Università degli Studi di Pisa

Dipartimento di Informatica
Dottorato di Ricerca in Informatica

Ph.D. Thesis

Parallel Patterns for Adaptive Data Stream Processing

Tiziano De Matteis

Supervisor
Marco Danelutto

Supervisor
Marco Vanneschi

Abstract

In recent years our ability to produce information has been growing steadily, driven by an ever increasing computing power, communication rates, hardware and software sensors diffusion. This data is often available in the form of *continuous streams* and the ability to gather and analyze it to extract insights and detect patterns is a valuable opportunity for many businesses and scientific applications. The topic of *Data Stream Processing* (DaSP) is a recent and highly active research area dealing with the processing of this streaming data.

The development of DaSP applications poses several challenges, from efficient algorithms for the computation to programming and runtime systems to support their execution. In this thesis two main problems will be tackled:

- **need for high performance**: high throughput and low latency are critical requirements for DaSP problems. Applications necessitate taking advantage of parallel hardware and distributed systems, such as multi/manycores or cluster of multicores, in an effective way;

- **dynamicity**: due to their long running nature (24hr/7d), DaSP applications are affected by highly variable arrival rates and changes in their workload characteristics. *Adaptivity* is a fundamental feature in this context: applications must be able to autonomously scale the used resources to accommodate dynamic requirements and workload while maintaining the desired *Quality of Service* (QoS) in a cost-effective manner.

In the current approaches to the development of DaSP applications are still missing efficient exploitation of intra-operator parallelism as well as adaptations strategies with well known properties of stability, QoS assurance and cost awareness. These are the gaps that this research work tries to fill, resorting to well know approaches such as *Structured Parallel Programming* and *Control Theoretic* models. The dissertation runs along these two directions.

The first part deals with intra-operator parallelism. A DaSP application can be naturally expressed as a set of operators (i.e. intermediate computations) that cooperate to reach a common goal. If QoS requirements are not met by the current implementation, bottleneck operators must be internally parallelized. We will study recurrent computations in *window based stateful operators* and propose patterns for their parallel implementation. Windowed operators are the most representative class of stateful data stream operators. Here computations are applied on the most recent received data. Windows are dynamic data structures: they evolve over time in terms of content and, possibly, size. Therefore, with respect to traditional patterns, the DaSP domain requires proper specializations and enhanced features concerning data distribution and management policies for different windowing methods. A *structured approach* to the problem will reduce the effort and complexity of parallel programming. In addition, it simplifies the reasoning about the performance properties of a parallel solution (e.g. throughput and latency). The proposed patterns exhibit different properties in terms of applicability and profitability that will be discussed and experimentally evaluated.

The second part of the thesis is devoted to the proposal and study of predictive strategies and reconfiguration mechanisms for *autonomic DaSP operators*. Reconfiguration activities can be implemented in a transparent way to the application programmer thanks to the exploitation of parallel paradigms with well known structures. Furthermore, adaptation strategies may take advantage of the QoS predictability of the used parallel solution. Autonomous operators will be driven by means of a *Model Predictive Control* approach, with the intent of giving QoS assurances in terms of throughput or latency in a resource-aware manner. An experimental section will show the effectiveness of the proposed approach in terms of execution costs reduction as well as the stability degree of a system reconfiguration. The experiments will target shared and distributed memory architectures.

Contents

List of acronyms

ACS Autonomic Computing System 81

CEP Complex Event Processing 29

DaSP Data Stream Processing 9

DSMS Data Stream Management System 28

DVFS Dynamic Voltage and Frequency Scaling 103

HFT High Frequency Trading 117

MPC Model Predictive Control 88

QoS Quality of Service 17

SPE Stream Processing Engine 29

SPP Structured Parallel Programming 16

1 Introduction

Nowadays we are living an Information revolution. The amount of data generated by automatic sources such as sensors, infrastructures and stock markets or by human interactions via social media is constantly growing. The numbers of this data deluge are impressive: every day 2.5 exabytes of data are created, so much that 90% of the data in the world today has been created in the last two years alone[1]. Furthermore, these numbers are expected to constantly grow driven by an ever increasing adoption of sensors, towards tens of billions of internet connect devices by 2020.[2]

This live data is usually dispatched as a continuous flow of information: the possibility to gather and analyze it to extract insights and detect patterns has become a valuable opportunity for many businesses and scientific applications. Such applications arise from different contexts and in many cases there are stringent performance requirements. Systems for high frequency trading, healthcare, network security and disaster managements are typical examples: a massive flow of data must be processed on the fly to detect anomalies and take immediate actions. Having a late response is useless and in some cases also harmful.

Classical *store-then-process* (or *batch*) frameworks are not sufficient for this time sensitive applications. They are designed for a world in which data has a beginning and an end. This has led to the emergence of the *Data Stream Processing* (DaSP) paradigm, a new way to deal and work with streaming data. Its main peculiarities are the following [Babcock et al., 2002; Andrade et al., 2014]:

- differently than traditional applications, data is seen as transient continuous streams rather than being modeled as traditional permanent, "in memory" data structures;

[1] http://www-01.ibm.com/software/data/bigdata/what-is-big-data.html

[2] http://www.ibmbigdatahub.com/blog/foresight-2020-future-filled-50-billion-connected-devices

- systems in charge of analyzing these flows of information have no control on how the data elements arrive to be processed;
- due to the unbounded input and the stringent requirements in terms of performance, processing must be performed "on the fly" or, in any case, by using a limited amount of memory. Resorting to techniques for approximate processing or *window* model may be necessary.

DaSP falls under the umbrella of the so called *Big Data* processing. Big Data is generally characterized by the 3Vs properties [Laney, 2001]: *variety*, *volume* and *velocity*. Variety refers to the nature and structure of the information. Volume refers to the magnitude of data produced while velocity refers to the frequency of data generation as well as the dynamic aspects of the data. While classical batch applications and systems cover the variety and volume issues, DaSP focus on the velocity and variety aspects of the "Big Data Challenge".

1.1 The requirements

DaSP applications pose several challenges that regard different contexts of Computer Science, from efficient algorithms for on-the-fly computations to programming environment and runtime system to support their deployment and execution [Stonebraker et al., 2005; Andrade et al., 2014]. We will discuss some of these challenges in particular the ones that regard the supports for expressing and executing these kind of applications.

Need of high performance Stream processing computations must keep up with data ingest rates, while providing high quality analytical results as quickly as possible. Thus, high throughput and low latency are critical requirements for DaSP applications that necessitate taking advantage of parallel hardware and distributed systems such as multi/manycores or cluster of multicores. The desired performance behavior of the application is usually expressed by means of *Quality of Service* (QoS) requirements, i.e. requirements on quantitative metrics describing the performance level achieved by the application. Typical concerns could be "the mean response time of the application must be less or equal than a given threshold" or " the application must be able to support a given throughput".

Handle dynamicity DaSP applications are affected by highly variable arrival rates and exhibit abrupt changes in their workload characteristics, due to their long-running

nature (24h/7d) and dynamic execution scenario. *Adaptivity* (sometimes referred as *elasticity*) is a fundamental feature in this context: applications must be able to autonomously scale up/down the used resources to accommodate dynamic requirements and workload while maintaining the desired QoS in a cost-effective manner. This must be done in an automatic and transparent way to end-users and possibly also to application programmers.

Providing availability Systems have to ensure that the applications are up and available and the integrity of the data maintained at all times, despite failures. There is a need for tolerance to failures in application components or data sources, as well as failures in the processing hardware and middleware infrastructure. For these reasons, applications must be designed with fault tolerance and resilience requirements in mind.

These are all well known challenges in the HPC community, which deserve special attention in this context due to the continuous nature of the problem. Over the recent years many Data Stream Processing systems have been proposed and a lot of research has been done in both the academia and the industry, highlighting that this topic, besides being a fascinating and challenging research one, can be of strategic importance for many businesses. Still, these problems are not completely tackled in DaSP community. The aim of this thesis is to study and propose new solutions for the *parallelism* and *adaptivity* problems. They will be studied hand in hand, resorting to the twenty years experience of our HPC group here in Pisa. Fault tolerance issues are not addressed in this dissertation but they could constitute a natural extension of this work.

In the following we discuss various motivating scenarios that highlight the principal characteristics of DaSP applications. Then we introduce our vision and approaches to the aforementioned challenges.

1.2 Motivating examples

Examples of real applications that convey the characteristics previously described arise in various contexts. In the following we detail three of them, but a plethora of similar situations can be found also in the automotive context, healthcare or the Internet of Things. As the "sea change", we can only imagine what kind of applications could arise in the next few years. But one thing is sure: Data Stream Processing is here to

stay and here will be no value on all this huge amount of real time data, till applications will not be able to ingest and analyze it in a fast, adaptive and reliable way.

1.2.1 Financial applications

In the last 20 years financial markets have been revolutionized by the *algorithmic* and *high frequency trading*. This results in a transition from physical exchanges to electronic platforms, where automated applications monitor markets and manage the trading process at high frequency. Information about quotations, trades and transactions occurring on different exchanges are disseminated in a continuous basis by real time data distributions services such as the *Options Price Reporting Authority* (*OPRA*) [OPRA] or *Bloomberg B-Pipe* [Bloomberg]. Financial applications require to analyze these market data feeds in order to identify trends and spot opportunities. Typical computations range from simple filtering and aggregation (e.g. bargain index computation [Andrade et al., 2009]) to complex computation or correlations between different streams. Just to mention, recognition of chart patterns within stock price is a common method which is used to predict the future development of stock prices [Investopedia]. If the last received quotes of a given stock (e.g. "the last 1000") exhibit a well defined behavior, automatic trading actions can be taken to anticipate competitors ones.

Financial applications are emblematic situations in which latency is one of the primary design considerations [Brook, 2015]. A trader with significant latency will be making trading decisions based on information that is stale. On the other hand, guaranteeing a few millisecond of processing time could constitute a significant advantage over competitors. On the other hand, in the last years data rates are constantly growing, fueled by the growth of algorithmic and electronic trading. Recent analysis show that currently (beginning of 2016) a single produced stream that conveys information about different stock symbols may reach a peak message rate of 400,000 messages per 100-milliseconds, with an increasing trend [3]. However the average rates are well below that figure, often of one order of magnitude, highlighting a very dynamic environment in which bursts are due to external events (e.g. news, public mood, political decisions) and may regard only part of quoted stocks. High performance solutions are unavoidable to process this ever increasing feeds rate and adaptivity is a desirable future to quickly handle their natural volatility in a cost-effective manner.

[3] `http://www.opradata.com/specs/opra_bandwidth_100ms_jan2016_update.pdf`

1.2.2 Network applications

Managing a large data communications network requires constant network monitoring. Analysis objectives can be disparate. For example network administrators could be interested in traffic monitoring performed on the traffic summaries produced by network apparatus. Common computation regards the analysis of the load of a system (e.g. which share of bandwidth is used by different applications, which part of the network uses a certain amount of bandwidth), of characteristics of the flow (like distribution of life time and size), of characteristics of sessions and many more, [Plagemann et al., 2004].

Beyond infrastructure management, cyber security is another hot topic strictly related to traffic monitoring. Combat security threats could avoid productivity and economic losses as well as data theft. Recently *Network Intrusion Detection Systems* have been introduced to face this problem. They monitor and analyze traffic in real-time in order to identify possible attacks, generating alerts as fast as possible in order to allow taking appropriate countermeasures [Scarfone and Mell, 2007]. Usually their detection algorithms involve complex stream processing and analysis of multiple traffic flows that must keep up with the line speed [Yu et al., 2006; Zhao et al., 2015]. Similar systems have to deal with streaming data from multiple sources with aggregated rates approaching several to tens of gigabyte per second for a reasonably large corporate network, especially during working hours. Clearly, there are requirements on performance and scalability, with desired detection latencies have to be in the order of milliseconds: slowly reacting can lead to considerable damage in both economic and security terms.

1.2.3 Social media analysis

Social media have become ubiquitous and important for social networking and content sharing. Among them, Twitter, a microblogging service, has become a very popular tool for expressing opinions, broadcasting news or simply communicate with friends. It has played a prominent role in socio-political events, such as the Arab Spring or the Occupy Wall Street movement, or in quickly reporting natural disasters.

Twitter's popularity as an information source has led to the development of applications and researches in various domains. In particular, data mining techniques can be applied for expanding researchers' capability of understanding new phenomena [Gundencha and Liu, 2012]. One popular kind of reasoning conducted on Twitter regards the identification of *real-world events* from the tweets stream. Convention-

ally, event detection has been conducted on static document collections, where all the documents in the collection are somehow related to a number of undiscovered events. Recent works on Twitter have started to process data as a stream, as it is produced. Many of them are focused on identifying events of a particular type, such as breaking news [Sankaranarayanan et al., 2009] or earthquakes and typhoons [Sakaki et al., 2010]. Especially in the case of disasters caused by natural hazards, crisis responders may want lower latency in order to better respond to a developing situation. Other works try to address the problem of identifying any type of event (and its related tweets) without *a priori* knowledge of the topic. Here the techniques are various, ranging from on-line clustering methods to statistical techniques [Weiler et al., 2016]. Typically, these analyses focus their attention on the last received data, more important for the end-users.

Another interesting area of research regards the interaction between Twitter discussions and stock markets. Many traders, investors, financial analysts and news agencies post tweets about the stock market in Twitter. As a result, there can be thousands of tweets each day related to certain stocks. In general, the number of tweets concerning a stock varies over days, and sometimes exhibits a significant spike, indicating a sudden increase of interests in the stock. Strategies for stock options trading could use the mood expressed in these tweets to drive their trading action [Bollen et al., 2011].

These interesting applications have to face the intrinsic characteristics of Twitter data feed. Currently, Twitter has more than 320 million active users publish over 500 million of tweets every day, with an average of 5,700 tweets per second. This rate may vary substantially in short periods of time, for example in reaction to a particular social event, increasing the rate of tweets on a particular topic of several orders of magnitude.[4] Since relevant topics or events rapidly change, systems must adapt their behavior to the amount of incoming data in order to provide prompt results.

1.3 High performance DaSP applications

Without loss of generality, we can express Data Stream Processing applications by means of a *computation graph* (sometimes referred as *data flow graph*, see Figure 1.1) [Andrade et al., 2014; Cugola and Margara, 2012c]. This has in input a certain set of continuous data streams and outputs are possibly returned as others continuous flows of data: they represent, respectively, the input and output streams of the application.

[4] https://blog.twitter.com/2013/new-tweets-per-second-record-and-how

Internal nodes express *operators*, i.e. intermediate computations, at least one, in which the application can be decomposed.

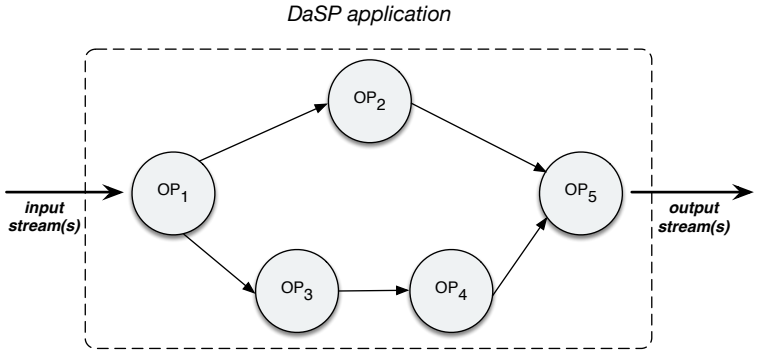


Figure 1.1: Example of computation graph

Generally, an operator consumes input data and applies transformations on them according to its internal processing logic. Operators can be *stateless* or *stateful*. Stateless operators process each input data item independently. On the other hand, stateful operators keep an internal state while processing input stream. Output results depend on the value of the internal state which is continuously updated each time a new item is received. Since continuous data streams may be infinite, there is the problem of maintaining only a portion of the incoming information. In many applications, recent information is usually more relevant than the old one. Therefore, a very common solution is to maintain only the most recent data, in temporary *window* buffer and perform the computation over it.

Various Stream Processing frameworks facilitates the work of developers of mapping their application into a computation graph, by providing mechanisms to instantiate operator, streams and so on. Usually set of reusable operators that can be found in many applications is provided to programmers to increase the productivity [Andrade et al., 2014]. Just to mention some of them, consider many relational operators such as *filters*, *join*, *sort*, but also *aggregates* (sum, count, max), *windowed operators* or *edge adapters* that provide conversion from external sources data format into a format that can be used by an application (and vice versa). In addition, most recent frameworks support customizable operators implemented in general-purpose programming language (e.g. [Apache Storm], [Apache Flink]). The abstract computation is then executed on a set of computational resources. The runtime system provides all the mechanisms to map the computation into a set of communicating threads or processes, deploy and maintain them on the available resources.

1.3.1 Parallelism opportunities in DaSP applications

DaSP applications should be able to efficiently exploit parallel hardware in order to meet requirements of high throughput (i.e. applications must be able to cope with high volume of incoming data) and low latency (i.e. results must be computed in a short period of time). Since graph operators are independent of each other, a first parallelism opportunity is given by *inter-operator* parallelism, when different operators run concurrently on different processing elements. If inter-operator parallelism is not enough the computation graph has to be restructured. We have to understand which sequential operator is a bottleneck and parallelize it. The final result has to be a computation graph, functionally equivalent to the initial one, in which some nodes are transformed through an internal parallelization.

With the exception of rare cases, hand made parallelization is not a viable solution. Besides being an impediment to software productivity and reduced time to development, such low level approaches prevent code and performance portability. As well known in the HPC community, to avoid these problems we have to resort to a *high level approach to parallel programming* [Darlington et al., 1995; Skillicorn and Talia, 1998; Cole, 2004]. The programming environment has to offer the high level parallel constructs directly to programmers that can use them to compose their parallel application. This simplifies programming by raising the level of abstraction: the programmer concentrate on computational aspects, having only an abstract high-level view of the parallel program, while all the most critical implementation choices are left to the programming tool and runtime support. Moreover this will render the program architecture independent, providing reasonable expectation about its performances when executed on different hardware.

As will be described in Chapter 2, most of the existing frameworks provide constructs to express intra-operator parallelism in very simple forms but this regards only stateless operator or simple cases of stateful operators. Although recurrent, these two situations are far from exhaustively cover all the possible solutions, leaving to the programmer the burden of implementing its own parallel schema when they are not sufficient. For example, the processing logic of a windowed operator is not so easily parallelizable. Its internal state creates dependencies between the processing of individual tuples as each tuple might alter the operator's state. A legitimate parallelization must be safe, in the sense that is must preserve the sequential semantic. For this reasons, we can state that a complete methodology for intra-operator parallelism, that cover the majority of the cases, is still lacking.

We advocate that a *Structured Parallel Programming* (SPP) approach [Bacci et al.,

1995; Vanneschi, 2002; Aldinucci et al., 2003; Cole, 2004; Vanneschi, 2014] could be beneficial for the DaSP context. The main idea behind structured parallel programming is to let the programmer define an application by means of *parallel patterns* (also called *paradigms*). Parallel patterns are schemas of parallel computations that recur in the realization of many real-life algorithms and applications for which parametric implementations of communications and computation patterns are clearly identified. With parallel paradigms the programmer just selects the proper pattern and describes the sequential code. The rest of the code is produced by the programming environment and its runtime system. Once it is detected that an operator is a hotspot, a proper parallelization should be found by searching in a limited and feasible set of alternative solutions. A significant property of this approach is the existence of proper *Performance Models* able to predict and quantify the *Quality of Service* (QoS) offered by the parallel computation in several execution and environmental conditions. These characteristics are the basic building blocks to ensure performance portability on the various architectures. Furthermore, besides being a methodology to reduce the effort and complexity of parallel programming, SPP simplifies the reasoning about the properties of a parallel solution in terms of throughput, latency and memory occupancy. Exactly what is needed by intra-operator parallelism.

Structured parallel programming is probably the most interesting class of high-level parallel models. It has a long story in the HPC community and our research group. For the aforementioned reasons, we believe that parallel patterns could constitute a good solution also for intra-operator parallelism in DaSP application.

1.3.2 The road to autonomous adaptivity

The use of parallel paradigms, with well known computation/communication schema and relative cost models, makes it possible to define adaptive applications in which reconfiguration activities are completely transparent to the application programmer and adaptation strategies may benefit from the presence of well defined performance models.

To perform run-time adaptation of the application, it is necessary to apply a *reconfiguration*, i.e. a change in the current operator behavior. Reconfigurations can regard increase/decrease in the number of executors, their mapping onto physical resources, changes in the CPU frequency, and so on. This involve intrusive actions on the computation, such as rearrangements of the state partitions and changes in the communication channels to reflect a new parallelism degree. On the very same line of the previous discussion, currently programmers are directly involved in defin-

ing and implementing the reconfiguration activities, relying, when possible, to some basic mechanisms provided by frameworks and sometimes stopping the application execution to implement the reconfigurations (see Chapter 2). Clearly this led to the explosion of problem complexity and to performance problems. Again, we can leverage the presence of well known parallelism patterns in defining reconfiguration mechanisms that are reusable and with well known effects. This could allow us put them directly in the runtime support of the high level programming model, hiding these aspects from the programmers' viewpoint

In every case the decision to execute a reconfiguration, changing the current application configuration, is taken by the control logic exploiting a proper *adaptation strategy*. Given the operating scenario, we can detail some desiderata that an adaptation strategy should have:

- it must ensure execution properties such as the stability of a system configuration and the optimality of the adaptation process;
- most of the existing approaches are throughput oriented. Assuring a certain average latency is a crucial factor in DaSP applications and therefore this critical metric deserves special attention;
- by exploiting performance models of structured parallel programs we can evaluate the key-parameters that mainly influence the behavior of our applications and adopt a *predictive* approach rather than a *reactive* one;
- it should be cost-effective, i.e. avoid wasting of resources. This does not mean that minimize the number of resources used is always the best solution. An interesting opportunity regards the possibility to act on the used resources to reduce the overall energy consumption and cut down the operating costs. This could involve, for example, the adjustment of the operating CPU frequency through architectural mechanisms. Energy reduction and QoS satisfaction are orthogonal aspects with respect to the adaptivity problem. Their synergic study deserves special attention and represents a quite innovative topic in DaSP processing.

Currently it is essentially missing in the DaSP community a methodological approach to adaptation strategies that cover these topics. The solutions proposed in this dissertation will rely on the a control based approach, the *Model Predictive Control* (MPC) [Camacho and Bordons Alba, 2007]. MPC-based adaptation strategies use a system model to predict the system behavior over a future time horizon while

choosing the reconfigurations to execute. It constitutes a powerful strategy able to achieve good optimality and stability in uncertain environments that has been already successfully applied by our research group in structured parallel applications.

