream processing and its scheduling techniques", *International Journal of Parallel, Emergent and Distributed Systems*, vol. 35–5, mar 2020, pp. 571–601, 10.1080/17445760.2019.1585848.

[TTMD22] Tonci, N.; Torquati, M.; Mencagli, G.; Danelutto, M. "Distributed-memory fastflow building blocks", *Int. J. Parallel Program.*, vol. 51–1, dec 2022, pp. 1–21.

[TZ14] Tian, W. D.; Zhao, Y. D. "Optimized cloud resource management and scheduling: theories and practices". Morgan Kaufmann, 2014, 284p.

[VAR19] Voss, M.; Asenjo, R.; Reinders, J. "Pro TBB: C++ parallel programming with threading building blocks". New York, NY, USA: Apress, 2019, 754p.

[vDVdP20] van Dongen, G.; Van den Poel, D. "Evaluation of stream processing frameworks", *IEEE Transactions on Parallel and Distributed Systems*, vol. 31–8, mar 2020, pp. 1845–1858.

[VGDF19] Vogel, A.; Griebler, D.; Danelutto, M.; Fernandes, L. G. "Minimizing self-adaptation overhead in parallel stream processing for multi-cores". In: Proceedings of the International Parallel Processing Workshops, 2019, pp. 12.

[VGDF22] Vogel, A.; Griebler, D.; Danelutto, M.; Fernandes, L. G. “Self-adaptation on parallel stream processing: A systematic review”, *Concurrency and Computation: Practice and Experience*, vol. 34–6, mar 2022, pp. e6759.

[VGF21] Vogel, A.; Griebler, D.; Fernandes, L. G. “Providing high-level self-adaptive abstractions for stream parallelism on multicores”, *Software: Practice and Experience*, jan 2021.

[VGS+18] Vogel, A.; Griebler, D.; Sensi, D. D.; Danelutto, M.; Fernandes, L. G. “Autonomic and latency-aware degree of parallelism management in SPar”. In: Proceedings of the International Parallel Processing Workshops, 2018, pp. 28–39.

[Víl20] Vílchez Moya, C. “Application parallelization and debugging using pattern-based programming”, Technical Report, Undergraduate Thesis of Double Degree in Computer Engineering and Mathematics, Faculty of Informatics UCM, Department of Computer Architecture and Automation, 2020, 50p.

[vKAH+15] v. Kistowski, J.; Arnold, J. A.; Huppler, K.; Lange, K.-D.; Henning, J. L.; Cao, P. “How to build a benchmark”. In: Proceedings of the International Conference on Performance Engineering, 2015, pp. 333–336.

[vTLv16] Čermák, M.; Tovarňák, D.; Laštovička, M.; Čeleda, P. “A performance benchmark for netflow data analysis on distributed stream processing systems”. In: Proceedings of the International Conference Network Operations and Management Symposium, 2016, pp. 919–924.

[Wan16] Wang, Y. “Stream processing systems benchmark: Streambench”, Master’s Thesis, Aalto University, 2016, 66p.

[WCB01] Welsh, M.; Culler, D.; Brewer, E. “SEDA: An architecture for well-conditioned, scalable internet services”. In: Proceedings of the International Symposium on Operating Systems Principles, 2001, pp. 230–243.

[WFM+19] Wang, L.; Fu, T. Z. J.; Ma, R. T. B.; Winslett, M.; Zhang, Z. “Elasticutor: Rapid elasticity for realtime stateful stream processing”. In: Proceedings of the International Conference on Management of Data, 2019, pp. 573–588.

[YLL+22] Yang, J.; Liu, S.; Li, Z.; Li, X.; Sun, J. “Real-time object detection for streaming perception”. In: Proceedings of the International Conference on Computer Vision and Pattern Recognition (CVPR), 2022, pp. 5375–5385.

[YT13] Yogita; Toshniwal, D. “Clustering techniques for streaming data-a survey”. In: Proceedings of the International Advance Computing Conference, 2013, pp. 951–956.

[YWVS17] Yao, F.; Wu, J.; Venkataramani, G.; Subramaniam, S. “TS-Bat: Leveraging temporal-spatial batching for data center energy optimization”. In: Proceedings of the International Global Communications Conference, 2017, pp. 1–6.

[ZGQB17] Zhao, X.; Garg, S.; Queiroz, C.; Buyya, R. “A taxonomy and survey of stream processing systems”. In: *Software Architecture for Big Data and the Cloud*, Mistrik, I.; Bahsoon, R.; Ali, N.; Heisel, M.; Maxim, B. (Editors), Boston: Morgan Kaufmann, 2017, chap. 11, pp. 183–206.

[ZHD+17] Zhang, S.; He, B.; Dahlmeier, D.; Zhou, A. C.; Heinze, T. “Revisiting the design of data stream processing systems on multi-core processors”. In: Proceedings of the International Conference on Data Engineering, 2017, pp. 659–670.

[ZHZH19] Zhang, S.; He, J.; Zhou, A. C.; He, B. “Briskstream: Scaling data stream processing on shared-memory multicore architectures”. In: Proceedings of the International Conference on Management of Data, 2019, pp. 705–722.

[ZLCH20] Zheng, X.; Li, P.; Chu, Z.; Hu, X. “A survey on multi-label data stream classification”, *IEEE Access*, vol. 8, dec 2020, pp. 1249–1275.

[ZMK+19] Zeuch, S.; Monte, B. D.; Karimov, J.; Lutz, C.; Renz, M.; Traub, J.; Breß, S.; Rabl, T.; Markl, V. “Analyzing efficient stream processing on modern hardware”, *Proc. VLDB Endow.*, vol. 12–5, Jan. 2019, pp. 516–530.

[ZSRS16] Zhang, Q.; Song, Y.; Routray, R. R.; Shi, W. “Adaptive block and batch sizing for batched stream processing system”. In: Proceedings of the International Conference on Autonomic Computing, 2016, pp. 35–44.

[ZWZH20] Zhang, S.; Wu, Y.; Zhang, F.; He, B. “Towards concurrent stateful stream processing on multicore processors”. In: Proceedings of International Conference on Data Engineering, 2020, pp. 1537–1548.

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Pesquisa e Pós-Graduação
Av. Ipiranga, 6681 – Prédio 1 – Térreo
Porto Alegre – RS – Brasil
Fone: (51) 3320-3513
E-mail: propesq@pucrs.br
Site: www.pucrs.br