1.4 Contributions of the thesis

In the current approach to the development of DaSP applications are still missing efficient exploitation of intra-operator parallelism, as well as adaptations strategies and reconfiguration mechanisms with well known properties of stability, QoS assurance and cost awareness. In our opinion, a Structured Parallel approach in DaSP has the effect of killing two birds with one stone. From one side we ease programmer life by providing a set of reusable patterns to facilitate the parallel implementation of DaSP operators. From the other, we can exploit the deep knowledge of the patterns to provide transparent reconfiguration mechanisms and adaptation strategies with well known behaviors. With these objectives in mind we can summarize the research contributions of the thesis. The fundamental contributions are the following:

- identification of recurrent computations and proposal of parallelization patterns for intra-operator parallelism. We will focus on stateful windowed operators, but some considerations are valid also for generic stateful operators. The proposed patterns are suitable to be integrated also in existing DaSP or SPP frameworks;

- study and proposal of predictive strategies for autonomic DaSP operators. Our approach will be based on the *Model Predictive Control*, with the intent of giving QoS assurances in term of throughput or latency in a cost effective manner. The combination of latency models and predictive strategies is still unexplored, especially in this application context.

Other minor contributions are the following ones:

- implementation of the proposed parallel patterns over shared memory architecture;

- study and implementation of efficient mechanisms for dynamic reconfigurations for a partitioned DaSP stateful operator, targeting shared and distributed architectures;

- a first partial contribution to energy-aware adaptive strategies for DaSP applications. A basic power cost model is introduced to evaluate different application configurations. This, coupled with the exploitation of the *Dynamic Voltage and Frequency Scaling* (DVFS) mechanism typical of modern multicore CPUs, allow us to enhance the proposed predictive strategies with the possibility of reducing the power consumption while assuring a given QoS imposed by the user.

All these aspects will be accompanied with thorough experimental phases on a shared memory architecture in order to asses the validity of the proposed solutions. Experiments on predictive strategies and reconfiguration mechanisms target also distributed memory machines. In our opinion a similar execution scenario represents an interesting and common use case in data stream processing and deserves special attention especially for what concerns the autonomic management of these applications.

1.5 Outline of the thesis

Apart from the introduction and background chapters, the thesis is essentially organized in two main parts. The first part (that spans over Chapters 3 and 4) tackles the problem of inter-operator parallelism exploitation. After this, the second part (Chapters 5, 6 and 7) is devoted to the proposal and study of predictive strategies and reconfiguration mechanisms for parallel and autonomic DaSP operators. More in details, the outline of the thesis is the following:

- Chapter 2 acts as background in the DaSP context and discusses how parallelism and adaptivity is exploited in the related literature and existing frameworks. Firstly we analyze the principal characteristics of a DaSP application, in terms of structure, operators and internal state. Afterwards, we review various DaSP frameworks, examining how parallelism is exploited in similar systems and in the literature. Finally we will discuss how adaptivity is approached in similar applications;

- Chapter 3 briefly reviews the basic concepts on the Structured Parallel Programming approach. We motivate the use of high level approach to ease the development of parallel applications. A well known set of basic paradigms is briefly surveyed together with an analysis of existing frameworks that exploit this idea. Finally we discuss why classical parallel paradigms are not always

sufficient for an efficient exploitation of intra-operator parallelism in DaSP applications;

- in Chapter 4 the first contributions of the thesis are given. The focus is on parallel patterns for windowed operators, which represent the most notable class of stateful operators in DaSP applications. Starting from the analysis of recurrent computations, four different patterns are presented. They differ in terms of applicability, impacts on performance and drawbacks: these characteristics are experimentally evaluated over a multicore-based shared memory architecture;

- Chapter 5 provides some insights on the concepts of Autonomic Computing and its application to adaptive parallel computations. Reconfiguration mechanisms and adaptation strategies are presented. Then, we review different types of adaptation strategies based on reactive and predictive approach. Finally we introduce the Model Predictive Control, as a method for devising predictive strategies with good properties in terms of accuracy, stability and resource consumption;

- Chapter 6 tackles the problem of enhancing DaSP operators with an autonomic behavior in order to meet performance requirements while reducing the operating costs. The focus is on one of the patterns proposed in Chapter 4 due to its generality and diffusion. Solutions here presented are in any case applicable, without many difficulties, also to the other discussed parallelization schema. MPC-based strategies that tackle, among the other, latency requirements and power efficiency are presented and detailed. Then we focus on reconfiguration mechanisms to implements changes in the parallel structure of an operator without impairing its working characteristics;

- Chapter 7 presents a thorough experimental analysis of the previously presented mechanisms and strategies over a realistic High Frequency Trading application. The experiments are performed on shared and distributed memory architectures in order to asses the effectiveness of the proposed solutions;

- finally Chapter 8 presents the conclusions of the thesis and depicts possible future works.

How to read this thesis

There are various ways of reading the contents of this thesis (depicted in Figure 1.2).

This dissertation is conceptually divided into two main parts: one devoted to intra-operator parallelism exploitation and the other related to the adaptive management of DaSP applications. Clearly, the contents are organized in a natural sequence for which it is possible to read this thesis straight through from start to finish following the chapters ordering (solid line in figure). However, although there is usually some connection to the preceding chapter, the readers should feel free to focus only on one of these two main subjects. Therefore, apart from Chapter 1, 2 and 8 that constitute common background and conclusions, reading Chapters 3 and 4 will give insights on the approach pursued in exploiting parallelism in DaSP operators (dashed line in figure). On the other hand, the reader interested in the adaptive management of similar applications could pass from Chapter 2 directly to the heart of the problem addressed in Chapters 5, 6 and 7 (dashed-dot line).

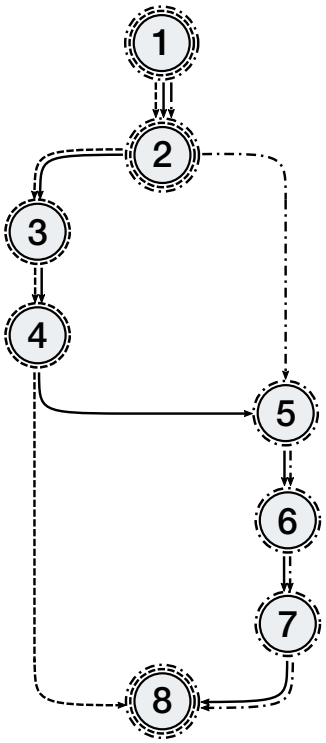


Figure 1.2: Thesis reading paths

1.6 Current publications of the author

The work of this thesis is mainly based on three recent publications of the author:

I. Tiziano De Matteis, Gabriele Mencagli. Parallel Patterns for Window-based Stateful Operators on Data Streams: an Algorithmic Skeleton Approach. In *International Journal of Parallel Programming*, 2016. pp. 1–20.

II. Tiziano De Matteis, Gabriele Mencagli. Keep Calm and React with Foresight: Strategies for Low-Latency and Energy-Efficient Elastic Data Stream Processing. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2016. pp. 13:1–13:12.

III. Tiziano De Matteis, Gabriele Mencagli. Proactive elasticity and energy awareness in data stream processing. In *Journal of Systems and Software*, 2016. In press.

The following represent partially related publications that the author worked during the Ph.D.research:

1. Tiziano De Matteis, Salvatore Di Girolamo, Gabriele Mencagli. Continuous skyline queries on multicore architectures. In *Concurrency and Computation: Practice and Experience*, 2016. pp. 3503-3522.

2. Marco Danelutto, Tiziano De Matteis, Gabriele Mencagli, Massimo Torquati. Parallelizing High-Frequency Trading Applications by using C++11 Attributes. In *Proceedings of the first IEEE International Workshop on Reengineering for Parallelism in Heterogeneous Parallel Platforms*, 2015. pp. 140-147.

3. Tiziano De Matteis, Salvatore Di Girolamo, Gabriele Mencagli. A Multicore Parallelization of Continuous Skyline Queries on Data Streams. In *Proceedings of the 2015 International Conference on Parallel Processing (Euro-Par)*, 2015. pp. 402–413.

4. Tiziano De Matteis. Autonomic Parallel Data Stream Processing. In *High Performance Computing Simulation (HPCS), 2014 International Conference on*, 2014. pp. 995-998.

5. Daniele Buono, Tiziano De Matteis, Gabriele Mencagli. A High-Throughput and Low-Latency Parallelization of Window-based Stream Joins on Multicores. In *12th IEEE International Symposium on Parallel and Distributed Processing with Applications.*, 2014. pp. 117-126.

6. Daniele Buono, Marco Danelutto, Tiziano De Matteis, Gabriele Mencagli, Massimo Torquati. A Lightweight Run-Time Support for Fast Dense Linear Algebra on Multi-Core. In *Proceedings of 12th IASTED International Conference on Parallel and Distributed Computing and Networks*, 2014.

7. Daniele Buono, Tiziano De Matteis, Gabriele Mencagli, Marco Vanneschi. Optimizing Message-Passing on Multicore Architectures Using Hardware Multithreading. In *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on.*, 2014. pp. 262-270.

8. Tiziano De Matteis, Fabio Luporini, Gabriele Mencagli, Marco Vanneschi. Evaluation of Architectural Supports for Fine-Grained Synchronization Mechanisms. In *Proceedings of the 11th IASTED International Conference on Parallel and Distributed Computing and Networks*, 2013.

2 Background on Data Stream Processing

This chapter introduces the basic concepts in Data Stream Processing applications and existing frameworks. The first part focuses on the principal characteristics of a DaSP application: we will introduce the concepts of stream, tuples and the main properties of stream processing operators, including the presence of internal state and the concept of *windowing*. In the second part we will review various DaSP frameworks, focusing on the programming environment, in terms of functionality, operators and features that they offer to programmers. We will discuss how parallelism is exploited in similar systems and in the literature, also analyzing the approaches for an autonomous management of these applications. This will clearly describes the feature and limitations of existing framework as well as of research approaches in the field, a clear starting point for this thesis work.

2.1 Characteristics of a DaSP application

A DaSP application is driven by data flowing from external (remote) sources in a continuous and, theoretically, infinite way. The business logic of the application must process the data on the fly, as soon as it arrives in order to provide timely responses to users.

An application is usually modeled as a direct graph, whose vertices are operators and arcs are streams. The input stream conveys a sequence of individual data items, consumed and analyzed by the application, that represent occurred *events* or, more in general, interesting information. In the following, the term *tuple* will be used as a synonym of input data: a tuple is the fundamental, or atomic, data item embedded in a data stream and processed by the application. It is composed by a set of named and typed *attributes* (or *fields*). Each instance of an attribute is associated with a value. In

these terms, we can view to a data stream as a potentially infinite sequence of tuples sharing a common schema.

In many cases, a physical input stream conveys tuples belonging to multiple *logical substreams* multiplexed together. The correspondence between tuples and substreams is usually made by considering a *key* attribute of the tuple. For example, in trades and quotes sent from financial markets, a key can be the stock symbol to which this information refers; in networking management, a stream can be partitioned according IP addresses. Given this multiplexed streams, various applications need to perform a computation on each substream independently. Back to previous examples, it could be required to perform pattern recognition in an independent way for each stock in the financial market or analyze data by IP source/destination. For the rest of this dissertation we will refer to the aforementioned case as *keyed* stream. In contrast in an *unkeyed* stream we do not have the distinction in logical substreams (i.e. we have only one key).

Generally, an operator consumes the input tuples and applies a *function* (a transformation) on them according to its internal processing logic. As a result, the operator might emit new tuples as new stream, possibly with a different schema. The number of output tuples produced per tuples consumes is called *selectivity* of the operator.

2.1.1 Operator state

Operators can be classified in various ways. According to [Andrade et al., 2014], we will categorize them with respect to state management into three groups:

- a *stateless* operator works on a tuple-by-tuple basis without maintaining data structures created as a result of the computation on earlier data. *Selection*, *filtering* and *projection* are all examples of stateless operators: the processing of each tuple is independent from that of the previous ones;

- in contrast, a *stateful* operator maintains and updates a set of internal data structures while processing input data. The result of the internal processing logic is affected by the current value of the internal state. Examples are *sorting*, *join*, *cartesian product*;

- finally, a *partitioned stateful* (or *keyed*) operator is an important special case of stateful operator. In the case of keyed stream, the operator applies the same computation on each substream in an independent way. Therefore the data structures supporting the internal state are separated into independent partitions according to the relative substream.

Stateful and partitioned stateful operators are the most interesting cases of operators, given the obvious difficulties that can be encountered in devising their parallel implementations.

2.1.2 State type and windowing mechanisms

In a stateful operator, maintain the entire stream history is unfeasible due to the unbounded nature of the input streams. There are two solutions for this problem:

- the state can be implemented by succinct data structures such as *synopses*, *sketches*, *histograms* and *wavelets* [Aggarwal and Yu, 2007] used to maintain aggregate summary metrics. In this case, individual tuples are not stored;
- in various applications, tuples need to be maintained as a whole in internal state. Fortunately, in many applications the significance of each tuple is time-decaying and it is more important to focus the attention on recent data. A solution consists in implementing the state as a *window* buffer in which only the most recent tuples are kept.

Windows are the predominant abstraction used to implement the internal state of DaSP operators and many frameworks provide some form of windowed operator. There are many ways of characterizing the semantics of a window. We will classify windows according to two criteria:

a) in terms of the properties of the tuples that can be held in the buffer (sometimes referred as *eviction policy*);

b) based on how the windows move over new elements (*triggering policy*).

According to the first criteria, various types of policies can be defined. The most used two are *count based* or *time based* windows. In the former case the window has a fixed size specified in terms of number of tuples N: at any time, the window will contain N consecutive stream elements, for example *"the last 1000 received tuples"*. In contrast, a time-based window contains all the elements received in a given time interval T, e.g. *"the elements received in the last 30 seconds"*. This means that the window size is not fixed since the number of maintained tuples can change over time. Time based windows impose that the data stream elements have a timestamp attribute, assigned by the data source or explicitly at their arrival at the applications borders. Less common are *delta-based* window [Andrade et al., 2014], specified using a *delta threshold value* and a tuple attribute referred to as the *delta attribute*. The threshold value specifies

how far the oldest and the newest tuples in window can deviate from each other. For example using a timestamp attribute of the tuple, we can define a delta based window that *"contains all the tuples having a timestamp within one minute one from the other"*.

The second classification criterion regards how the window moves. We can have *tumbling* and *sliding* windows. When a tumbling window is full, its content is ready to be processed. After the window processing is complete, all the tuples in windows are evicted (Figure 2.1). In this way, consecutive activations of the operator's logic will work on completely different tuples.

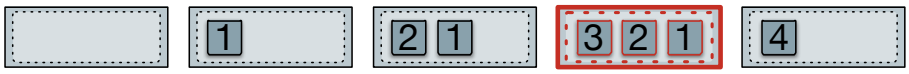


Figure 2.1: The evolution of a count-based tumbling window with size equal to 3. Numbers represent the order of arrival of the different tuples. In bold red it is signaled the situation in which the window content is ready to be processed by operator's logic. After the calculation the window content is discarded.

On the other hand, a sliding window continuously maintains the most recent tuples. When the window is full, only the oldest tuples are evicted to make up room to the new one. A sliding factor δ, expresses when the window's content get processed by operator's algorithm, e.g. after the arrival of δ tuples (Figure 2.2). Depending on

Figure 2.2: The evolution of a count-based sliding window with size equal to 3 and sliding factor 2. In bold red are depicted windows valid for the computation. In this case only the oldest tuples are evicted.

how it is implemented, in time-based sliding windows the operator's logic may be invoked independently from a tuple arrival, just because tuples expire as time passes.

All the four combinations of the previous characteristics are meaningful, although count based and time based sliding window are the most used. We will use them for the rest of this dissertation.

2.2 Data Stream Processing systems

Traditionally, custom coding has been used to solve Data Stream Processing problems. This approach has obvious drawbacks, mainly due to its inflexibility, high cost of development and maintenance, slow response time to new feature requests.

Starting from the early 2000s, several traditional software technologies, such as main memory DBMSs and rule engines, have been repurposed and remarketed to address this application space. The technologies and proposal in these fields, have paved the way to modern and general DaSP systems.

In the following we revisit this history, going from the so called *Data Stream Management Systems* (DSMS) and *Complex Event Processing* (CEP) systems to current *Stream Processing Engines* (SPE). We will focus on the programming environment, which comprises all the high level functionalities offered to programmer to compose its applications, and on the runtime, i.e. the infrastructure that provides the mechanisms and support for running a DaSP application. As we will see, a number of systems has been developed in the last years (see Figure 2.3) in both the academia, open source and industry communities, highlighting that this topic, besides being a fascinating and challenging research one, can be of strategic importance for data and communication intensive applications.

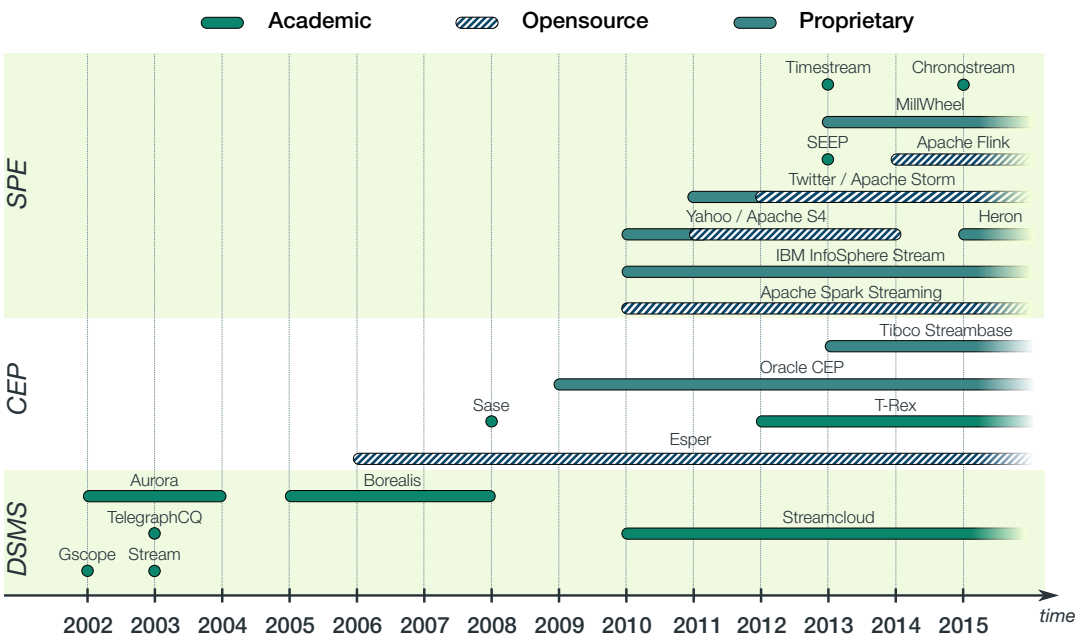


Figure 2.3: Evolution in time of the various DaSP systems, categorized according their type and reference community. A fading bar on the right of the timeline means that the framework is still maintained.

2.2.1 Data Stream Management Systems

Traditional *Database Management Systems* (DBMS) are built around a persistent storage where all the relevant data is kept and whose updates are relatively infrequent. They are neither designed for rapid and continuous loading of individual data nor for fast queries response. To overcome these limitations, the database community developed a new class of systems oriented toward processing large streams of data in a timely way: *Data Stream Management System* (DSMS). DSMSs differ from traditional DBMSs for the fact that they are specialized in dealing with transient data that is continuously updated. In particular they are able to execute *continuous queries* which run continuously and provide updated answers as new data arrives: users do not have to explicitly ask for updated information; rather the system actively notifies it according to installed queries. DSMSs offer to programmers an SQL-like declarative language to define continuous queries.

Several DSMS software infrastructures have been proposed over the first years of 2000s. Examples of these frameworks are *TelegraphCQ* [Chandrasekaran et al., 2003], *STREAM* [Arasu et al., 2003] and *StreamCloud* [Gulisano et al., 2012]. Some of them have been developed for a particular target domain such as *Gigascope* (AT&T Labs) [Cranor et al., 2003], specifically designed for network traffic analysis. Among these proposals, the most important ones are surely *Aurora* and *Borealis*. Aurora [Abadi et al., 2003] was jointly developed by Brandeis University, Brown University and MIT. Its programming model was built on a visual programming language named *SQuAl* (Stream Query Language). SQuAl operators are inherited by relational algebra and can be single-tuple operators and windowed operators. Aurora has a complete runtime system that included a scheduler, a QoS monitor and a load shedder. The project has been extended to investigate distributed processing with Medusa [Stonebraker et al., 2003]. Communications between processors take place using a distributed data transport layer with dynamic bindings between operators and flows. Borealis [Abadi et al., 2005] inherited the functionality of these two projects and added features such as dynamic deployment and adaptation of the application. While their development was blocked years ago, Aurora and Borealis systems proposed multiple ground-breaking ideas that have been included in modern commercial DaSP systems.

2.2.2 Complex Event Processing systems

Many information systems are *event driven* [Etzion and Niblett, 2010]. Incoming data items are seen as notifications of events happening in the external world, which

have to be filtered and combined to understand what is happening. A *Complex Event Processing* (CEP) system may be useful in implementing such applications: it has in input a certain number of streams of *primitive events* that are analyzed for detecting particular patterns that represent *complex events* whose occurrence has to be notified to the interested parties. These systems are essentially pattern matching engines that validate the incoming events with respect to a set of *rules* written in a proper language.

Several CEP systems have been proposed and various are still available and in production, from academia (e.g. *Sase* [Wu et al., 2006], *T-Rex* [Cugola and Margara, 2012a]), industries (e.g. *Oracle CEP* [Oracle, 2016], *Tibco Streambase CEP* [Tibco, 2016]) and open-source community (e.g. *Esper* [EsperTech, 2016]). All of them focus heavily on rule-based detection tasks, expressed in term of complex patterns, and offer windowed operator in order to allow the programmer to specify constraints in terms of time or number of events for the validity of a match. However, they are limited by the language, that do not allow to modify incoming data, and by a poor support for handling unstructured data or complex computations beyond rule-based logic.

2.2.3 Stream Processing Engines

DSMSs or CEP engines provide succinct and simple solutions to many streaming problems, but complex application logics are more naturally expressed using the operational flow of an imperative language rather than a declarative one. For this reason, over the years more generalist frameworks, often referred as *Stream Processing Engine* (SPE), have been proposed to allow programmers more flexibility in defining the applications. Operators can be composed in an arbitrary way and can encapsulate user defined code. At the same time they provide a complete runtime system for the deployment and maintenance of applications, typically in commodity clusters. Now we will review some of them which constitute the state of the art in the field.

Yahoo! S4

S4, proposed by Yahoo! [Neumeyer et al., 2010], was one of the first SPE. Applications are written in terms of small execution units called Processing Elements, using Java. Processing elements are meant to be generic, reusable and configurable in order to be employed in different applications. The runtime automatically handles communications, scheduling and distribution across all the nodes. After the initial release of 2010, S4 became an Apache Incubator project in 2011. The community announced the project retirement in 2014.

Apache Storm

Storm [Apache Storm, 2016] was originally developed at Backtype, a company acquired by Twitter in 2011. Twitter open-sourced Storm in 2012, which is now part of the Apache Software Foundation. In Storm, the computation graph is defined by means of a *topology*, detailed in Java, Scala or Python. Two primitive components can be used to express it: *spout*, i.e. a source of stream, and *bolt*, i.e. the operator that encapsulate the business logic. Multiple spouts and bolts can be packaged in a topology to construct an application. Bolts are essentially empty containers: Storm provides the basic mechanisms to connect and manage them but there are no predefined operators and their implementation is completely left to programmers. It is language agnostic in the sense that processing logic can be implemented in any language (even though Java is the natural choice).

Due to its popularity, various libraries have been designed on top of Storm. *Trident* is a library incorporated into the Storm framework that provides micro-batching and high level constructs like `groupBy`, `aggregate`, `join`, etc. Apache Samoa [Apache Samoa, 2016] contains programming abstractions for distributed streaming machine learning algorithms than can be compiled into Apache Storm topologies. Windowing support was recently added to Apache Storm (in the last release of the framework, `v1.0`). Bolts can use tumbling/sliding time/count based window. In this way, bolt's business logic is triggered as the window advances according to its definition.

Apache Spark and Spark Streaming

Spark was initially started at UC Berkeley AMPLab in 2009 and open sourced in 2010. In 2013, the project was donated to the Apache Software Foundation and in February 2014, Spark became an Apache Top-Level Project [Apache Spark, 2016]. Spark is a general engine for large-scale data processing and provides a streaming library, Spark Streaming [Apache, 2016] that leverage Spark's core in order to allow the definition and execution of Data Stream Processing applications.

While S4 and Storm are based on a record-at-a-time processing model (tuples are processed as they arrive) the approach used in Spark Streaming is different [Zaharia et al., 2012]. The framework performs *micro-batching* and run each streaming computation as a series of deterministic batch computations on small time intervals. The span of time intervals has to be defined by the programmer and can be as low as half a second, therefore this framework is not well suited for applications that necessitate of latencies below this value. Spark Streaming programs can be written in Scala, Java or Python. In contrast to Storm, various stream operators are already defined

and ready to use such as `map`, `reduce` and `join`. There are also basic operators that work on windows and multi-keyed input streams. In particular the `reduceByWindow` and `reduceByKeyAndWindow` are respectively for unkeyed and keyed streams and apply a programmer-defined function to the elements in window. The main limitation is that functions can only be associative. Due to how windows are implemented, in Spark Streaming a programmer can use only time-based sliding window.

IBM Infosphere Streams

IBM Infosphere Streams (IIS) [IBM, 2016; Ballard et al., 2012] is a commercial and proprietary system of IBM. It traces its roots to the *IBM System S* middleware which was developed between 2003 and 2009 at IBM Research. In IIS applications are written in an ad-hoc language (SPL) that is used to describe operators and their stream connections. SPL offers a set of generic built in stream processing operator (e.g. `Aggregate`, `Join`, `Sort`) and the ability to extend this set with user defined operators written in C++ or Java. Windows are associated and defined on the inputs of operators. The language requires the definition of the window characteristics that are relative to the type of the window (count, time and delta based), the eviction policy and the trigger policy allowing the possibility to mimic various window policies.

Apache Flink

Flink [Apache Flink, 2016] has its origin in the Stratosphere project [Alexandrov et al., 2014], a big data framework result of an EU funded research project. In December 2014, Flink was accepted as an Apache top-level project. Flink provides the possibility to define batch applications that runs over its streaming systems. This is in contrast to Spark Streaming, which exploits a batch processing system to run also streaming applications.

Like Spark Streaming and IIS, Flink provides various predefined operators such as `Map`, `Filter`, `Reduce`. It has an extensive support for windowed operators that can be defined over keyed and unkeyed data stream. There are the classical tumbling/sliding and count/time based windows, but it is possible to define other type of windows by defining their eviction and trigger policies (also delta-based)

Interestingly, Flink employs different concepts of time while defining time-based windows. In particular it distinguishes between:

- *processing time* is the time measured by a machine that processes an input data. Processing time is measured simply by the clock of the machines that run the

stream processing application;

- *event time* is the time that each event happened in the real world, usually encoded by a timestamp attached to the data record associated to it. When using event time, out-of-order data can be properly handled. For example, a tuple with a lower timestamp may arrive after an event with a higher timestamp, but transformations will handle these events correctly.

Flink bundles libraries for domain-specific use cases such as machine learning and graph analysis.

Other systems

The ones mentioned above are just notable examples of Stream Processing Engines. There are many other projects in the academic and open source communities, as well as in businesses. Each one has its own particular feature, but they all share a common ground: provide to programmer a way to define and execute data processing applications that work over unbounded data. In academia, new systems are approaching clouds execution of DaSP applications, such as *SEEP* [Castro Fernandez et al., 2013], *Timestream* [Qian et al., 2013] and *Chronostream* [Wu and Tan, 2015]. Authors in [Urbani et al., 2014] propose *AJIRA*, a middleware for both batch and stream processing. It is released as a library and in contrast to other distributed systems, like Storm, Spark Streaming or Flink, it does not require a dedicate cluster and external programs to be used.

Among businesses, practically every big IT company has its own SPE, publicly available or not. Linkedin has *Samza*, that now has become an Apache project [Apache Samza, 2016], Twitter has developed *Heron* to substitute Storm [Kulkarni et al., 2015], Ebay open sourced Pulsar [Murthy et al., 2015]. In this plethora of systems, it seems interesting the recent work of Google that proposes its *Dataflow model* [Akidau et al., 2015]. As the name suggests, it is not really a stream processing system but rather a model that tries to unify batch and streaming computations. It supports different type of windows considering, like Flink, also the event time and not only processing time. The Dataflow processing model has been implemented on top of *MillWheel* for streaming, which is the streaming-native processing system used by google [Akidau et al., 2013].

2.3 Parallelism exploitation in DaSP systems

DaSP applications must take advantage of parallel systems in order to meet their typical performance requirements. In this thesis we will deal mainly with intra-operator parallelism adopting a structured approach. In the following we will detail and review how this type of parallelism is currently exploited in DaSP systems. Most of the existing frameworks express intra-operator parallelism in very simple forms. For stateless ones the most common solution consists in replicating the operator a certain number of times according to a *parallelism degree*, assigning input tuples to the replicas in a load balanced fashion. For partitioned stateful operators the parallel solution consists in using replicas each one working on a subset of the keys, while for unkeyed case there are few proposals. Also in the literature, approaches to this problem are not distant from this spectrum of solutions.

2.3.1 Parallelism in DaSP systems

Most recent systems provide some basic exploitation of parallelism in DaSP application in a (quasi-)transparent way to programmers. This is usually done by executing in parallel some of the high level constructs released to programmers.

In Storm, bolts can be replicated according to the parallelism degree specified by the programmer (*parallelism hint* in their terminology). Different parallelizations can be expressed by specifying how tuples are partitioned among multiple replicas of an operator (the so called *grouping*). *Shuffling* results in a round robin distribution, which is feasible for stateless operator. On the other hand a *field grouping* distribution is feasible for keyed operators: it assures that tuples with the same key are always sent to the same replica. *Custom grouping* can be used by the programmer to define other patterns. If operator's replicas have to interact each other, the cooperation has to be fully specified by the programmer.

In Spark Streaming, each operator is translated into a set of tasks, with proper precedencies and dependencies, that are executed on the available resources as soon as possible. This is in line with the limitation imposed to users: functions applicable to window must be associative. Also in this case the parallelism degree can be set by the programmer.

In IBM Infosphere Streams, programmers can use user-defined parallelism. It is an annotation, posed at the beginning of an operator, which specifies the number of operator's replica (in their terminology *channels*) that have to be created. Developers can replicate any part of an application any number of times to perform parallel

processing, and subdivides (using a *splitter*) the streaming records using either round-robin or hash-based algorithms for keyed streams. The amount of parallelism can be specified at compile time. Developers can also force parallel channels to run on different hosts to improve application performance.

In Flink operations on streams are split into individual tasks which are assigned to task slots and executed on the available resources as soon as possible. Window on keyed stream are evaluated in parallel (at most one task per key). In contrast, windowed operator on unkeyed stream are evaluated at a single task and therefore executed sequentially.

2.3.2 Literature approaches

A considerable part of the research work on the subject, regards the parallelization of particular computations. In the field of continuous queries parallelization various works focus on the parallelization of a particular relational operator. Just to mention some examples, this has been done for parallelizing *join* evaluation [Gedik et al., 2007; Teubner and Mueller, 2011; Buono et al., 2014a] and *skyline* queries [Lu et al., 2013; De Matteis et al., 2015] over windows. In Streamcloud [Gulisano et al., 2012] the approach is a little bit more general: programmer expresses continuous queries that are automatically parallelized for shared-nothing execution environment. Streamcloud support inta-operator parallelism also for stateful operator (e.g. windowed). However given the limited nature of the operator (e.g. aggregates, join, cartesian product) ad hoc solutions are proposed. Operators are replicated and tuples distribution and collection are carefully executed taking into account the semantics of the downstream operator.

In complex event processing, recent works concentrate on parallelizing single rules. In [Hirzel, 2012], the author proposes an extension to IBM SPL language to accommodate parallel execution of rules over keyed streams. As usual, pattern evaluation on events with different keys can be run in parallel. Others approaches try to parallelize rule evaluation for unkeyed streams. In [Cugola and Margara, 2012b] it is proposed an evaluation algorithm parallelized on GPU for windowed match; in [Balkesen et al., 2013a] the parallelization targets commodity multicore.

Concerning more general approaches, parallelization of stateful operators in DaSP systems is addressed in [Wu et al., 2012]. A distributed shared state mechanism is proposed to facilitate parallelization. By default, tuples are routed to the various operator replicas in a round-robin fashion. Therefore, since multiple replicas can be executed simultaneously, the access to the shared state may have to be synchronized

(protected by *locks*). A theoretical model is provided to determine the right level of parallelism as well. Apart from the explicit use of synchronization on shared state that may result in poor performances, this approach does not efficiently handle the case of partitioned stateful parallelism. In [Schneider et al., 2015], authors tackle the same problem but focusing on generic stateless or keyed stateful operator with dynamic selectivity. The solution resembles the ones proposed in Storm and IIS, but in this case it is taken into account also the ordering of produced results. These two approaches can be used in parallelizing generic state and windows, but the execution over the single window is still performed in sequential.

One of the few approaches that treat the problem of parallelizing the computation of aggregate computation on the single window is [Balkesen and Tatbul, 2011]. Author resort to the concept of *panes* introduced in [Li et al., 2005]: each window is divided into non overlapping contiguous partitions called *panes*. Sub-aggregate can be computed in parallel over panes whose results can then be combined into the final aggregate. This clearly requires particular properties on the function to be computed over the window.

2.4 Adaptivity techniques

Given the long running nature of DaSP applications, there is the need to properly address situations of dynamic arrival rates and workloads, in order to respect the Quality of Service contracted by the users or at least minimize its violations.

In the early days, DaSP systems managed this dynamic situation either by statically over-provisioning resources or by means of *load shedding* [Tatbul et al., 2003; Babcock et al., 2004]. The first solution is clearly non cost effective, since a-priori provisioning of resources for handling unpredictably loads can lead to under utilized resources. On the other hand, load shedding implies that in the presence of an increased input rate the system will start discarding part of the data stream to cope with the incoming rate. In many practical cases such as financial applications, health monitoring and network intrusion detection systems, losing data is unacceptable since it can produce a wrong or unwanted effect.

Existing DaSP systems, still fall short in handling this problem. Delegating the scaling decisions to users (like in the case of Storm) or to applications (as in [Qian et al., 2013]) is not a wise decisions, since it will requires a human intervention or a deep knowledge of the parallel computation to the application programmer. For these reasons, the ability to automatically and adaptively change the resources usage

in order to respect the Quality of Service requirements, is a *must* for this kind of systems.

In the current literature various works are trying to tackle this problem. In the following we will try do differentiate them on the basis of *when* the system reacts to a change in the execution environment, *which* properties about the Quality of Service are guaranteed and *how* resources consumption is taken into account. For the first aspect, the majority of those works propose a *reactive strategy*: the system tries to continuously match the amounts of demanded resources to react to a modification in workload or arrival rate. Clearly this could happen only when these events are already occurring, typically resulting in QoS violations. In [Gulisano et al., 2012; Castro Fernandez et al., 2013; Heinze et al., 2014a] authors use threshold based rules on the CPU-Utilization, adding or removing computational resources according to its value. Other works try to collect a more complete set of performance metrics to drive their strategy. In [Lohrmann et al., 2015] measurements on operator's latency, mean and standard deviation of service time and interarrival time are used to reactively enforce QoS constraints. The strategy of [Gedik et al., 2014] has been developed on top of IBM IIS. It takes into account the congestion index (a measure of blocking time at the splitter) and the throughput in scaling up and down the number of operator's replica. Moreover, by remembering the past performances achieved at different operating points they avoid taking continuous reconfigurations. In the DaSP context, this approach is currently the only one that explicitly tries to target the assure properties of the adaptation strategies such as *stability* (i.e. avoid continuous reconfiguration) and *accuracy* (i.e. minimize QoS violations). With respect to reactive policies, a different approach is the one of *proactive strategies*: they try to anticipate a future situation, usually by means of predictions on interesting metrics, and proactively reconfigure the system to avoid resource shortages. The work in [Kumbhare et al., 2014] is one of the few that try to apply a predictive approach in SPEs. It leverages the knowledge of future resource and workload behavior to plan resources allocation. Authors in [Balkesen et al., 2013b] tried to forecast the exact event arrival rate and assumes a fixed per-tuple processing cost (a condition that does not always hold) when determining the optimum parallelization degree.

Concerning the QoS guaranteed, in many of the cited cases ([Gulisano et al., 2012; Castro Fernandez et al., 2013; Heinze et al., 2014b; Gedik et al., 2014; Kumbhare et al., 2014]) strategies are best effort. They are designed to assure the ability of the application to cope with the incoming rate. As indicated in the introduction, we would like to have guarantee also on the experienced latency. More formally, we are interested in guarantee properties on the average *response time*, intended as the time

elapsed from the reception of a tuple that triggers the operator internal processing logic and the production of the corresponding output. The approach of [Mayer et al., 2015] is interesting but it does not give real guarantees. The authors propose methods to dynamically adapt the parallelization degree to limit the length of the input queue to the operator by using Queuing Theory. This gives bounds on the length of the queue but not on the experienced latency. Moreover they assume that input rate and processing time distributions are exponential or deterministic, assumptions that can not hold in general cases. In [Heinze et al., 2014b], authors study how to minimize latency spike during operator movements due to scaling decision, but this don't enforce constraints on the average latency. In [Das et al., 2014] a control algorithm is used for dynamically adapting the batch size in batched stream processing system such as Spark Streaming, in order to minimize, but not guarantee, end to end application latency. Moreover, latency in this kind of system is in the order of hundreds of milliseconds, intolerable for various time sensitive applications. To the best of our knowledge, the strategy proposed in [Lohrmann et al., 2015] is the only one designed to minimize the violations of user defined latency constraints. They resort to performance model based on Queuing Theory for estimating the expected response time of an operator.

Finally, for what concern resource usage, independently from the execution architecture (multi-core, distributed systems, clouds) all the works that regard adaptivity in DaSP systems model the resource consumption by taking into account the number of processing units used. This is clearly an important parameter but, in our opinion, *energy awareness* for this kind of applications should be also taken into account. Energy minimization of parallel applications is an emerging challenge for modern computing systems and in this context it is of vital importance, given that DaSP applications are in execution on 24hr/7d basis. The idea of adapting a parallel program by playing on the number of replicas and CPUs operating frequencies (*Dynamic Voltage and Frequency Scaling*, *DVFS*) to minimize energy consumption while maintaining some performance level is not new but it is still unapplied in the context of DaSP applications. Works such as [Li and Martinez, 2006; Cochran et al., 2011] propose models and methods for determining the optimal combination of parallelism degree and DVFS settings at runtime for parallel workloads in multi-core processors, optimizing energy efficiency given a performance target (or vice versa). In [Shafik et al., 2015] it is proposed an energy minimization approach that considers similar objectives taking into account workload predictions for properly performance annotated OpenMP program. In [Holmbacka et al., 2014] authors focus on parallel dataflow applications with the intent of minimizing the power while providing suf-

ficient performance above a given QoS limit. They find the optimal configuration of clock frequency and number of active cores, but parallelism degree is fixed for the entire program execution: this will render difficult to handle dynamic workloads. In contrast, in [Danelutto et al., 2015] authors use an energy aware approach for the elastic scaling of stateless operators. On the DaSP side, the only work that takes into account energy consumption is [Sun et al., 2015]. It treats the problem of scheduling performance annotated DaSP applications with energy and latency awareness in mind, but not proposes elastic scaling strategies.

2.5 Summary

It should be clear to the reader that current approaches to intra-operator parallelism and adaptivity, although various, are far from being exhaustive. It is still missing a complete coverage of intra-operator parallelism schema, especially considering window based operators which represent a very common type of computation. In our opinion, a programmer has to have the possibility to deal with keyed and unkeyed streams in parallel, possibly exploiting parallelism also in the single window computation. This is not possible with existing approaches that allow to implement certain computations (i.e. stateless or partitioned stateful operators) without efficiently exploit parallelism over single windows. Similarly, from an autonomic point of view, there is the need of adaptation strategies and mechanisms with well known properties of stability, latency assurance and cost awareness in order to deal with very dynamic execution scenarios. These are the gaps that this research thesis tries to fill, resorting to well know approaches such as Structured Parallel Programming and Control Theoretic models.

3 Structured Parallel Programming

In this chapter we will briefly review the basic concepts on Structured Parallel Programming that will be helpful for the rest of the dissertation. The first part introduces the general ideas of the structured programming methodology and motivates the need for such kind of programming. The next section presents a well known set of basic paradigms, discussing their properties and utilization. In the last part a brief review of parallel programming frameworks that exploit the idea of a structured approach is provided. Finally, we will discuss the applicability of classical parallel paradigms to the exploitation of intra-operator parallelism in DaSP applications.

3.1 A structured approach to parallel programming

Parallel programming has become mainstream over the last years, due to the exponential growth in computation, communication and storage capabilities. Historically, parallel programmers resort to hand made parallelization and low level libraries that, giving a complete control over the parallel application, allowed them to manually optimize the code and exploit at best the architecture. Besides being an impediment to software productivity and reduced time to development, such low level approaches prevent *code* and *performance portability*. As usual, code portability represents the ability to compile and execute the same code on different architectures. On the other hand, performance portability represents the ability to efficiently exploit different underlying parallel architecture without rewriting the application. This is an important feature that parallel programs should exhibit, especially in the world of today dominated by multicore CPUs with different organizations, heterogeneous systems and clouds: individuals and industries cannot afford the cost of re-writing and re-tuning an application for every architecture.

As typical in computer science, the answer is to abstract the problem and work at higher level. *High level approaches* to express parallel programs have the advantage of make the programming effort less costly and less time-consuming. They hide the actual complexity of the underlying hardware, providing the programming productivity and performance portability required to ensure the economic sustainability of the programming efforts [Darlington et al., 1995; Skillicorn and Talia, 1998; Cole, 2004]. The programming environment has to offer the high level constructs directly to programmers that can use them to compose their parallel applications. The programmer concentrates on the computational aspects, having only an abstract high-level view of the parallel program, while all the most critical implementation choices are left to the programming tool and runtime support. Exploiting different runtimes for different execution architectures, will render the program architecture independent, providing reasonable expectation about its performances when executed on different hardware.

Low-level mechanisms (such as Posix threads [Butenhof, 1997]), programming libraries (like *MPI* [Gropp et al., 1996]) or extensions (such as *OpenMP* [Dagum and Menon, 1998]), express some characteristics of high level parallel programming and ease the programmer's life. However, she should still know how to orchestrate a parallel schema and in some cases sequential code reorganization is required to introduce proper annotations. For these reasons, the *Structured Parallel Programming* (SPP) approach has been proposed as an effective and attractive approach to high level parallel programming [Bacci et al., 1995; Vanneschi, 2002; Aldinucci et al., 2003; Cole, 2004; Vanneschi, 2014].

The main idea behind structured parallel programming is to let the programmer define an application by means of *parallel paradigms* (also called *patterns*). Parallel paradigms are schema of parallel computations that recur in the realization of many real-life algorithms and applications for which parametric implementations of communications and computation patterns are clearly identified. The *QoS predictability* of these parallel schemes can be used to evaluate their profitability, the best application configuration and to provide adaptivity support.

In our research group, the SPP approach has been successfully applied in various parallel environments, from clusters [Danelutto, 2001] to shared memory architecture [Aldinucci et al., 2014a], from grid [Aldinucci et al., 2006a] to cloud and pervasive environments [Bertolli et al., 2010]. The development of data mining [Coppola and Vanneschi, 2002], signal processing [Buono et al., 2014b] and computational biology [Aldinucci et al., 2014b] applications has benefit of parallel paradigms.

3.2 Parallel paradigms

Without loss of generality, we express a parallel (distributed) application by means of a *computation graph*. Nodes represent modules (or operators) i.e. intermediate computations in which the application can be decomposed. They, communicate by means of internal data streams of data, on which the computations are applied. If the performance requirements are not met, modules that act as *bottlenecks* have to be internal parallelized according to some *parallelism paradigm*. They exhibit the following features [Cole, 1988; Pelagatti, 1993; Bacci et al., 1995; Vanneschi, 2014]:

- they restrict the parallel computation structure to certain predefined patterns;
- they have a precise semantics;
- they are characterized by a specific cost model;
- they can be composed with each other to form complex computations.

This approach frees the programmer from detailed concerns of the mapping between parallel computation schemes and their implementation on the target parallel architecture.

Cost models allow evaluating the performance metrics as functions of other parameters that are typical of the application (e.g. calculation time, data size) and of the architecture (e.g. processor performance, memory access time,...). Important performance measures to consider are:

Definition 3.2.1 (Throughput)**.** *The **throughput** is the average number of stream elements which can be completed in a time unit. It is the inverse of the service time, that is the average time interval between the beginnings of the executions on two consecutive stream elements;*

Definition 3.2.2 (Computation Latency)**.** *The **computation latency** is the average time needed to execute the computation on the single input element*

Definition 3.2.3 (Response time)**.** *The **response time** is the time elapsed from the reception of an input element and the production of the corresponding output. It is given by the computation latency plus the time that the element waits to be processed.*

Parallel paradigms can have different impact on these metrics, captured by their respective cost models. The *QoS predictability* of these parallel schemes has been studied by exploiting formal analysis, rendering the performance modeling of this class

of computations usable also by automatic tools as compilers (e.g. to statically decide the best application configuration on a given architecture) and from run-time supports (e.g. for proving efficient fault tolerance [Bertolli, 2008] or dynamic reconfiguration mechanisms [Vanneschi and Veraldi, 2007; Aldinucci et al., 2008]). QoS predictability of a parallel computation is a fundamental feature also for devising optimal adaptation strategies.

In the following we will review a set of well know parallel patterns, highlighting their major characteristics and performance impacts.

3.2.1 A basic set of parallel patterns

Historically, two broad categories of parallel patterns have been recognized:

- *stream parallel* (or *task parallel*) paradigms: as the name suggest, these patterns work on streams of homogeneous elements. Parallelism is obtained by simply processing multiple elements concurrently. They do not speed up the computation of a single element but the computation of the stream: this means that they do not improve the computation latency. On the contrary, they improve the throughput;

- *data parallel* paradigms: in this case the single computation (possibly also relative to a stream element) is parallelized. Usually this requires to partitioning the data structure and, by reflection, partitioning the computation. Paradigms that fall in this category are able to improve the computation latency.

A parallel paradigm describes the structure of the interactions of a parametric set of entities. We can recognize different types of involved entities, from executors to interface from/to input/output streams. In the following we will refer to units that are in charge of performing the computation on the received data as *workers* (or simply executors). Data distribution towards workers can be performed by an *emitter*, which receives data from input stream(s). Finally a *collector* receives the computed results from workers and transmits the final results onto the output streams of the parallel module.

Pipeline

The *pipeline* paradigm is a very simple solution to some parallelization problems. It assumes that the computation is expressed as a sequential composition of functions

over the input elements, i.e.:

$$\mathcal{F}(x) = \mathcal{F}_n(\mathcal{F}_{n-1}(\dots(\mathcal{F}_1(x))\dots))$$

In this case a possible parallelization is a linear graph of n executors, also called pipeline stages, each one corresponding to a specific function (see Figure 3.1). A similar solution can be adopted to increase the throughput. The service time is given by the more computational intensive stage. The latency may be increased due to the communications overhead between stages.

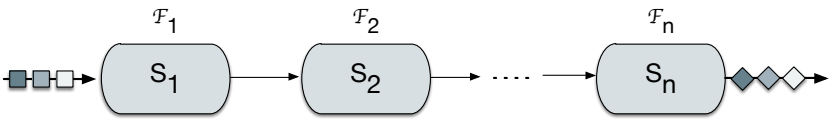


Figure 3.1: Pipeline parallel paradigm

Task-Farm

The *task-farm* (or simply *farm*) paradigm corresponds to the replication of a pure function (i.e. stateless) among a set of identical workers. Assuming that we want to apply the same function $\mathcal{F}$ to each input element. The emitter provides to send every input element to a worker, according to a certain *scheduling policy* able to balance the workers load (see Figure 3.2). If the calculation time of the function has low variance, a round robin solution could be sufficient. The worker receives the input elements and applies on each of them the function $\mathcal{F}$. The results are sent toward the collector that is in charge of collect and transmit them onto the output stream. Being a stream parallel pattern, this solution is able to increase the throughput of the computation, but the computation latency of the single element is the same (the computation is still performed in sequential by the worker).

Data parallel

The data-parallel paradigm consists in the partitioning (or replication) of the input data and replication of the function. In this way distinct but functionally equivalent workers are able to apply the same operation to a distinct data partition in parallel. The emitter provides the distributions of the various partitions (scatter). Collection of the worker results is achieved by the collector exploiting a gather operation, to receive partial results and build the final one.

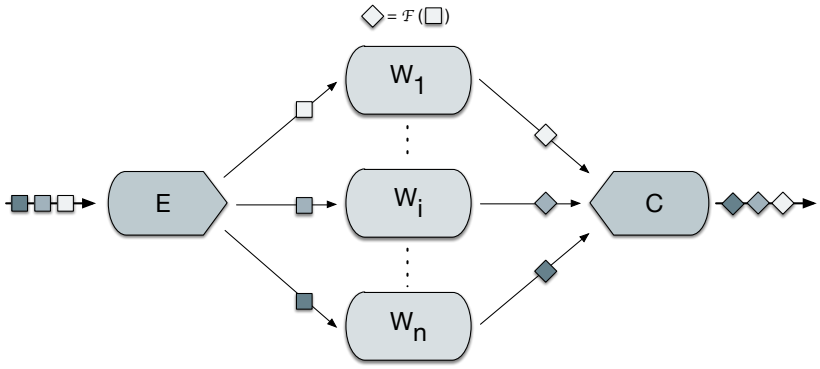


Figure 3.2: Task-Farm parallel paradigm

The simplest data parallel schema is the so called *map* (sketched in Figure 3.3), in which workers are fully independent. Each of them operates on its own local data only, without any communication during the execution.

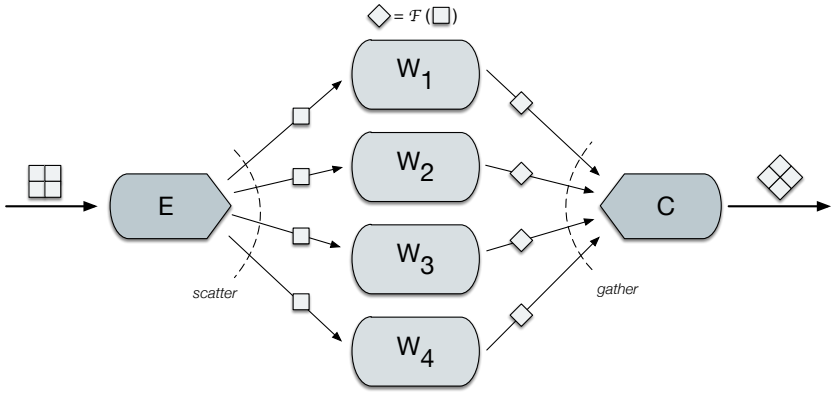


Figure 3.3: Map paradigm, exemplified with 4 Workers

More complex, yet powerful, computations are characterized by cooperating workers: in order to apply a function, a worker may require to access data contained in other worker partitions, because *data dependencies* are imposed by the computation semantics. In this case we speak about *stencil-based* computations, where a stencil is a data dependence pattern implemented by information exchanging between different workers. Data parallel paradigms are able to reduce the computation latency for a single input element. When applied to a stream, they can also improve the throughput of the computation given the reduction in the mean service time.

Reduce

The reduce pattern is a data parallel paradigm, applicable every time we have a computation of the form:

$$y = x_1 \oplus x_2 \oplus \ldots \oplus x_n$$

where the result is a single value obtained by applying an associative function $\oplus$ to all the elements of an input data structure. The input data can be partitioned in n workers, each one performing a "local" reduce on their partition, followed by a "global" reduce.

3.2.2 Patterns composition

Parallel paradigms can be composed to form complex structures. Stream-parallel and data parallel approaches can be composable in stream-based computations, for example a pipeline in which one, or more, stages are implemented as a map. A lot of algorithm can be defined a composition of map and reduce: at first a function is applied to all the elements in a data structure and then the results are merged using a reduction function. Recently, this idea has been applied also in the Google MapReduce [Dean and Ghemawat, 2008] programming environment. It and its popular open source implementation (Hadoop [Apache Hadoop, 2016]) saw an enormous success in the last years in the field of batch processing, demonstrating how a high level approach (combined with proper and efficient run time) could be beneficial.

3.3 Structured Parallel Programming frameworks

A high level parallel framework should provide to programmers a set of reusable patterns that allow an easy parallelization of most algorithms. In this sense, the most used approach to SPP is based on the concept of *algorithmic skeletons*. Algorithmic skeletons were firstly introduced by Cole with his Ph.D. Thesis [Cole, 1988]. He defined a skeletal programming framework as follow:

> *The new system presents the user with a selection of independent "algorithmic skeleton", each of which describes the structure of a particular style of algorithm, in the way in which higher order functions represent general computational frameworks in the context of functional programming languages. The user must describe a solution to a problem as an instance of the appropriate skeleton.*

Each skeleton corresponds to a single parallel pattern. The programmer has to specify only its business logic: orchestration and synchronization of the parallel activities are implicitly defined by the skeleton itself and implemented by the runtime of the framework. Cole proposed four skeletons (*Fixed Degree Divide & Conquer*, *Iterative Combination*, *Cluster* and *Task Queue*), obtained both by the isolation of particular algorithm and by an analysis of patterns that could perform well on the initial target architecture (a transputer). This first proposal, has its own limitations: there was no mention to the fact that skeletons could be composed or nested, therefore limiting the expressiveness of the approach. Moreover in a skeletal framework the programmer has no other way (but instantiating skeletons) to structure her parallel computation. However this work attracts the attention of the research community, and a lot of groups focused on finding the general yet effective patterns that could be promoted to skeleton. This led to the proposal and implementations of various frameworks [González-Vélez and Leyton, 2010] that here we briefly review. Among the others, our research group history in structured parallel programming is quite long, starting with the P^3L skeleton language in 1992, ASSIST in the early 00s' and arriving Fastflow, highlighting a strong background in the field.

P^3L (Universitá di Pisa) was a coordination language that provided *pipeline*, *task farm*, *map*, *reduce* and data-parallel with stencil patterns [Bacci et al., 1995]. A compiler is provided for the language. It uses implementation templates: each one implements a skeleton on a specific architecture and provides a parametric process graph with a performance model. The performance model can then be used to decide program transformations which can lead to performance optimizations. SKELib [Danelutto and M., 2000] inherited the stream based skeleton from P^3L but differs from it because a coordination language is no longer used, but instead skeletons are provided as a library in C.

Lithium [Aldinucci et al., 2003] and its successor Muskel [Danelutto and Dazzi, 2006] were developed at Università di Pisa. Both provide nestable skeletons, such as *pipe*, *map*, and *farm* to the programmer as Java libraries.

Among the most recent skeleton framework we should cite Skandium [Leyton and Piquer, 2010]. It implements *seq*, *pipe*, *farm*, *map*, *farm*, *divide & conquer* and *fork* skeletons, not introducing new patterns with respect to previous works. It is implemented in Java and target multicores. It takes advantage of shared-memory to simplify parallel programming. The Muesli skeleton library [Ciechanowicz et al., 2009] was developed by the University of Münster. Skeletons are provided as a C++ template library. It offers stream parallel skeletons (e.g. *Pipeline*, *Farm*,...) and data

parallel skeletons available as functions of a distributed data structure (e.g. array or matrix). Muesli supports multicore programming with OpenMP; cluster support is offered via MPI.

Finally, two currently maintained projects are Skepu [Enmyren and Kessler, 2010] and Fastflow [Aldinucci et al., 2014a], both of them released as C++ libraries. Skepu is developed at Linköping University. It provides six data-parallel (*map*, *reduce*, *scan*, *mapreduce*, *maparray*, *mapoverlap*) and one task-parallel (*farm*). Each offered skeletons has multiple implementation targeting multicore and multi-GPU systems both with CUDA and OpenCL.

Fastflow is developed at Universitá di Pisa and target multicore as well as GPU. It has a three tier structure:

1. a set of *basic mechanisms*, that comprises efficient *lock-free* and *wait-free* communication channels, processes and threads containers;

2. a set of *core patterns*: at this level there are two patterns (*farm* and *pipeline*) implemented using the basic mechanisms;

3. *high level patterns*: built on top of the core patterns, they are clearly characterized in a specific usage context and are targeted to the parallelization of sequential (legacy) code. Examples are *parallel for*, *map*, *stencil-reduce*, *macro data flow*.

The programmer can exploit Fastflow features by using any of the mentioned levels. Skeletons can be nested. The framework is in active development at the moment of writing this thesis.

A different approach compared to the one of skeletal frameworks, is the one pursued by ASSIST (A Software development System based upon Integrated Skeleton Technology) [Vanneschi, 2002] developed at Universitá di Pisa. In ASSIST there has been the attempt to overcome some of the limitations imposed by the skeleton model. The major concerns are related to the fact that, even if they are very powerful, parallel patterns cannot efficiently capture *every* parallel application. For example, stencils with dynamic dependencies are not always exploitable in existing skeleton frameworks. In the same way we need a larger degree of flexibility in expressing parallel and distributed program structures: we cannot force the programmer to write applications respecting the few well studied patterns. Also Cole [2004] recognizes this lack of expressiveness stating that *"It is unrealistic to assume that skeletons can provide all the parallelism we need. We must construct our systems to allow the integration of skeletal and ad-hoc parallelism in a well defined way"*.

ASSIST provided a structured coordination language to express parallel programs as an arbitrary graph of software modules connected by streams. These allow inter-module parallelism exploitation, by pipelining operators, and complex behavior and loops among the modules. However, parallelism is also available inside the nodes, because each module represents a parallel pattern. Lastly, a module is not forced to be implemented as a parallel pattern: the programmer may provide its specific, hand-made implementation of a parallel module (*parmod*). This effectively solves the cases in which a parallel paradigm cannot be applied. ASSIST is not currently maintained.

3.3.1 Is this sufficient for Data Stream Processing?

At this point, a natural question arises: "are these parallel patterns sufficient to deal also with intra-operator parallelism in DaSP applications?" The answer is "not completely". Clearly for stateless operators the aforementioned patterns (e.g. stream parallel ones) are still valid and easily applicable. In contrast, when it is necessary to parallelize a stateful operator, things become a little bit more complicated. In DaSP stateful operators the results of the computation are obtained by processing a set of the input elements that belong to one or more input streams. In particular windows, define a continuous segmenting of the input data streams: at any instant, they define the set of input elements that must be considered in order to produce the results. Classical stream paradigms assume that the operations are independently applied to distinct elements of a stream. On the other hand, also data parallel paradigm fall short: they work on the single element assuming that it has a finite size.

Recently some research efforts have been directed towards adapting Hadoop (and therefore the Map Reduce pattern) as real time stream processing engine. In [Condie et al., 2010], *MapReduce Online (HOP)* is introduced to support continuous query. Results from mapper nodes are sent directly to reducers as soon as they are produced. However reduce functions (*e.g.* aggregate computation) can be applied at predetermined milestone resulting in very low throughput even for small windows [Brito et al., 2011]. Authors in [Brito et al., 2011] propose *Stream MapReduce*, whose implementation maintains a backward compatibility with MapReduce API. Despite this fact, the MapReduce abstraction is here completely upset. Stateless operators are implemented in mapper node, while stateful ones are implemented as reducers, allowing the use of tumbling or sliding windows. M^3 [Aly et al., 2012] tries to avoid the overhead due to the HDFS (the distributed file system used for data distribution) which introduces significant delays that make it inapplicable for streaming application, introducing main-memory-only data-path between mappers and reducers. They never

terminate, allowing to have only one MapReduce job per query operator that is continuously running. In this case, no performance evaluation is given. In general all these solutions seem a try to fit a square peg in a round hole, breaking the paradigm's abstraction and leading to unacceptable performances.

In our opinion, the DaSP domain requires proper specializations and enhanced features in terms of data distribution and management policies and windowing methods that can not always be found in traditional patterns. To exploit intra-operator parallelism, windows must be distributed or partitioned among workers. Such distributions can be critical, since windows are data structures with particular features. They are dynamic in the sense that their content (in terms of buffered tuples) changes over time according to their triggering and eviction policy. Furthermore, in the case of the time-based semantics, the cardinality of the window in terms of elements changes over time, based on the current stream arrival rate. This means that, after the window of a stream has been distributed among a set of workers, expired elements must be deleted and the new arrived ones inserted. For these reasons two problems arise:

- how to keep up-to-date the window for each input stream. Particularly important are the distribution policy of new arrived elements and the expiration policy. We have to understand which entity of the parallel computation is in charge of checking the expiration of past received elements and which is in charge of their removal from the window;

- how to keep the window partitions balanced, in order to maintain a similar workload among workers.

In our opinion, also accounting these problematic, parallelization issues in DaSP computation can be still dealt with SPP in order to reduce the effort and complexity of parallel programming and simplify the reasoning about the properties of a the parallel solutions in terms of throughput, latency and memory occupancy. This will require to revisit existing patterns and to propose new ones.

3.4 Summary

In this chapter we have provided an overview of structured parallel programming. Various classical patterns have been reviewed highlighting that they cannot completely cover the computation class typical of DaSP.

Proper parallel patterns should be devised to allow the exploitation of intra operator parallelism to cover a reasonable large set of recurrent computations. In our opinion SPP is still a cornestone also for this kind of problems. It eases the programming effort giving, as it will be clearer in the next chapters, the possibility to implement adaptation strategies with performance guarantees and reconfiguration mechanisms transparent to the programmer.

4 Parallel patterns for windowed operators

The goal of this chapter is to study recurrent computations in *window based stateful DaSP operators* and propose patterns for their parallel implementation. We describe the features of parallel patterns in relation to their internal organization as well as their applicability and profitability. Patterns will be presented by abstracting the target architecture, i.e. they can be instantiated both on shared-memory machines and on distributed-memory architectures provided that proper runtime supports are used. We discuss how the proposed solutions can be implemented or emulated in existing DaSP and skeletal frameworks. Finally the results of experimental evaluations performed on a multicore architecture are reported.

The set of proposed patterns will help the programmer providing a set of reusable solutions and simplifies the reasoning about the properties of a parallel implementation in terms of throughput, latency and memory occupancy. In addition, a structured approach to the parallelism exploitation in DaSP computations will be the basis also for what concern the autonomous management of such applications, as we will see in Chapter 6.

The contents of this chapter have been mainly published in [I].

4.1 Preliminaries

When a DaSP application is not able to meet the performance requirements imposed by users (e.g. sustain a given input rate or maintain a certain response time), it has to be restructured. Operators that act as bottlenecks must be internally parallelized. *Stateful* operators maintain and update a set of internal data structures while processing input data. This creates dependencies between the processing of individual

tuples. Parallelizing a stateful operator requires particular attention in preserving its sequential semantics.

In our opinion, parallelization issues can be resolved in an elegant way by means of structured parallel patterns that provide a reusable and easy to use solution to recurrent problems. We will mainly focus on windowed operator. Windows are the predominant state abstraction used to implement the internal state of an operator. Window semantic is specified by eviction and triggering policies (see 2.1.2). In the following we will refer to a general case, highlighting meaningful differences when necessary. Therefore a window is defined by:

- a window size $|\mathcal{W}|$, expressed in time units (seconds, minutes, ...) for *time based* windows or in number of tuples for *count based* windows;
- a sliding factor δ that expresses how the window moves. Analogously to the window size, the sliding factor can be expressed in time units or in number of tuples. If $\delta = |\mathcal{W}|$ we have a *tumbling* window, if $\delta < |\mathcal{W}|$ we have a *sliding* window.

It is important to observe that consecutive windows may have overlapping regions, i.e. the same tuple may belong to multiple consecutive windows. This situation is true for *sliding windows*, that, at a given time instant contains, also tuples not belonging to the preceding windows.

For the sake of generality, we will assume that the computation over the window content is performed at triggering time, e.g. when a window slide. This assumption holds for any type of computation. However there could be cases in which the computation can be performed incrementally, as each tuple arrives into window. Typically this requires that the computation must have certain characteristics, such as distributive or algebraic aggregates [Tangwongsan et al., 2015]. Complex statistics and processing like interpolation, regression or sorting may need the entire window.

In the following we assume a generic window-based stateful operator working on a single input physical stream and producing one output stream. The treatment can be easily generalized. With more than one input stream the usual semantics is the *non-deterministic* one, i.e. the operator receives input items from *any* streams. With more than one output stream the results can be transmitted to one/a subset/all of them according to a given predicate on the results' attributes. On the other hand, we will explicitly deal with *keyed* operators, in which the physical input stream conveys tuples belonging to multiple *logical substreams* multiplexed together. The correspondence between tuples and substreams is usually made by considering a *key* attribute of the

tuple, e.g. the stock symbol in a stream of quotes coming from a financial market. We refer to the set of all possible keys as $\mathcal{K}$.

Finally the last assumption regards tuple ordering. We assume that tuples arrive in order, e.g. by timestamp or sequence number, and that they are processed in the same order of arrival by the sequential operator. The parallel operator may be subject to ordering requirements: downstream operators may need to receive the resulting tuples in an order logically corresponding to the original one. That is the parallel operator must produce results in the same order of the sequential implementation: if, due to tuples arrival, the computation of window i is triggered before window j, the result of window i must be sent outward before the one of window j. To implement this, the parallel operator may rely on additional tuple attributes, such as internal sequence numbers, that are added by the operator itself. In presence of logical substreams we can have a weakest ordering, a *partial* one, if it is required that results are produced in order on substream basis only.

4.2 Parallel patterns categorization

In the context of window based operators, task parallelism assumes a special characterization. The internal state consists in a subsequence of the input tuples received so far. In contrast to classical parallel paradigms, a task is no more a single input element: it is a segment of the input stream corresponding to *all* the tuples belonging to the same window.

Given this new definition of task, the basic performance measurements that we introduced in Chapter 3 should be revised:

Definition 4.2.1 (Throughput and service time)**.** *The **throughput** is now the average number of windows that the operator is able to process in a time unit. It is the inverse of the **service time**, that is the average time interval between the beginnings of the executions of two consecutive windows;*

Definition 4.2.2 (Latency)**.** *The **latency** is the average time needed to execute the computation on the single task, i.e. a window.*

Definition 4.2.3 (Response time)**.** *The **response time** is the time elapsed from the reception of the last tuple triggering a window computation and the production of the correspondent output.*

In order to define quantitatively "how good" a parallel solution is, we introduce the concept of *scalability*:

Definition 4.2.4 (Scalability)**.** *The **scalability** provides a measure of the relative speed of the n-parallel computation with respect to the same computation with parallelism equal to 1. It is defined as the ratio between the throughput achieved with n workers and the one obtained with only one.*

Patterns for windowed operators can be categorized in:

- *window parallel* paradigms: these patterns are capable of executing in parallel at the same time instant computations over multiple windows. Therefore they will not improve the computation latency over the single windows but the throughput of the whole operator;
- in *data parallel* paradigms the single window computation is parallelized. This will require to partition the window among a set of identical executors. Like traditional data parallel, patterns in this category are able to improve the computation latency.

As usual, all the relevant characteristics of a pattern (e.g. impact on throughput, latency, memory utilization) will be derived from its definition and structure

Due to the dynamic nature of the windows, in term of tuples contained and their cardinality, the distribution phase is particular important. Two orthogonal aspects will be considered:

- the *granularity* at which input elements are distributed by the emitter functionality to a set of workers, e.g. the unit of distribution can be a single tuple, a group of tuples or entire windows;
- the *assignment policy*, i.e. how consecutive windows are assigned to parallel workers of the pattern.

Distribution strategies lead to several possible optimizations of the same pattern or, in some cases, to the identification of new patterns. The distribution may also influence memory occupancy and the way in which the window management is performed. About this point, we identify patterns with *agnostic* or *active* workers. In the case of *agnostic* workers all the processing/storing actions needed to build and to update the windows are performed by the distribution logic. Workers are agnostic of the window management and data distribution, i.e. they are just in charge of applying the computation to the received data. In contrast, with *active workers* the window management is partially or entirely delegated to the workers. They receive elementary stream items or data window segments from the emitter and are in charge of

managing the window boundaries by adding/removing the expired tuples. In other words, workers are active on sliding windows management too, thus their code is very similar to (coincides with) the sequential version. Emitter and collector act mainly as intelligent interfaces with respect to the workers.

All these characteristics can contribute in deriving variations of the proposed patterns. Each pattern will be presented in a section by itself, highlighting its applicability, limitations and impact. Patterns are intended to be applicable to time/count based window: for the sake of simplicity they will be exemplified on count based ones.

4.3 Window Farming

This first pattern exploits a simple intuition. Let's say that each window triggering implies the application of a function $\mathcal{F}$ on its content. Each window can be processed independently, that is the result of the computation on a window does not depend on the results of the previous windows. This is clearly true also considering windows that belong to different logical streams. Therefore, a simple solution is to adapt the classic *task farm* pattern to this domain, as sketched in Figure 4.1.

In this first variant we can assume an employment of *agnostic* workers. The emitter is in charge of buffering tuples coming from the stream and builds and updates the *canonical* copy of the windows, one for each logical substreams. In the figure we show an example with two keys X and Y. Tuples and windows are marked with unique identifiers. Once a window has been completely buffered, it is transmitted to a worker. The assignment must be aimed at balancing the workload. In the case that the function $\mathcal{F}$ has a low variance processing time we can use a simple *round-robin* policy. Otherwise an *on-demand* assignment can be a better solution: each worker signals to the emitter the availability to accept a new task. In the figure we adopt a round-robin strategy: windows ω_i^x and ω_i^y are assigned to worker j s.t. $j = (i + 1) \mod n$ where n is the number of workers.

Multiple windows of the same or of different substreams are executed in parallel by different agnostic workers. Workers receive a bulk of data (i.e. the tuples that are part of a window), apply the function $\mathcal{F}$ and discard the data. The emitter is responsible for receiving new tuples, inserts them into the related window and removes expired ones according to the window semantic. The collector functionality receives the results and may be responsible for reordering them. If a round robin distribution strategy is used, this will simply require to collect results from the workers in the same

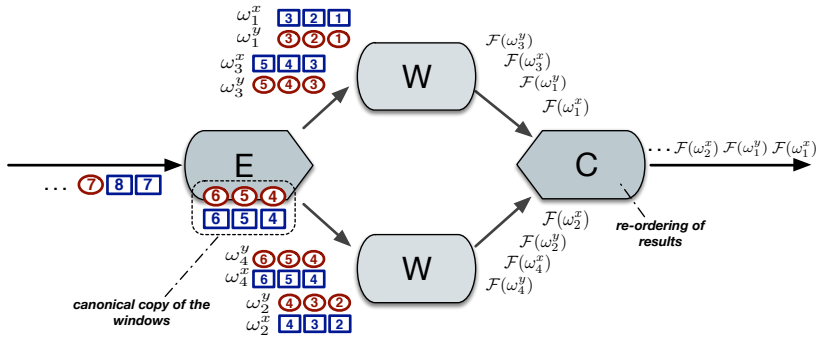


Figure 4.1: Window Farming with two workers. It has in input two logical substreams X and Y, whose elements are identified with squares and circles respectively. In the example $|\mathcal{W}| = 3$ and $\delta = 1$. ω_i^x represents the i-th window of substream X. $\mathcal{F}(\omega_i)$ represents the result of the processing function over a window.

order in which windows are scheduled to them. Otherwise collector should rely on the windows sequence numbers.

Different variants of this pattern can be obtained by adopting a different distribution and active workers. As a first optimization, instead of buffering entire windows and then transmit them, the distribution can be performed with a finer granularity. Single tuples (or small groups of consecutive tuples) can be transmitted by the emitter to the workers as they arrive from the input stream without buffering the whole window, i.e. on-the-fly. Each tuple is routed to one or more workers depending on the values of the window size and sliding parameters and the assignment policy of windows to workers. Figure 4.2 shows an example with three workers, window size $|\mathcal{W}| = 3$ and slide $\delta = 2$. Windows are assigned to the three workers in a round-robin fashion. Each tuple can be transmitted to *one* or *more* workers, depending to which worker(s) the corresponding windows are assigned to. For example the tuple x_4 is part only of window ω_2^x which is assigned to the second worker. Tuple x_5 belongs instead to windows ω_2^x and ω_3^x and thus is multicasted to the second and third worker.

The emitter, for each received tuple t, is now in charge of determining the windows on which the tuple belongs. The tuple is transmitted to worker j if t belongs to window ω_i and ω_i is assigned to worker j. Workers are now active in the management of the windows. They must be aware of the window semantic and assignment policy to manage the computation triggering and tuples expiration. In fact, if a round robin strategy is used by the emitter, then worker can handle incoming tuple like it is working on a window with size $|\mathcal{W}|$ and slide $n\delta$ where n is the number of workers. After the computation the first $n\delta$ tuples of the window can be safely discarded since

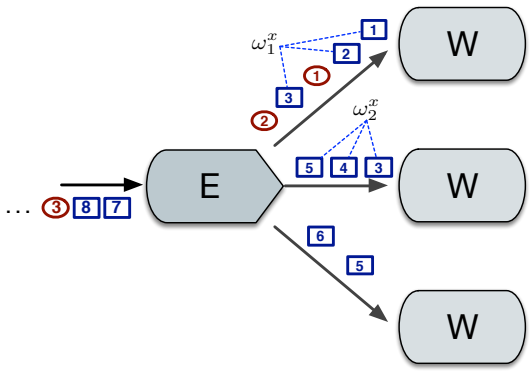


Figure 4.2: Window Farming with fine grained distribution. In the example $|\mathcal{W}| = 3$ and $\delta = 2$. Collector is omitted.

they are no more needed. This clearly holds also for time based windows, taking into account the tuple's timestamp. If a different strategy is used then it must be the emitter that communicates explicitly to the worker, to which the window is assigned, that the evaluation of $\mathcal{F}$ can be triggered.

This optimization reduces the buffering space in the emitter and its service time. The latter is important if the distribution of a window as a whole makes the emitter a bottleneck. In addition, fine grained distributions can be useful to reduce the latency and improve throughput if $\mathcal{F}$ is incrementally computable: the workers start the computation as new tuples arrive without needing to have the whole window. On the other hand, it is important to observe that tuples are replicated to distinct workers due to the fact that consecutive windows overlap. Here the term *replicated* assumes a different meaning according to the execution architecture: on multicore tuples replication can be avoided by sharing data, i.e. by passing memory pointers to the input tuples. In a distributed one the replication is real: if the same tuple must be sent to two different workers, two copies of it must be created.

A further optimization consists in assigning *batches* to workers. A batch is a set of $\mathcal{B} \geq 1$ consecutive windows of the same substream [Balkesen and Tatbul, 2011]. A tuple present in more than one window in the same batch is transmitted just one time to the corresponding worker. The key idea behind this strategy is to reduce the need for tuples replication to multiple partitions by reducing the overlap across those partitions.

Figure 4.3 shows an example with three workers and batches of two windows assigned in a round-robin fashion. Each tuple is multicasted to two workers instead of

three as in the case of standard assignment of single windows. Batching can increase latency and the buffering space in the emitter. This can be still mitigated by using fine-grained distributions.

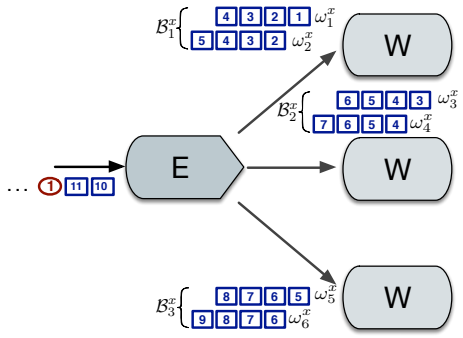


Figure 4.3: Window farming with batching distribution. In the example $|\mathcal{W}| = 4$, $\delta = 2$ and $|\mathcal{B}| = 2$. Collector is omitted.

In conclusion, we can summarize the characteristics of this pattern as follows:

Applicability The pattern can be applied to any window based stateful operator, also keyed ones. No particular property is required for the function $\mathcal{F}$.

Profitability Being a window parallel pattern, Window Farming is able to optimize the throughput. Load balancing can be easily obtained through proper assignment policies of windows to workers.

Issues In the *agnostic* workers version, the emitter may become a bottleneck. The same tuple can be replicated in several workers by potentially increasing the overall memory consumption. The pattern does not optimize latency.

Variations Using *active* workers, we can adopt a fine grained distribution (on-the-fly) to reduce the buffering space in the emitter and improve its service time. If $\mathcal{F}$ is incrementally computable this can reduce also the latency. *Batch-based* assignments can be used to reduce data replication.

4.4 Key Partitioning

Key Partitioning is a variant of Window Farming, characterized by a constrained assignment policy. The idea is to split the set of keys $\mathcal{K}$ into n partitions, where n is the number of workers. The emitter assigns windows to the workers based on the

value of the key: tuples with the same key are always routed to and handled by the same worker.

If windows are distributed as a whole, workers are *agnostic* of the window management. With fine grained distribution (see Figure 4.4), workers become *active* and manage the window boundaries. Results with the same key arrive to the collector *ordered*, i.e. partial ordering is automatically guaranteed.

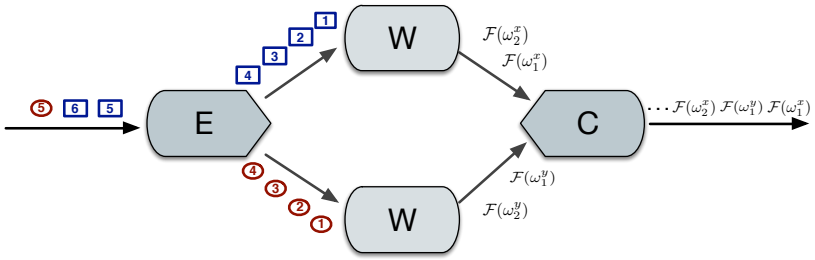


Figure 4.4: Key Partitioning with fine grained distribution. Substream X is routed to the first worker, substream Y to the second one.

This variant deserves to be considered a pattern *per se* due to its wide diffusion in the literature [Gedik, 2014] and modern SPEs [Apache Storm, 2016; Ballard et al., 2012]. Moreover, it can be used also when the state is a generic data structure rather than a window, e.g. *synopses*. In this case key partitioning is the only solution to preserve consistency of data structures, since all the tuples with the same key are assigned to the same worker.

The pattern expresses a limited parallelism: only windows belonging to different substreams can be executed in parallel, while the windows of the same substream are processed serially. Due to the fixed routing, load balancing becomes a problem when computation time is different per key or when there is skew in the distribution of the keys. In the latter case, if p_{max} is the highest frequency of key, the parallel pattern can scale up to $1/p_{max}$, assuming computation time equal for all the keys. This is due to the fact that at least one key is assigned to each worker. Only with a uniform distribution of keys, the maximum scalability is equal to the number of distinct key $|\mathcal{K}|$.

The main characteristics of the pattern are:

Applicability The pattern can be applied to any keyed stateful operator. No particular property is required for the function $\mathcal{F}$.

Profitability It is able to optimize the throughput. No data replication is necessary. Partial ordering is naturally guaranteed.

Issues Load balancing can be difficult or even impossible if the key distribution is very skewed. This is particular challenging in a dynamic context, as we will see in Chapter 6. The pattern does not optimize latency.

Variations Like in Window Farming, distributions with finer granularity reduce the buffering space in the emitter and improve its service time. This can reduce the latency if $\mathcal{F}$ is incrementally computable.

4.5 Pane Farming

The pane-based approach has been proposed for the centralized processing of sliding window aggregates in [Li et al., 2005]. The idea was to reduce the space and computation cost of sliding window aggregates by sub-aggregating and sharing computation. It has been already applied in [Balkesen and Tatbul, 2011] in a parallel context. It can be generalized to define a pattern with interesting properties in terms of throughput, latency and memory.

Each window is divided into non-overlapping contiguous partitions called *panes* of size $\sigma_p = gcd(|\mathcal{W}|, \delta)$. Each window ω is composed of r disjoint panes $\omega = \{\mathcal{P}_1, \ldots, \mathcal{P}_r\}$ with $r = |\mathcal{W}|/\sigma_p$.

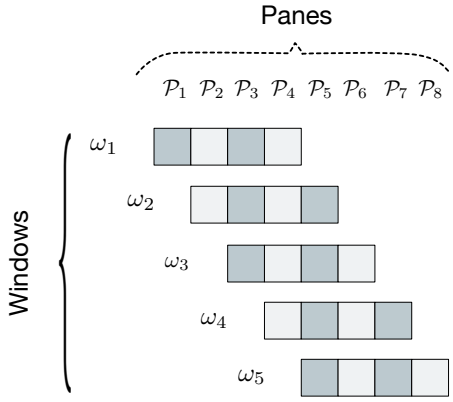


Figure 4.5: Sliding window composed by 4 different panes. Each pane is present in 4 consecutive windows.

This pattern can be applied if the internal processing function $\mathcal{F}$ can be decomposed into two functions $\mathcal{G}$ and $\mathcal{H}$ such that:

$$\mathcal{F}(\omega) = \mathcal{H}(\mathcal{G}(\mathcal{P}_1), \ldots, \mathcal{G}(\mathcal{P}_r))$$

i.e. $\mathcal{G}$ is applied to each pane and $\mathcal{H}$ is computed by combining the pane results. Therefore computations over consecutive windows can partially re-use pane results. Examples of computations that can be modeled in this way are *holistic* aggregates (e.g. median, mode, quantile), *bounded* aggregates (e.g. count and sum), *differential* aggregates (e.g. average) and many others [Li et al., 2005; Balkesen and Tatbul, 2011]. The idea of the pattern is sketched in Figure 4.6, exemplified in an unkeyed scenario. Like in Window Farming, a round robin or an on-demand policy can be used to assign panes to the workers. Panes are *tumbling subwindows* distributed to

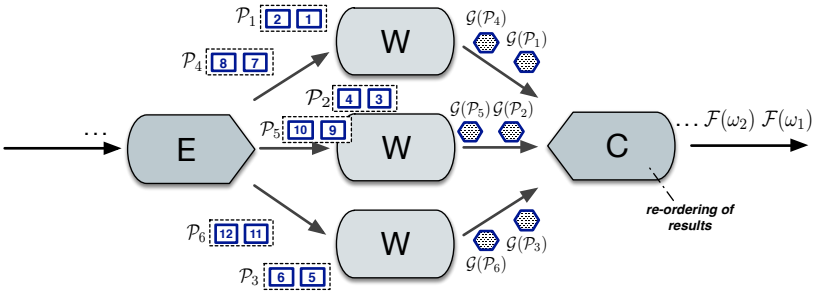


Figure 4.6: Pane Farming with unkeyed stream. Window has size $|\mathcal{W}| = 6$ and slide $\delta = 2$. Therefore each pane is composed by $\sigma_p = 2$ elements.

a set of agnostic workers applying the function $\mathcal{G}$ on each received pane. Denoting the Window Farming as W-Farm$(\mathcal{F}, |\mathcal{W}|, \delta)$, Pane Farming can be derived applying it as W-Farm$(\mathcal{G}, \sigma_p, \sigma_p)$. The collector is in charge of applying the function $\mathcal{H}$ over a sliding window of pane results, having size r and slide $\delta_p = \delta/\sigma_p$. If necessary, this stage can be further parallelized using Window Farming. In that case the whole pattern can be seen as a two-staged pipeline where each stage is parallelized using Window Farming, defined as:

$$\text{Pipe}(\text{W-Farm}(\mathcal{G}, \sigma_p, \sigma_p), \text{W-Farm}(\mathcal{H}, r, \delta_p))$$

This solution is depicted in Figure 4.7. The first stage has been parallelized with three workers while the second with two. Collector of the first stage and emitter of the first one are collapsed in the same entity.

Pane farming can also be used for keyed operators. Panes of different substreams are dispatched by the emitter to the workers of the first stage, while the corresponding

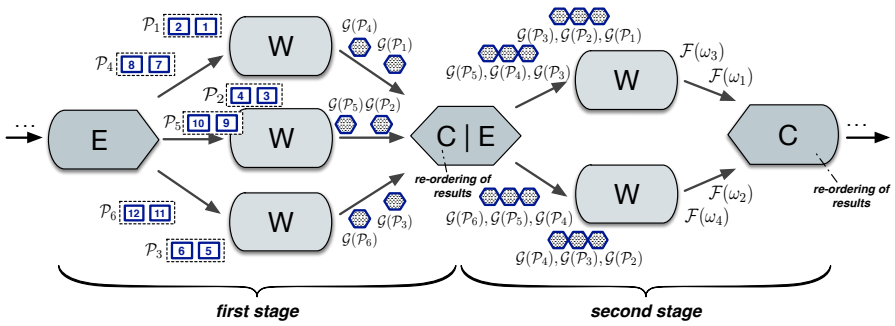


Figure 4.7: Pane Farming with collector parallelized using the Window Farming approach. The collector of the first stage and the emitter of the second one have been merged in a single `C|E` functionality.

windows are calculated in the second one. In this situation the evaluation of $\mathcal{H}$ can be parallelized using also the Key Partitioning schema.

Pane Farming belongs to the window parallel paradigms, since more than one window is executed in parallel in the workers. Therefore it improves throughput. The interesting property is that it is able to reduce also the latency by sharing overlapping pane results between consecutive windows. Being $T_{\mathcal{G}}$ and $T_{\mathcal{H}}$ respectively the processing time of the function $\mathcal{G}$ and $\mathcal{H}$, we can define the computation latency of the sequential version as:

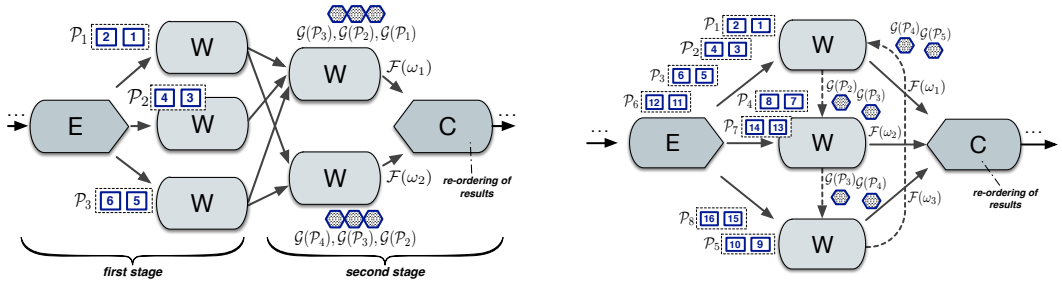
$$\mathcal{L}^{seq} = rT_{\mathcal{G}} + T_{\mathcal{H}}$$

In contrast the one of the pane based approach is:

$$\mathcal{L}^{pane} = T_{\mathcal{G}} + T_{\mathcal{H}}$$

due to the fact that panes are processed in parallel. When the last tuple of the window arrives, only the last panes remain to be processed (plus the aggregation function $\mathcal{H}$). Therefore, assuming that the function $\mathcal{G}$ has a low variance processing time, the latency reduction factor $\mathcal{L}^{seq}/\mathcal{L}^{pane}$ approaches r as $T_{\mathcal{H}} \to 0$.

Further pattern variations can regard the interaction between the two stages. The functionality `C|E` merges the pane results coming from the workers of the first stage and assigns windows of pane results to the worker of the second stage. This can be critical for latency and throughput. *Shuffling* can be used to remove this potential bottleneck: rather than merging the pane results and then distribute windows of them, workers of the first stage can multicast their pane results directly to the workers of the second stage (see Figure 4.8a). This clearly requires that workers of the first stage are aware of the assignment policy of windows of pane results.



(a) Pane Farming with shuffling

(b) Pane Farming with a ring topology

Figure 4.8: Variations of the Pane Farming pattern

Alternatively, the two stages can be merged by organizing the worker on a *ring* topology as suggested in [Balkesen and Tatbul, 2011] (see Figure 4.8b). Workers apply the function $\mathcal{G}$ on the received panes and the function $\mathcal{H}$ on their pane results and on the ones received by the previous workers on the ring. As explained in [Balkesen and Tatbul, 2011], there is always a way to assign panes to the workers such that a pane result can be transmitted at most to the next worker on the ring.

The properties of the Pane Farming pattern are:

Applicability The pattern can be applied on sliding window (also for keyed streams) provided that the function $\mathcal{F}$ can be expressed as $\mathcal{F}(\omega) = \mathcal{H}(\mathcal{G}(\mathcal{P}_1), \ldots, \mathcal{G}(\mathcal{P}_r))$.

Profitability There is no data replication in the first stage because panes are disjoint. In the second stage, replication can be reduced applying batching (see Window Farming). The pattern improves throughput and latency. Load balancing can be easily achieved through proper assignment policies of panes to workers.

Issues The pattern is not useful if the sliding factor δ is equal to one tuple.

Variations The computation of function $\mathcal{H}$ can be parallelized using a Window Farming approach (or Key Partitioning for the keyed scenario). *Shuffling* can be adopted to remove the `C|E` functionality, if it is a bottleneck. This came at the cost of additional communication cost in the workers of the first stage. The *ring* variant makes it possible to merge the two stages into a unique structure if communications between workers can be expressed.

4.6 Window Partitioning

Window Partitioning is the natural adaptation of the *map-reduce* paradigm to the case of data streams. Assuming that the computation can be expressed as a *map* over the window elements, the current window can be partitioned among n workers responsible for computing the internal processing function $\mathcal{F}$ on their partitions. A *reduce* phase may be necessary to aggregate/combine the final result of the window. This is a data parallel solution, since exactly *one window at a time is in execution*. We can have agnostic or active workers. In the former case, the emitter is responsible of buffering the whole window and scatters it to the various workers. Each worker receives a bunch of data, applies the function $\mathcal{F}$ and discards the data. More conveniently, the emitter can distribute single tuples to workers. In this case they are fully active in the window management, since they are responsible for adding new tuples to their partitions and removing expired ones according to the window size and slide parameters (for both *count* and *time* based cases).

Figure 4.9 exemplifies the pattern. Tuples are distributed in a round-robin fashion to two workers. This is possible if the internal processing function can be performed by the workers in parallel on *non-contiguous* partitions of the same window. If this is not the case, other distributions preserving the contiguity of data must be used. In a keyed scenario, the workers maintain a window partition per logical substream.

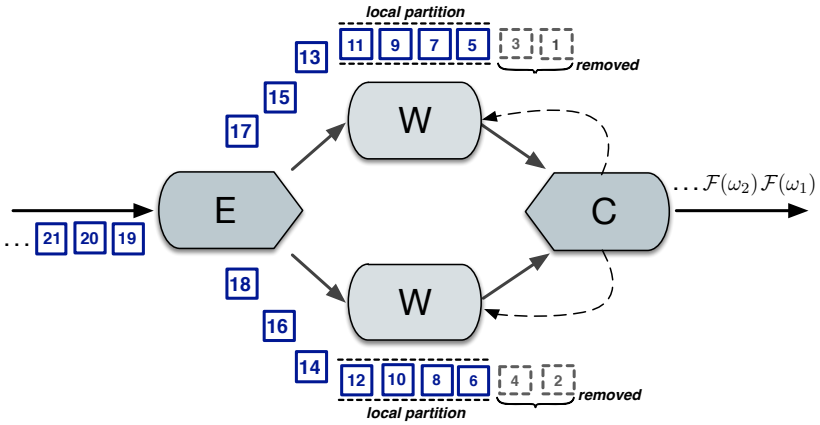


Figure 4.9: Window Partitioning with two workers and one key. In the example $|\mathcal{W}| = 8$ and $\delta = 4$

Once the last tuple of the slide has been received, it is transmitted to one worker and a special meta-tuple is multicasted to all the workers in order to start in parallel the map function on the partitions. If required by the computation, the workers

execute the local reduce on their partitions and communicate the local reduce results to collector as depicted in figure. The collector is in charge of computing the global reduce result and send it in output. Depending on the computation semantics, the reduce phase $\oplus_r$ can be performed in two different ways:

- *asynchronously* with respect to the computation of the workers. In this case the result of the global reduce is not needed by the workers. An example is the computation of algebraic aggregates, e.g. "finding the number of tuples in the last 1000 tuples such that the price attribute is greater than a given threshold". In this case $\mathcal{F}$ is the `count` function (applied over the workers' partitions) and $\oplus_r$ is the `sum` function (to compute the global value);
- *synchronously* with respect to the computation of the workers, which need explicitly to receive the global reduce result from the collector (dashed arrows in Figure4.9). A similar interaction could be required by the computation semantics. For example a second map-reduce phase must be executed over the window, like in the case "finding the tuples in the last 1000 tuples such that the price attribute is higher than the average price in the window".

It is worth noting that the computations that can be parallelized through pane farming are a subset of the ones on which window partitioning can be applied. This pattern is able to improve throughput and optimize latency. The latency reduction is proportional to the partition size, which depends on the number of workers. This is an important difference with Pane Farming that gives a latency reduction independent from the parallelism degree (it depends on the number of panes per window).
The properties of the Window Partitioning pattern are:

Applicability The pattern can be applied when the computation is expressed as a *map* followed by an optional *reduce* phase. The repetitive application of maps and reduces can be captured as well. It is applicable also to keyed streams.

Profitability The pattern improves throughput and optimizes latency, in a proportional way with respect to the number of used workers. Tuples are partitioned without *data replication*.

Issues Load balancing can be difficult to achieve if the internal processing function has a high variance processing time depending on the data values.

Variations Distribution can be performed on-the-fly by using *active* workers, reducing the buffering space in the emitter and its service time. This can be profitable to further reduce latency if the function $\mathcal{F}$ is incrementally computable.

4.7 Nesting of patterns

An interesting property of parallel paradigms is that they can be composed with each other to form complex computations. We have already encountered an interesting case of composition in Pane Farming: the whole pattern can be defined as the composition of two Window Farming patterns applied for the parallelization of $\mathcal{G}$ and $\mathcal{H}$ functions, respectively.

Nesting can be considered as a potential solution to balance pros and cons of the different patterns. We detail here two interesting nestings that apply a Window Partitioning schema to internally parallelize the workers of a window parallel pattern. Others schemes are clearly possible.

Window Farming & Window Partitioning We can combine this two patterns resulting in Window Farming with *macroworkers* internally implemented according to the Window Partitioning pattern. This approach can be useful to sustain the actual speed of the input stream with lower latency than using Window Farming alone. Window Partitioning avoid data replication inside macroworkers. Batching can be used at the outermost level to reduce data replication also between macroworkers. The distribution, at both levels, can be performed on-the-fly. The whole solution benefits from the easier load balancing at the outermost level.

Key Partitioning & Window Partitioning In a similar way of the previous case, we can have a Key Partitioning with macroworkers implemented using Window-Partitioning. Clearly, this solution can be applied in a keyed scenario only. It increases throughput and lowers latency. In contrast to the previous nesting, this results in optimal memory occupancy, since there is no tuple replication at any level. On the other hand, load balancing can be hard as it is critical for both the parallel patterns.

4.8 Exporting the patterns

It is interesting to understand if the proposed patterns are exploitable in existing frameworks or if they can be implemented on these systems. In this section we will elaborate on this topic, trying to highlight how to incorporate the proposed solutions in available high level parallel frameworks and which kind of actions would be required.

Existing Stream Processing Engines allow the programmer to express only a subset of these patterns, usually without some of the possible optimizations and variants.

In Storm [Apache Storm, 2016] there are not pre-defined built-in stream operators: the programmer has to define the operator logic from scratch. The proposed patterns could be implemented on top of existing mechanisms and offered to programmer as a library for Storm. The concepts of grouping can be used to specify how input tuples are distributed among multiple replicas of an operator. For keyed stateful operator *field grouping* is a naturally choice: it assures that tuples with the same key are always sent to the same replica. Compared with our work this is similar to Key Partitioning, in which bolts are essentially active workers. The other parallel patterns introduced could be implemented by using active workers and *custom grouping* policies implementing user-defined distributions.

IBM InfoSphere [Ballard et al., 2012] provides a rich set of built-in operators and the possibility to implement user-defined operators. Windowing is primitive in IIS. This, combined with the hash based distributions offered by the framework, is more or less or Key Partitioning. However, there is no support for customizable tuple routing. This, combined with the closed source approach, render impossible to mimic the other patterns.

In Spark Streaming [Apache, 2016] programmers can rely on a support to time based window. The `count/reduceByWindow` operators are similar to the Window Partitioning pattern, while `count/reduceByKeyAndWindow` recalls the nesting of Key and Window Partitioning. This results in only a partial coverage of the various computations possibilities that we introduced in this chapter, since the `reduce` operators clearly require that the function to be applied has certain characteristics. Moreover, in Spark windows have a particular semantics: they can be only time based and they are essentially micro-batch of contiguous data. Only when the batch is complete, the computation can start. This resembles more or less the agnostic workers structure, in which the emitter schedules window partitions. No optimizations such as finer grained distributions are possible. Introducing this concepts and functionalities requires intervention at the runtime support level.

Finally, Flink [Apache Flink, 2016] has a complete support to window but lacks in parallelism exploitation for window based operators: window on keyed stream are evaluated in parallel, one task per key. This is more or less equivalent to the Key Partitioning approach. For unkeyed stream, computations are not performed in parallel. A possible integration of pattern requires interventions on its runtime system and their promotion to the APIs level.

With respect to SPEs, skeletal frameworks could represent an alternative solution amenable of being integrated with patterns for window based stateful operators. The proposed patterns have to be provided as *skeletons*, but this can be done in a natural way from the discussion done in this chapter. Skeletons require to be specialized by the programmers by indicating the function and window parameters.

A first step towards the integration of the proposed pattern in the Fastflow framework has been done in the experimental phase. We leveraged the hierarchical structure of Fastflow and patterns have been implemented and tested using the basic mechanisms (non blocking queues) offered by Fastflow. A full integration in the Fastflow environment could regard their promotion in top level patterns. In frameworks in which exploiting low level mechanism is not possible or it is not so easy to define additional skeletons, internal implementation modifications are required.

Table 4.1 summarizes the relations discussed so far between the proposed patterns and the considered frameworks.

Framework	**Window Farming**	**Key Partitioning**	**Pane Farming**	**Window Partitioning**
Apache Storm	▲	✔	▲	▲
IBM Infosphere	✘	✔	✘	✘
Spark Streaming	▼	✔/▼	▼	✔/▼
Apache Flink	▼	✔	▼	▼
Fastflow	▲	▲	▲	▲

Table 4.1: Different patterns and their relations with the discussed frameworks. A pattern can be already provided by the framework (✔), it can be implementable on top of the framework (▲) or via modification to the framework runtime (▼). Finally, it could be not exploitable at all (✘).

4.9 Experiments

In this section we will describe the evaluation of the proposed patterns on a shared memory machine. The objective of this experimental evaluation is to asses the performance of the proposed patterns for window based stateful DaSP operators. First of all, we want to show that these patterns could ease the application development but this does not occur at the expense of performances. Secondly, we want to empirically

evaluate the different impacts on performance metrics (i.e. throughput and latency) that the different patterns are able to achieve.

A prototypal version of the the patterns has been implemented on top of *Fastflow*, a C++ based skeleton frameworks (see Section 3.3 for a brief introduction). Fastflow has a three tiers organization: *high-level patterns*, *core patterns* and *building blocks*. In this phase we operate at the building blocks level. Parallel entities (emitter, workers and collectors) are implemented as threads that cooperate through *non-blocking* and *lock-less* queues exchanging pointers to shared memory areas. Threads are pinned on distinct cores of a multicore architecture. More implementation details are available in Section 4.9.1.

The target architecture is a dual CPU Intel Xeon E5-2650, Sandy-Bridge based architectures, composed of 16 hyperthreaded cores operating at 2GHz. Each core has a private L1 (32KB) and L2 (256KB) cache. Each CPU is equipped with a shared L3 cache of 20MB. The machine has 32 GB of RAM. It runs a Linux based operating system. The used compiler is `gcc` (version `4.8.1`), programs are compiled with the `-O3` compiler flag. The used Fastflow version is the `2.0.5`.

In the following experiments, the computation is interfaced with a *Generator*, which is in charge of generating the incoming data, and a *Consumer* thread, which is in charge of receiving the results (see Figure 4.10). They are both executed in the same machine, pinned on different cores and communicate with the application through TCP/IP sockets. Unless otherwise specified, we will not exploit the hyper threading facility presents the processor. Thus, 12 is the maximum number of workers that we can use in our testing.

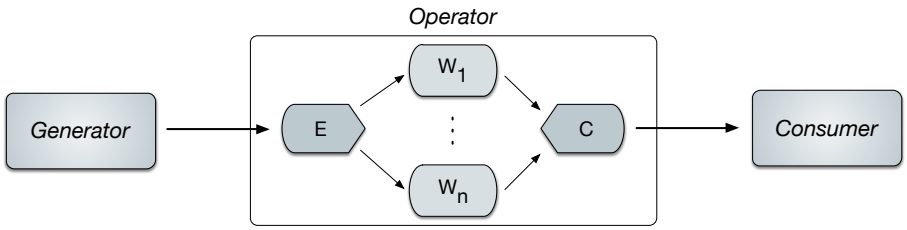


Figure 4.10: Computation schema

4.9.1 Implementation details

In the following, we detail the internal implementations of the discussed parallelization schemas for window based operators. The patterns have been implemented for keyed streams using an on-the-fly distribution. When required, the assignment of the

windows to the workers is done using a round robin policy. Possible data replication issues, implied by the pattern used, must be considered only a logical possibility since we target shared memory architecture. As mentioned, the proposed implementations rely on the building block mechanisms of Fastflow. Alternative implementations could be easily realized also exploiting classical threads and generic shared queues among the various entities, possibly at the expense of some performance degradation. We recall that $|\mathcal{W}|$ and δ indicates respectively the size and the slide of the used windows, while n refers to the number of workers.

Concerning the used data structures, we have introduced an abstract class `Window` that defines a set of virtual methods common to all the various window types discussed so far. The class is defined by means of two generic data types: a `tuple_t` type, that represents the type of the window elements, and a `result_t` that is the type of the result produced after that the window computation is executed. Among the various methods we should mention:

- the `insert(t)` method that inserts a tuple (passed as argument by const reference) into the window buffer;
- the `expire()` method that evicts all the expired tuples;
- the `isComputable()` method that returns a boolean value indicating if the window content is ready to be computed;
- the `compute()` method, that triggers the computation over the window content and returns the corresponding result.

Starting from this, we have defined an inherited `CBWindow` class that defines the behavior of a generic count based window. Its constructor takes as parameter the window size and slide. The window buffer is implemented as a circular buffer, preallocated at creation time. The compute method encapsulates the business logic of the computation (see the different benchmark used) and returns the result of the computation.

In general, once δ tuples have been received, the window content is ready to be computed (i.e. the method `isComputable()` returns true). Expired tuples (i.e. no more needed for the computation) are evicted after the computation. An exception to this modus operandi regards the windows used for the Window Partitioning pattern: in this case the computation can be started when the meta-tuple (sent by the emitter; see Section 4.6) is inserted into the window. To support this case we have

defined an additional class `CBPWindow`, whose `isComputable()` method is defined accordingly.

The emitter, workers and collector entities are connected through *Single Producer/ /Single Consumer* queues. In the following we will briefly detail their logic according to the implemented pattern. The emitter, is in charge of receiving the input tuples, assigning to them a sequence number (independent for the various keys) and distributing them (by passing a pointer) to a subset of workers according to the pattern semantics:

- for the Window Farming this requires to determine the *windows* to which the tuple belongs. This could be easily done by leveraging on the tuple's sequence number: the tuple is transmitted to the various workers according to the defined window characteristics. For the Pane Farming approach, the emitter has a quite similar behavior: in this case it has to determine the *pane* to which a tuple belongs (taking into account that panes are tumbling subwindows; see Section 4.5);
- in the Key Partitioning pattern, *keys* are partitioned among the available workers. Tuples with the same key are always routed to the same worker. The emitter maintains the association between keys and workers in a routing table explicitly stored;
- in the Window Partitioning pattern, tuples are distributed in a round-robin fashion to the workers. Once that the last tuple of the slide has been received, a special meta-tuple (for that particular key) is sent to all the workers in order to start the computation.

Independently from the implemented pattern, workers behave in a similar way: they receive a tuple from the emitter, insert it into the proper window (according to the key attribute) and if the window content is ready to be processed they invoke the `compute()` method and send the produced result to the collector. What changes, from one pattern to the other, is the type or definition of window used. Window Farming, Key Partitioning and Pane Farming use the `CBWindow` class, with different sizes or slides. In Window Farming, due to the round robin assignment policy of windows to workers, the slide of the window is defined as the maximum between $n\delta$ and $|\mathcal{W}|$ (see Section 4.3). In Key Partitioning, workers use windows whose size and slide are the ones defined by the user. In the Pane Farming case, the workers use a tumbling count based windows over which they apply the function $\mathcal{G}$: in this case the pane

size (σ_p) is used as both window size and slide. On the other hand, for the Window Partitioning pattern workers use the `CBPWindow`, whose computation is triggered by the arrival of the meta-tuple.

Finally, for the collector we distinguish two kinds of logic:

- for Window Farming and Key Partitioning, it receives the results produced by the workers and forward them to the output stream. In the former case, to preserve ordering the results are collected from workers in the same order used to schedule windows;
- for Pane Farming and Window Partitioning, the collector receives the partial results and produces the final result by applying an additional computation. In the former case the collector is in charge of applying the function $\mathcal{H}$ over a sliding window of pane results, having size $r = |\mathcal{W}|/\sigma_p$ and slide $\delta_p = \delta/\sigma_p$. Therefore it can use again a `CBWindow` of the proper size and slide. In the latter case the collector applies the reduce functions over the partial results produced by the workers.

4.9.2 The synthetic benchmark

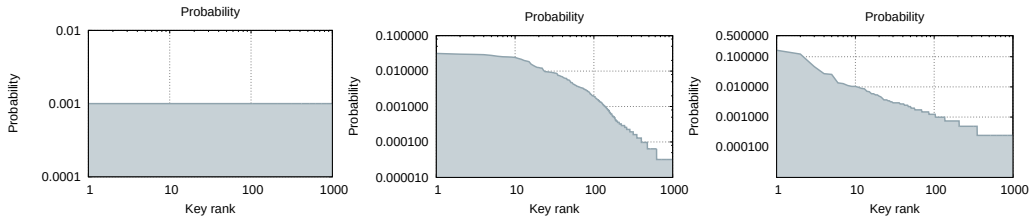
The benchmark computes a suite of statistical aggregates used for algorithmic trading, with $|\mathcal{K}| = 1000$ stock symbols and count-based windows. Experiments were performed with different window and slide sizes. In the following we indicate as *reference window* the one having $|\mathcal{W}| = 1000$ and $\delta = 200$.

Each tuple is a *quote* from the market, containing information such the symbol ID, the price and volume for ask and selling activities. It is stored in a record of 64 bytes. To use Pane Farming and Window Partitioning, the synthesized computation $\mathcal{F}$ is composed by a function $\mathcal{G}$ and $\mathcal{H}$. According to the presented descriptions, in the case of Window Farming and Key Partitioning patterns, the functions are both executed by the workers. For Pane Farming and Window Partitioning patterns, workers execute the function $\mathcal{G}$ while collector computes function $\mathcal{H}$ (it acts as *reduce* for the Window Partitioning pattern). For the reference architecture and windows we have that the computation times are equal to $T_{\mathcal{G}} =\sim 1500\mu$sec (per subwindow of 200 elements) and $T_{\mathcal{H}} =\sim 20\mu$sec. This result in a global computation time of $T_{\mathcal{F}} =\sim 7700\mu$sec. The computation times are slightly equal for all the keys.

We want to compare the different patterns in various scenario (window size and slide) and different parallelism degrees, i.e. number of workers. In the following, we

will identify the different configurations in terms of window size and slide indicating these values as $\langle |\mathcal{W}|, \delta \rangle$. Each test ran for 180 seconds. For each parallelism degree we determine the highest input rate (throughput) sustainable by the parallelized operator. To detect it, we repeat the experiments several times with growing input rates. The generator checks the TCP buffer of the socket towards the operator: if it is full for enough time, it means that the current operator configuration is a bottleneck. It stops the computation and the last sustained rate is recorded.

Input data is produced by the generator by randomly choose the attribute values of the tuples. Their generation rate is deterministic and it is given by the currently tested rate. Keys probability of appearance follows a given probability distribution passed to the generator. In order to study also the load balancing characteristics of the various patterns, we create three different scenarios according to different key probability distributions. In Figure 4.11 we report for each key the related probability of generation. Keys are ordered by rank, i.e. from the most frequent to the less frequent. Logarithmic scales are used to ease the readability. In the first scenario (Figure 4.11a) they keys probabilities are uniformly distributed with $p = 10^{-3}$ In the second one (4.11b) we use a *skewed* distribution with $p^{max} = 0.03$. The last case is a *very skewed* distribution with $p^{max} = 0.16$ (4.11c). In the following we will use



(a) Uniform distribution **(b)** Skewed distribution **(c)** Very skewed distribution

Figure 4.11: Keys probability distributions

the `WF`, `KP`, `PF` and `WP` abbreviations to refer to Window Farming, Key Partitioning, Pane Farming and Window Partitioning patterns respectively. We will firstly show the results obtained by the single patterns by changing the window size, slide size and distributions. In the subsequent section we will compare them.

Patterns evaluation

For each parallelism degree n, we denote with λ^n_{max} the maximum arrival rate that the parallel computation is able to sustain without being a bottleneck. If the workload is perfectly balanced among workers, the ideal service time T_S of the parallel operator

(Definition 4.2.1) with n workers is equal to:

$$T_S = T_{\mathcal{F}}/n \tag{4.1}$$

for the WF and KP cases. For PF and WP, provided that the computation of $\mathcal{H}$ has a low cost and collector is not a bottleneck, we have:

$$T_S = T_{\mathcal{G}}/n \tag{4.2}$$

A parallel operator is not a bottleneck if it is able to sustain the current stream rate, i.e.

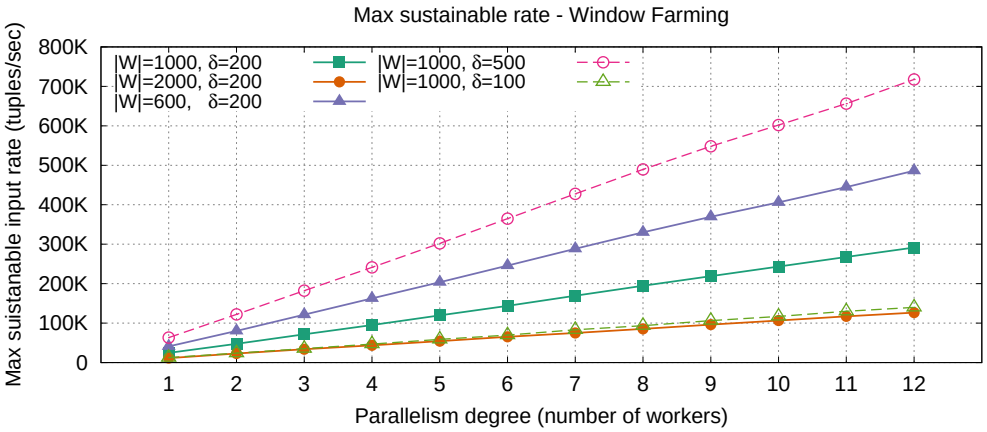
$$T_S \leq T_A \tag{4.3}$$

In this case T_A refers to the inter-arrival time of triggering tuples (of any key). The term *triggering tuple* denotes a tuple that triggers operator's internal processing logic. In this case, due to the use of sliding window, it is a tuple that lets the window slide and therefore activates the computation. Therefore, being λ the incoming input rate we have that $T_A = \delta/\lambda$. Since we know the calculation time and the slide we can easily derive the maximum sustainable rate by replacing in Equation 4.3 and derive the maximum sustainable rate as:

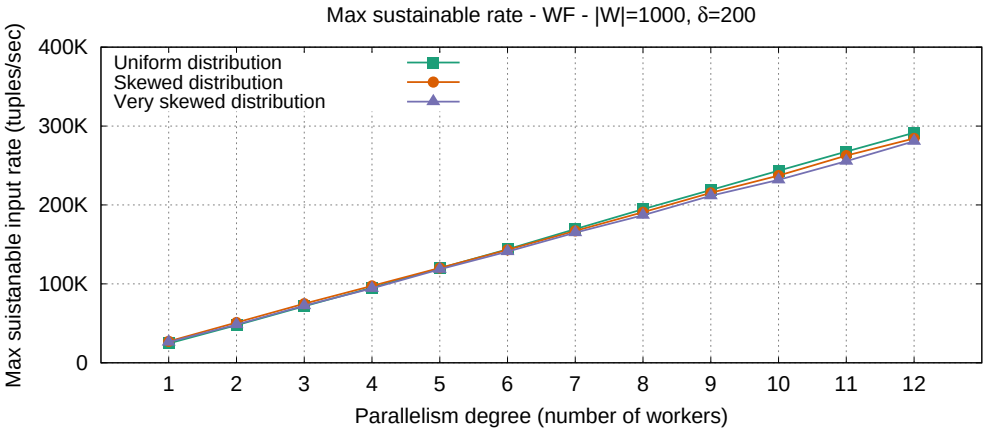
$$\lambda^n_{max} \leq \begin{cases} n\delta/T_{\mathcal{F}}, & \text{for WF and KP.} \\ n\sigma_p/T_{\mathcal{G}}, & \text{for PF.} \\ n\delta/T_{\mathcal{G}}, & \text{for WP.} \end{cases} \tag{4.4}$$

Figure 4.12a shows the behavior of WF while varying the window size (solid lines) and the slide size (dashed lines) with respect the reference window. In WF the window computation is triggered at every slide and regards the whole window content. The computation time is proportional to the number of elements contained in the window. For this reason the maximum sustainable rate decrease/increase with larger/smaller window sizes ($T_{\mathcal{F}}$ scales proportionally). In contrast, when we increase/decrease the slide of the window, the inter-arrival time of the triggering tuple will increase/decrease accordingly. This impacts the maximum sustainable rate as indicated in Equation 4.4, resulting in higher or lower values.

It is worth noting that although the computation with windows $\langle 2000, 200 \rangle$ and $\langle 1000, 100 \rangle$ should have led to the same sustained rate, we have that the latter configuration is able to achieve a slightly higher value with the increasing of the parallelism degree ($\sim$ 10% with 12 workers). This is probably due to a higher memory hierarchy



(a) Maximum sustainable rate per parallelism degree for the Window Farming pattern. In solid lines are depicted the results obtained with the reference window $\langle 1000, 200 \rangle$ and changing the window size only ($\langle 600, 200 \rangle$ and $\langle 2000, 200 \rangle$). With dashed lines are reported the results obtained in configurations in which we change the slide while maintaining the window size ($\langle 1000, 100 \rangle$ and $\langle 1000, 500 \rangle$).



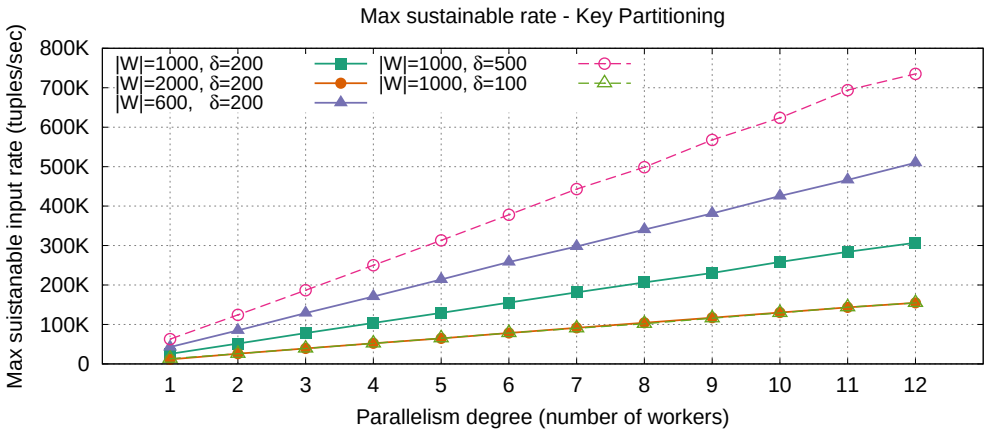
(b) Maximum sustainable rate for Window Farming using the reference window with different keys probability distributions.

Figure 4.12: Results for the Window Farming pattern

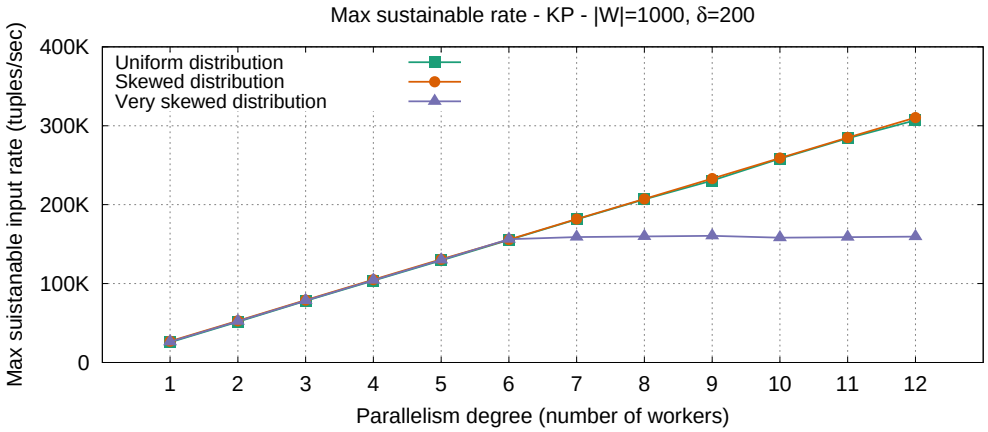
utilization in the $\langle 2000, 200 \rangle$ configuration. In Figure 4.12b are reported the sustained rates for the reference window and different key distributions. As expected, the values are similar thanks to the good load balancing achievable by the `WF` pattern.

In Figure 4.13, are reported the results for the `KP` case. They are qualitatively similar to the ones of the `WF` case. Figure 4.13b highlights the load balancing issues of this pattern. As mentioned in Section 4.4, the scalability of the pattern is limited by the probability of appearance of the most present key. In the very skewed scenario this is equal $p^{max} = 0.16$, therefore the scalability is limited to 6.25 and the sustainable

input rate stop to increase for $n > 6$.



(a) Maximum sustainable rate per parallelism degree for the Key Partitioning pattern. The results for the configurations $\langle 2000, 200\rangle$ and $\langle 1000, 100\rangle$ are superposed.



(b) Maximum sustainable rate for Key Partitioning and the reference window with different key probability distributions

Figure 4.13: Results for the Key Partitioning pattern

Figures 4.14 and 4.15 show the results for the `PF` and `WP` patterns respectively. We do not report the comparison using different keys probability distributions since their values are pretty much identical among the various possibilities. Interestingly, in the `PF` solution variations of the window or the slide size have little impacts: the obtained maximum sustainable rates are almost identical in all the considered cases. This is inline with Equation 4.4 provided that, like in the performed experiments, the computation of $\mathcal{H}$ has a low cost. In this case, the calculation time is essentially $T_{\mathcal{G}}$. It and the triggering tuple frequency depend on the pane size (the window slide in these cases). Varying only the size of the window has no effect on these two parameters.

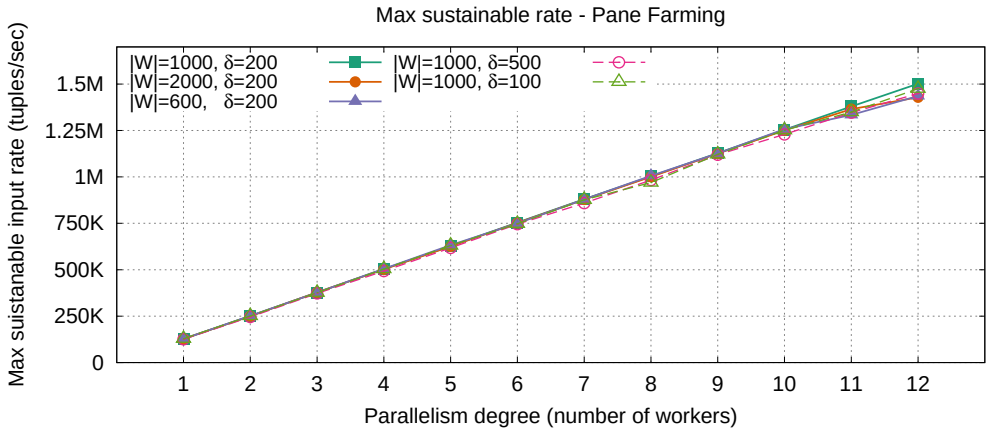


Figure 4.14: Maximum sustainable rate per parallelism degree for the Pane Farming pattern.

On the other hand, changing the window slide change the calculation time and these changes are compensated by a modification of the inter-arrival time. Increasing the window slide will led to a greater pane size and therefore a greater $T_{\mathcal{G}}$. However this will also decrease the inter-arrival time of the triggering tuples.

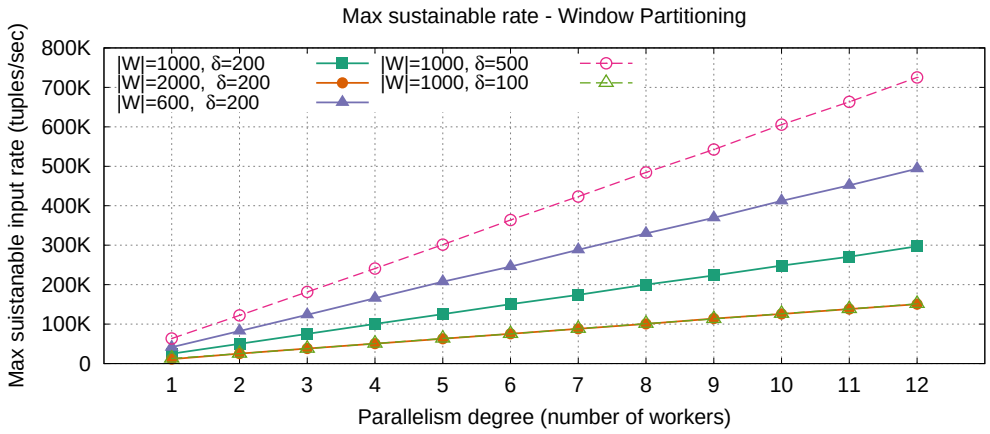
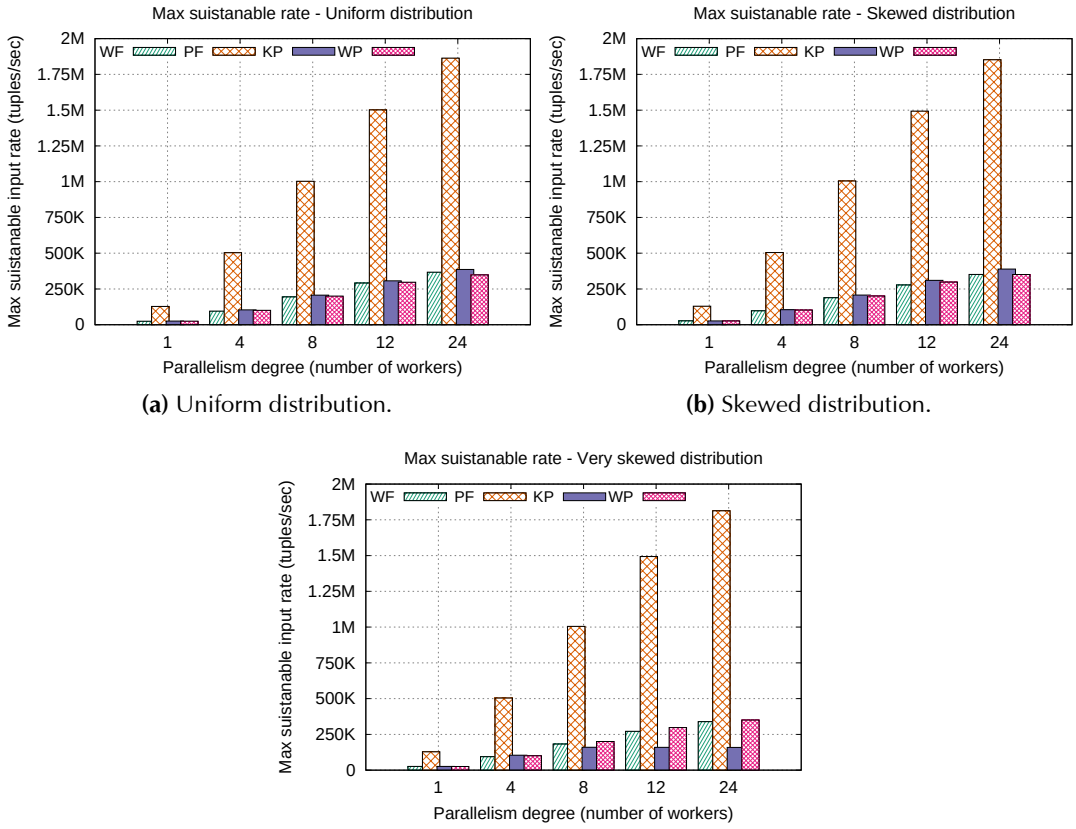


Figure 4.15: Maximum sustainable rate per parallelism degree for the Window Partitioning pattern. The results for the configurations $\langle 2000, 200\rangle$ and $\langle 1000, 100\rangle$ are superposed.

Patterns comparison

In this section we provide a comparison of the various patterns, summarizing the results previously reported for the case of the reference window $\langle 1000, 200\rangle$ and the various keys probability distributions. For the sake of completeness, we report also

the results with two worker threads per core ($n = 24$) which is the best hyperthreaded configuration found in our experimental settings.



(a) Uniform distribution.

(b) Skewed distribution.

(c) Very skewed distribution.

Figure 4.16: Maximum sustainable throughput for Window Farming (WF), Pane Farming (PF), Key Partitioning (KP) and Window Partitioning (WP) with different keys probability distributions.

In the first two scenarios (Figure 4.16a and 4.16b) KP reaches a slightly higher rate than WF and WP due to better reuse of window data in cache (each worker is responsible for a set of key and related windows). As already pointed out, in the very skewed distribution (Figure 4.16c) the maximum rate sustained by KP stops to increase with more than 6 workers, while WF and WP easily handle the unbalanced distribution. In all the cases the best results are achieved with PF: throughput is 5 times higher than the other patterns. The reason is that PF is able to share the pane results in common between consecutive windows of the same key.

In Table 4.2 are reported the best scalability results. In this case the scalability with n workers is computed over the maximum sustained rate as the ratio $\lambda^n_{max}/\lambda^1_{max}$.

Pattern	Uniform	Skewed	Very Skewed
WF	12:**11.85**, 24:**14.93**	12:**10.25**, 24:**12.95**	12:**10.37**, 24:**12.94**
KP	12:**12.02**, 24:**15.12**	12:**11.53**, 24:**14.46**	12:**6.08**, 24:**6.07**
PF	12:**11.77**, 24:**14.91**	12:**11.53**, 24:**14.76**	12:**11.61**, 24:**14.58**
WP	12:**11.83**, 24:**13.87**	12:**11.24**, 24:**13.24**	12:**11.61**, 24:**13.67**

Table 4.2: Scalability with 12 and 24 workers. Syntax ParDegree:**Scal**.

Latency

In order to study the latency impact of the various patterns, we compared them in an execution scenario in which all of them are able to carry out the computation without being a bottleneck. We use the reference window, with an input rate of $200K$ tuples/sec and key distribution uniform. To not being bottleneck we use 10 workers for WF, KP and WP and two workers with PF. The latency is reported in Figure 4.17: in the graph it is plotted using two logarithmic scales: the one on the left is used for PF and WP, the scale on the right for WF and KP. As expected WF and KP have similar

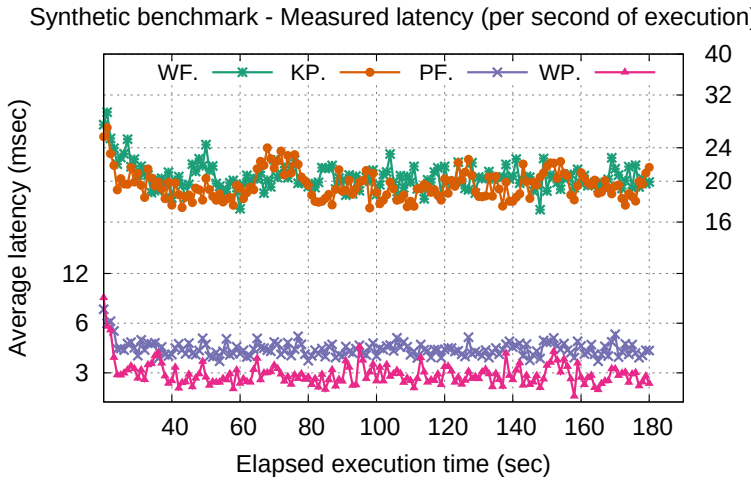


Figure 4.17: Latency measured, for second of execution, with the various patterns. The scale on the left refers to the values obtained with PF and WP, the scale on the right is used for WF and KP.

latencies because each window is processed sequentially. PF has a latency 5 times lower than WF and KP. As discussed in Section 4.5 the latency reduction factor given by the Pane Farming approach is roughly equal to the number of panes per window

(5 in this case) if $T_{\mathcal{H}} \sim 0$ as in this benchmark. In contrast `WP` produces a latency reduction proportional to the parallelism degree (and hence partition size). With 10 workers the latency is 27.53% lower than `PF`.

4.9.3 Time-based skyline queries

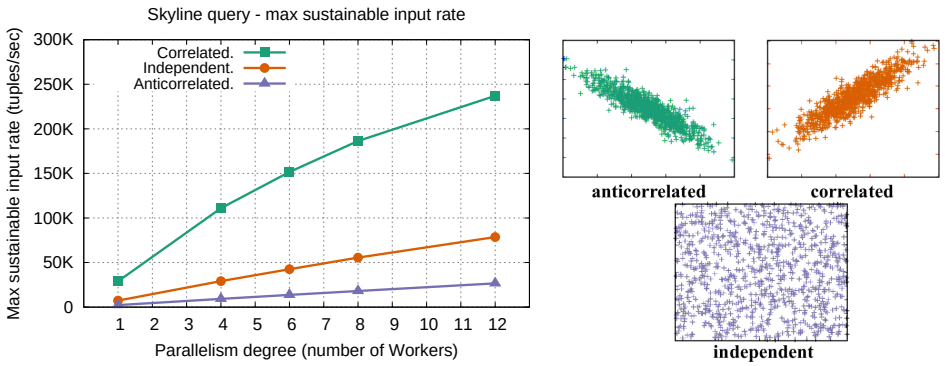
To conclude the experimental section we study a real-world continuous query computing the *skyline* set on the tuple received in the last $|\mathcal{W}|$ time units. The window slides (and therefore the computation is triggered) every time that a tuple arrives, resulting in a *tuple-by-tuple* time based window. We assume an unkeyed input stream. Skyline queries are preference queries frequently used in multi-criteria decision making to retrieve interesting points from large datasets. Recently, skyline queries have been computed over continuous data streams according to sliding window models. A skyline query returns the points that are not dominated by any other point. More formally, the single input x represents a d-dimensional point $x = \{x_1, x_2, \ldots, x_d\}$. Given two points x and y, we say that x $dominates$ y if and only if $\forall i \in [1, d]\ x_i \leq y_i$ and $\exists j \mid x_j < y_j$. A point belongs to the skyline set if there not exists any *dominator* in the current window composed by the points received in the last $|\mathcal{W}|$ time units.

On static dataset, the computation can be described as a *map-reduce*, in which a local skyline is computed for each partition of the dataset and the final skyline is calculated from the local ones (it exploits the skyline associativity). Thus in the data stream processing context, the natural pattern for this computation is the Window Partitioning: each worker receives a partition of the window and performs the local computation. Collector, receives the partial results (i.e. the skyline set of each worker partition) and computes the final results (the global skyline). It is important to note that the skyline algorithm performs an intensive *pruning* phase [Tao and Papadias, 2006]: tuples in the current window can be safely removed before their expiring time if they are dominated by a younger tuple in the window. In fact, these points will never be able to be added to the skyline, since they expire before their younger dominator. Pruning is fundamental to reduce the computational burden and memory occupancy. However it can produce sever load unbalance because the partition sizes can change very quickly at run time, even if the distribution evenly assigns new tuples to the workers.

Figure 4.18a shows the maximum suistanable input rate with the three point distributions studied in [Tao and Papadias, 2006]: `correlated`, `anticorrelated` and `independent`. Each distribution (represented in Figure 4.18b) is characterized by a different pruning probability. In the correlated case a small set of points dominate

the others and the pruning phase is very intensive. The anticorrelated case in on the opposite, with a large number of points that are part of the skyline set. The third one is an intermediate case with points uniformly distributed in the space.

We use a window of 60 seconds for the correlated case and 10 seconds for the anticorrelated and independent cases. Each new tuple is assigned to the worker with



(a) Maximum sustainable input rate **(b)** Point distributions

Figure 4.18: Skyline query: distribution of points and maximum sustainable input rate per parallelism degree with the Window Partitioning pattern. $|\mathcal{W}| = 60$ seconds for the correlated case and $|\mathcal{W}| = 10$ seconds for anticorrelated and independent distributions.

the smallest partition to try to balance the workload. Even if it works on a greater windows, the correlated case is the one with the highest sustained rate: the partitions are smaller (due to an intensive pruning) and therefore the computation has a finer grain. Load balancing is the most critical issue: with 12 workers scalability is 8.1, 10.7 and 11.6 in the correlated, independent and anticorrelated cases. With higher pruning probability is harder to keep the partitions evenly sized, highlighting the need of proper load balancing techniques.

4.10 Summary

In this chapter we presented four different patterns for windowed based computations. The proposed patterns exhibit differences in their applicability, impacts on the performance and limitations that are briefly summarized in Table 4.3.

The patterns have been implemented in a shared memory architecture using the Fastflow framework. The benchmark experimentally demonstrate the different features of the parallel patterns. When applicable `PF` is preferable for throughput optimization, while `WP` is the one giving the best latency outcome. `KP` has the ability to

Name	Type	Stream Type	Optimizes	Notes
Window Farming	Window Parallelism	Unkeyed/Keyed	Throughput	Applicable to any function $\mathcal{F}$. Possible data replication.
Key Partitioning	Window Parallelism	Keyed	Throughput	Applicable to any function $\mathcal{F}$ and any type of state. Load balancing can be problematic. No data replication.
Pane Farming	Window Parallelism	Unkeyed/Keyed	Throughput Latency	Applicable when $\delta > 1$ and $\mathcal{F}$ can be expressed as a composition of functions $\mathcal{G}$ and $\mathcal{H}$.
Windows Partitioning	Data Parallelism	Unkeyed/Keyed	Throughput Latency	Applicable when $\mathcal{F}$ can be expressed as *map* and *reduce*. No data replication.

Table 4.3: Characteristics of the proposed parallel patterns.

handle generic state (not only window) at the expense of a more difficult load balancing. It and `WF` do not require particular properties to be hold by the computation. In general these pattern represent a set of reusable solutions that cover many recurrent computations in window based DaSP operators. Therefore, in our opinion, the possibility to integrate them in existing DaSP or Skeletal frameworks should deserve special consideration from the community.

5 Adaptive parallel computations

This chapter reviews the basic idea behind Autonomic Computing, with a particular focus on adaptive parallel computations. The first part provides an overview on general concepts and related literature on the autonomic management in computing systems. Then we focus on parallel computations and how to enhance them with autonomic features, with the intent of guarantee Quality of Service requirements imposed by users. Reconfiguration mechanisms and reactive and predictive strategies will be presented and discussed. Finally we introduce adaptation strategies based on the Model Predictive Control approach that will constitute the methodology used for devising autonomic DaSP operators.

5.1 Autonomic Computing Systems

Modern applications experience continuous changes in their life cycle, due to dynamic execution scenarios, failures, variations in resources availability. These problems have to be faced by applications in a transparent way to final users. In the last years the study of these issues has laid the groundwork for a new model of computing called Autonomic Computing [Kephart and Chess, 2003].

An *Autonomic Computing System* (ACS) makes decision on its own using high level policies in order to achieve a set of goals. It constantly checks, monitors and optimizes its status and automatically adapts itself to the changing conditions. The goals to achieve can regard different properties of the system. As portrayed by IBM we can recognize at least four of them [IBM, 2005]:

- *self-configuring*: the system is able to automatically configure its components under dynamic and changing execution platforms;

- *self-optimizing*: the system monitors its behavior and controls the resources to ensure the optimal functioning with respect to the defined requirements, e.g. performance ones;
- *self-healing*: this property regards the ability to discover, diagnose and recover from fault;
- *self-protecting*: it is the ability of the system to anticipate, detect, identify and protect against threats and intrusions.

In general, an ACS can be seen as composed by interactive collections of *autonomic elements* [Kephart and Chess, 2003]. Autonomic elements will manage their internal behavior and their relationships with other autonomic elements in accordance with policies that users or programmers have established. Autonomic elements are composed by a *managed element*, a software or hardware element of the system, and an *autonomic controller* (or *manager*), an independent entity able to affect the element operational conditions in order to achieve the defined goals. Managed element and controller interact by exchanging information to enforce the various *self-** properties. This interaction can involve different phases, continuously executed by the two parties:

1. *monitoring*: the current status of the element is monitored by collecting measurements that are of significance to the *self-X* property of the system. This information can be acquired by different providers: sensors can be used to obtain environmental information, profiling services can measure execution parameters such as performance (e.g. calculation time) and resource utilization levels (e.g. memory occupancy);
2. *analyze*: the retrieved information are analyzed. The current status of the system is checked against the imposed goals;
3. *plan*: if needed, i.e. goals are not met, a reconfiguration strategy is planned according to the adaptation policy, with the goal of re-conveying the application in a legal status;
4. *execute*: the decided reconfiguration actions are applied to the controlled element.

These four phases and their periodical execution identify a closed-loop interaction scheme (*MAPE* loop) between the managed element and its controller, which is a general and well-known structure for adaptive systems (see Figure 5.1).

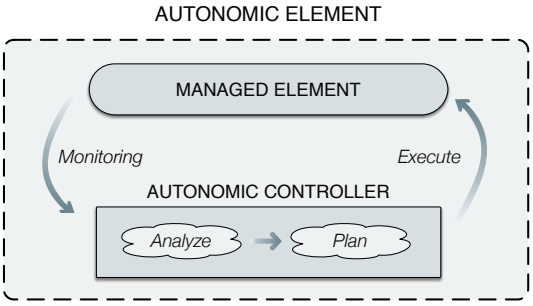


Figure 5.1: Control loop scheme of an Autonomic Element

Autonomic computing has attracted the interest of various research groups that focus on its applicability in grid [Rahman et al., 2011] and cloud [Singh and Chana, 2015] environments. In our research group it has been successfully applied to structured parallel applications. In ASSIST [Vanneschi and Veraldi, 2007; Aldinucci et al., 2006b] it was introduced the concept of autonomous parallel module that we will recall in the next section. In [Aldinucci et al., 2008], *behavioural skeletons* are introduced to obtain adaptivity for distributed high-performance computations. They were initially implemented within the reference implementation of the GCM, the CoreGRID Grid Component Model (result of the *CoreGrid* european project). Finally, in [Gabriele, 2012; Mencagli and Vanneschi, 2014] a novel control based approach to adaptation strategy has been devised. In the following we will review the key concepts behind these works and discuss why they will be a starting point for our adaptive and parallel DaSP operators.

5.2 Adaptive parallel programs

In an ACS different control-loops can be implemented to assure the different self-* properties. We are interested in parallel application having the autonomic ability to *self-optimize* their behavior. Our major concerns are about performances and resources consumption: the parallel operator must be able to monitor itself and reconfigure (e.g. increase the parallelism degree) when QoS requirements are not met. We refer to this ability as *adaptivity* (or *elasticity*).

In our vision, the parallel application (defined by the computation graph) represents the autonomic computing system. The autonomic element is the *adaptive parallel operator* (see Figure 5.2) composed by two parts:

- the *operating part*: this part is responsible for implementing the business logic

of the operator. It is a parallel computation expressed according to a certain structured parallelism scheme (e.g. task-farm, data-parallel, pane farming ...);

- the *control part*: this part represents the controller, an autonomous entity able to observe the operating part execution and respond to different cases of dynamicity.

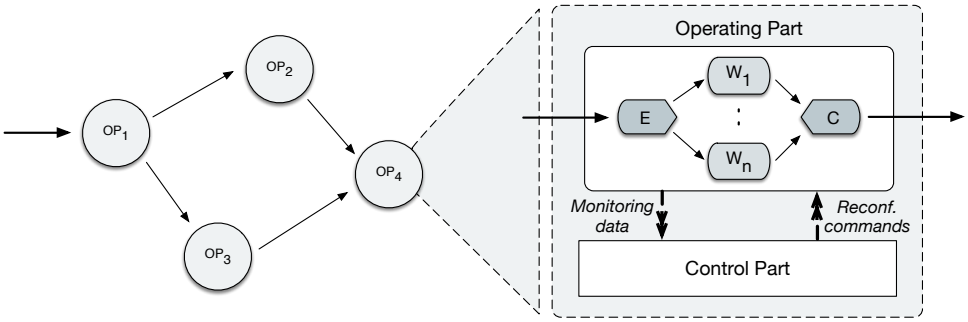


Figure 5.2: Computation graph and internal structure of an adaptive parallel operator

Between these two parts, interactions occur in the closed loop along two directions: *(i)* the managed operator observes its behavior by measuring non-functional metrics (e.g. the actual level of performance parameters) that are sent as monitoring data to the controller; *(ii)* based on the evaluation of a specific strategy, a set of actions are taken to influence the element behavior. Reconfiguration commands are issued by the controller. These trigger the execution of dynamic reconfiguration activities.

This highlights two important aspects of the adaptive operator: the presence of specific *adaptation mechanisms* able to modify the system behavior, and a *strategy* to select control actions as and when necessary. In the rest of this thesis we will focus on the definition, formalization and control of the single adaptive parallel operator.

5.3 Dynamic reconfigurations

Considering an autonomic operator, the computation (i.e. the operating part) can be executed according to different alternative *configurations*. A configuration consists in a specific selection of features such as the current parallelism degree, the current distribution policy of input elements to the workers, other architecture dependent parameters, such as mapping over available cores, CPU frequency and so on. A *reconfiguration*, driven by an adaptation strategy, involves changes in the current operator configuration.

Dynamic reconfigurations are intrusive actions on the computation (e.g. reassign state partitions over a set of workers) that may induce performance degradation and semantic inconsistencies if not properly implemented. To prevent the occurrence of these problems, reconfigurations must be executed according to specific optimized protocols in order to minimize the reconfiguration cost.

In structured parallel computations, the structure and organization of different entities (emitter, workers and collector) and their communication schema are well-defined. Therefore the exploitation of such methodology renders feasible the development of optimized reconfiguration mechanisms that can be encapsulated in the runtime supports system of the programming environment. Such mechanisms can be exploited to implement the reconfiguration activities decided by the control part without any programmer's intervention like in [Aldinucci et al., 2006b; Vanneschi and Veraldi, 2007]. In addition this removes the burden of correctness issues about reconfiguration from developers, enforcing an environment where users do not need to care about reconfiguration semantics at all.

We will focus on *non-functional* reconfigurations. They are adaptation processes involving the run time modification of some implementation aspects of a parallel operator:

- structural changes in the current parallelism degree, e.g. increase the number of workers to obtain a better performance;
- the runtime support can modify the mapping over the available resources;
- the runtime can also affect the operating behavior of the computing resources, by changing, for example, the CPU frequency for a better trade off performance/power consumption;
- data distribution changes: the way in which data is partitioned/distributed over workers can be changed. In the case of stateful computation this could involve also data redistribution among the processing entities.

The common aspects of the previous reconfigurations are that they do not modify the sequential algorithm performed by the parallel module, neither the parallelism scheme. Which kind of reconfiguration is legit and meaningful and how it should be efficiently implemented clearly depends on the particular parallelism paradigm exploited.

5.4 Adaptation strategies

Although reconfiguration mechanisms are an important part of autonomic systems, decision-making strategies are essential to achieve the system goals. Resources assigned to each component must be automatically adjusted to the changing environmental conditions. To accomplish the execution goals with the desired Quality of Service, decision-making strategies should be in charge of selecting the best reconfigurations.

Various approaches could be used in deriving control strategies. We will briefly review the reactive approaches and the predictive ones. A systematic comparison of different methodologies to develop adaptation strategies has been made in [Maggio et al., 2012]. For the following discussion we will adopt a *time-driven* controller. In time driven controllers, the adaptation strategy evaluation is performed periodically, at equally spaced time instant. We call the time interval between two subsequent decision points *control step*.

Strategies can be qualitatively and quantitatively compared using specific metrics of the adaptation process like the number of QoS violations achieved, number of used resources, frequency of reconfigurations. In the following we will take as reference the well-known SASO properties [Hellerstein et al., 2004; Gedik et al., 2014]:

- *Stability*: the strategy should not oscillate the configuration used, i.e. produce too frequent modifications of the actual configuration;
- *Accuracy*: the configuration chosen at each step should satisfy the QoS objectives, i.e. it should be able to minimize the number of QoS violations;
- *Settling time*: the strategy should be able to find a stable configuration quickly;
- *Overshoot*: the strategy should avoid overestimating the configuration needed to meet the QoS requirements under the actual workload level.

We are interested in finding the strategy achieving optimal (or a good trade off between) accuracy with few reconfigurations, short settling times and small overshoot.

5.4.1 Reactive approaches

In reactive strategies the control part decides the set of reconfigurations evaluating the current monitored data at each control step, reacting to the monitored events not matching the expected application behavior. Corrective decisions are chosen hoping

that decisions that have been taking at the current time will be effective also for future execution conditions.

A possible solution for expressing reactive strategies consists in providing a mapping between execution events and corresponding reconfigurations as a finite set of imperative *policy rules*. *Action* policies [Kephart and Walsh, 2004] are a well known-paradigm to express this kind of strategies. They dictate the actions that should be taken whenever the system is in a given current state. Typically they are expressed as:

$$\underline{if}\ (condition)\ \underline{then}\ (action)$$

the condition is checked (periodically in a time-driven controller) to ensure that the system is in a given state. If this condition holds, the corresponding action is enforced. As an example, a simple action policy could be the following one:

$$\underline{if}\ (response\_time > threshold)\ \underline{then}\ (increase\ parallelism\ degree\ by\ one)$$

It is worth noting that the state of the system that will be reached by taking the given action is not specified explicitly. Therefore the policy programmer has to known the desired effect of the selected policy. Action policies are designed for computational performance and simplicity at the potential cost of accuracy. In fact, such solutions generally cannot be proven to converge to the optimum or desired value. Action policies have been applied in [Liu and Parashar, 2006] for distributed emergency management systems. In [Aldinucci et al., 2008] were used to drive reconfigurations of *Behavioural Skeletons*, in which skeleton cost models are used to drive the reconfiguration decisions in a clever way with respect to using a heuristic solution. The approach is characterized by mechanisms for controlling also multiple non-functional concerns of a parallel computation (e.g. it is possible to simultaneously control different parameters like performance and security objectives). In this case the solution proposed in [Aldinucci et al., 2009] provides multiple autonomic managers for a single component, each one controlling a specific non-functional concern by using a set of policy rules. Different policies can lead to conflicting decisions: authors propose a distributed consensus-based solution to deal with these situations.

Although they are a very simple and flexible solutions, policy rules are not easy to be tuned: rules calibration may be hard and requires time. The use of this kind of strategies makes more difficult to prove the convergence to optimal solutions and to reach advanced properties such as the stability of control decisions and their optimality. Furthermore, when different rules make contrasting decisions, conflict resolution strategies must be taken into account to avoid letting the system oscillates between

different states. Another consideration is that in *reactive strategies* decisions are taken when a particular event is already occurring (e.g. the response time is above a threshold), typically resulting in a high number of QoS violations. It is worth recalling that in the DaSP context the majority of current approaches to the adaptive management of the application fall in this category (see Section 2.4).

5.4.2 A predictive and model driven approach

In contrast to reactive approaches, predictive strategies try to take in advance the corrective actions. The controller tries to estimate the future, thinking ahead of corrective actions such that certain undesired conditions can be prevented. As explained in [Maggio et al., 2012], standard and advanced control-based strategies may overcome the limit of reactive heuristic approaches in providing stability and optimality in the adaptation process. The applicability of such approaches to real world applications is constrained to a way to obtain future predictions of interesting monitored metrics and to the presence of a system model, in order to compare and evaluate alternative configurations from a QoS perspective.

Among this controllers family, *Model Predictive Control* (MPC) [Camacho and Bordons Alba, 2007] is a powerful strategy able to achieve good optimality and stability in uncertain environments. MPC is a design principle for controllers originally developed more than 40 years ago and widely adopted in the process industries.

First works to control computing systems using MPC have been described in [Abdelwahed et al., 2004] for controlling a server farm by dynamically varying the number of active nodes and in [Kusic and Kandasamy, 2006] to adapt the number of physical machines allocated to web servers. Authors in [Kusic et al., 2008] have used it to maximize the benefits of the resource provider by reducing energy usage and SLA outrage. On Clouds, these concepts have been studied for the dynamic allocation of virtual machines in [Yuan et al., 2011].

Although these past researches have some common points with the approach that we will use, they are heavily tailored to the target physical platform without exploiting any knowledge about the controlled computations. Instead, the idea that we pursue is to apply MPC to parallel DaSP applications by leveraging on the knowledge of the application structure. In our research group MPC has been already applied for self-optimizing structured parallel computations [Gabriele, 2012; Mencagli et al., 2013; Mencagli and Vanneschi, 2014], however considering only stateless paradigm (e.g. farm, map) and providing guarantees only on the sustainable throughput. Now we will bring the similar approach also in DaSP stateful operators with additional control

objectives in mind. In the next section the basic concepts of this technique will be introduced.

5.4.3 Model Predictive Control

The application of the MPC consists in different phases (sketched in Figure 5.3) that start from the observation of the managed element and arrive to a reconfiguration decision. In the following we describe their major characteristics, exemplifying them on the case of a generic parallel stream computation. It is worth recall that in a time-driven approach the controller evaluates the strategy at the beginning of the control step, in parallel with the system execution.

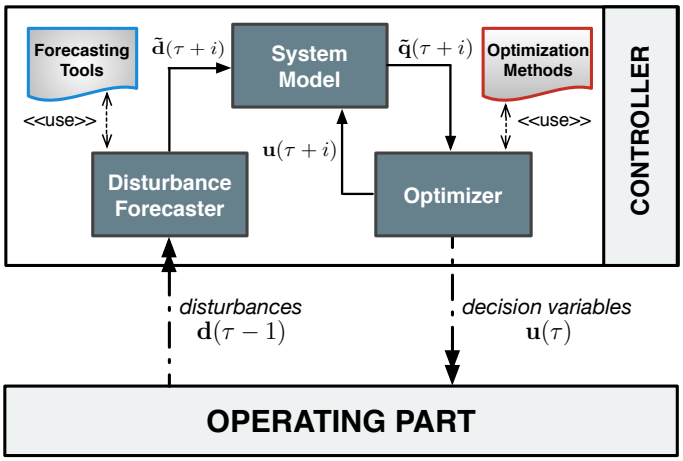


Figure 5.3: Internal structure of a MPC controller

Disturbance forecaster

The controller observes the system through the measurement $\mathbf{d}(\tau-1)$ of exogenous uncontrollable events affecting the system, the so called *disturbances*. The measurements are collected periodically, once per control step. That is at the beginning of control step τ the disturbances of the previous step are available to the controller. In stream processing applications, the arrival rate and the processing time per input tuple can be modeled as disturbances. For the dynamic use of cost models, future disturbance estimation is a crucial point. Therefore the controller predicts their future values through statistical forecasting tools that exploit the history of past samples (e.g. time-series analysis [Herbst et al., 2013]). At this point a future time horizon,

called *prediction horizon*, of h consecutive control steps is considered. A forecasting model has a general structure defined as follows:

$$\widetilde{\mathbf{d}}(\tau) = \psi\big(\{\mathbf{d}(\tau - i)\}_{i=1,\ldots,p}, \nu_1, \nu_2, \ldots\big)$$

where $\mathbf{d}$ is a disturbance variable and $\widetilde{\mathbf{d}}(\tau)$ is the predicted value for the control step τ. The model uses the last p measurements while $\nu_1, \nu_2, \ldots$ are parameters usually obtained by training the model using a representative sample of data. Multi-step ahead forecasting models return a sequence of the next $h \geq 1$ future values. In some cases, an acceptable estimation of the future value of a disturbance variable is represented by the last instantaneous monitored value, i.e. $\widetilde{\mathbf{d}}(\tau) = \mathbf{d}(\tau - 1)$.

System model

The nature of the MPC approach is *model driven*: an operating part model is used to compare alternative configurations and to evaluate them under the same disturbance scenario. The model captures the relationship between *QoS variables* (e.g. service rate, latency, energy consumption) and the current configuration expressed as a set of *decision variables*. In particular a model Φ put in relationship the following variables:

- **QoS variables** (denoted by $\mathbf{q}(\tau)$) represent information that characterize the quality of the execution in control step τ. They could regard, for example, the service rate, the computation latency and power consumption;
- **decision variables** (denoted by $\mathbf{u}(\tau)$) that identify the operating part configuration (e.g. number of workers, CPU frequency) in control step τ;
- the **disturbances variables** whose future evaluation is predicted by the forecaster.

In general, the model structure can be described by the following discrete-time equation:

$$\widetilde{\mathbf{q}}(\tau) = \Phi\big(\mathbf{u}(\tau), \widetilde{\mathbf{d}}(\tau)\big) \tag{5.1}$$

indicating that the future value of QoS variables at step τ depends on the particular operator configuration and expected disturbances variables.

Deriving system models can be a non trivial task, but the use of structured parallel paradigms can play a fundamental role. Each parallelism pattern is composed of a limited set of functionalities (collectors, distributors, workers) with a precise behavior and well known interactions. As already mentioned, the knowledge of these interactions schema is of great help in defining analytical or approximated cost models of interesting QoS metrics.

Optimizer

The MPC controller solves an optimization problem to obtain the optimal *reconfiguration trajectory* $U^h(\tau) = (\mathbf{u}(\tau), \mathbf{u}(\tau+1), \ldots, \mathbf{u}(\tau+h-1))$ over a prediction horizon of $h \geq 1$ steps. Formally, the optimization problem can be expressed as follows:

$$\min_{U^h(\tau)} \ \mathcal{J} = \sum_{i=0}^{h-1} L\big(\widetilde{\mathbf{q}}(\tau+i), \mathbf{u}(\tau+i)\big) \tag{5.2}$$

Subject to:

$$\widetilde{\mathbf{q}}(\tau+i) = \Phi\big(\mathbf{u}(\tau+i), \widetilde{\mathbf{d}}(\tau+i)\big), i = 0, \ldots, h-1$$
$$\mathbf{u}(\tau+i) \in \mathcal{U}, i = 0, \ldots, h-1$$

The optimization is constrained by the system model which uses a trajectory of disturbance forecasts $\widetilde{D}^h(\tau) = (\widetilde{\mathbf{d}}(\tau), \widetilde{\mathbf{d}}(\tau+1), \ldots, \widetilde{\mathbf{d}}(\tau+h-1))$. Furthermore, decision variables must belong to the admissible set $\mathcal{U}$ (e.g. all the feasible combinations of parallelism degrees and CPU frequencies). The function L is a *step-wise cost* that models the different control objectives, for example maintain the QoS levels on certain metrics while minimizing the used resources.

At this point a possible solution could be to apply this reconfiguration plan step by step and calculate a new optima control trajectory only at the end of prediction horizon. In practical scenarios this is not an effective approach: the uncertainty of disturbance estimations (which increases going deeper in the prediction horizon) and the potential imprecision of QoS models, suggest a more iterative approach. Only the first control decision of the optimal trajectory will be applied. When this control step ends, the entire procedure is repeated at the next control step exploiting new measurements from the system. The effect is to move the prediction horizon towards in the future following the so called *receding* or *rolling horizon* (see Figure 5.4).

The positive aspect of MPC is its intrinsic ability to incorporate this feedback mechanism [Camacho and Bordons Alba, 2007].The problem of a similar approach is related to its computational complexity. When the set of the decision variables is limited to a finite set of values, the optimization phase requires to explore the combinatorial set of all the feasible combinations of the decision variables. This, together with the fact that the optimization problem must be solved at each sampling interval, render the optimization phase is a crucial point of the whole approach even with relative short horizons ($h = 1, 2, 3$). To be implementable, MPC requires to complete the optimization process within the temporal constraints dictated by the control step of the system. Therefore, *computational efficiency* is a critical issue and will be carefully taken into account in Chapter 6.

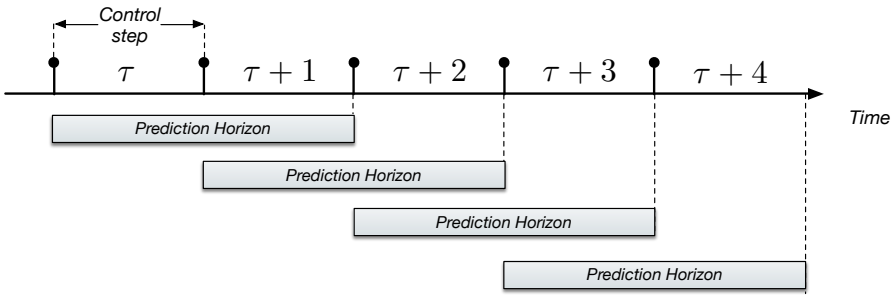


Figure 5.4: Example of the receding horizon principle with a prediction horizon length of two control steps. At the beginning of each step the strategy is re-evaluated

5.5 Summary

In this chapter we introduced the basic concepts of adaptive parallel computations. The MPC approach has been presented and it will constitute the starting point for the developing of adaptive DaSP operators having properties of high accuracy and stability and low overshoot. In the next chapter we will apply the approach to the developing of DaSP operators and we will evaluate its effectiveness.

The use of well defined parallel paradigms to exploit parallelism in computation will allow to encapsulate reconfiguration mechanisms and adaptation strategies directly into the runtime support level of the programming environment. In this way programmer is only requested to properly use high-level constructs or annotations to express structured parallel variants and identify desired QoS aspects to be attention of the adaptation strategy.

6 Strategies and mechanisms for adaptive DaSP operators

One of the characterizing aspects of Data Stream Processing applications is their long running nature (24hr/7d). Their workload and input rate may exhibit wide variations that need to be sustained in order to provide the QoS required by the users without interruptions. Adaptivity is a fundamental capability that these applications should exhibit: they must be able to autonomously scale up or down the used resources to accommodate dynamic requirements and workload by maintaining the desired QoS in a cost effective manner.

In Chapter 4 we have introduced various patterns to deal with the parallelization problems of common DaSP computations. The natural prosecution of that work is to enhance those solutions with autonomic capabilities. In this chapter we will tackle the problem of devising mechanisms and adaptation strategies that take into account performance guarantees (in terms of throughput and latency) and resources consumption of the parallel solution. We will use a predictive and control theoretic approach. This, together with the exploitation of well known parallelization schema, will allow to provide *adaptive DaSP operators* that are able to meet the performance requirements by reducing the operating costs. Everything concerning the adaptive behavior could be encapsulated inside the runtime support of a high-level programming environment, in such a way as to completely hide these aspects from the programmer's viewpoint and, clearly, to final users.

In this chapter we will study these aspects for the Key Partitioning (KP) solution (presented in Section 4.4). We concentrate on this pattern for a variety of reasons:

- Key Partitioning is a generalist pattern: provided that it is used over a keyed stream, it does not require neither particular properties on the computation that must be applied nor on the type of internal state used. For this reasons in this chapter we will not make explicit reference to the case of a windowed

operator, but rather we will take into account a generic keyed stateful operator (among which, we know, windowed ones are a notable example);

- it is a well known and diffused parallelization schema in the literature and existing SPEs. Therefore solutions and strategies here proposed could be amenable to find a rapid application also in these frameworks;
- mechanisms and strategies that will be developed are suitable also for the other parallel patterns introduced in Chapter 4. Mechanisms will deal with common reconfiguration activities such as changes in the parallelism degree and internal state movements and therefore can be at least partially reused. Strategies will rely on the Model Predictive Control and can be easily adapted also to the other patterns.

The chapter is organized as follows. In the first part we will describe the dynamicity challenges that a partitioned stateful operator parallelized with a KP pattern should face. The pursued control objectives, in terms of QoS assurances and resources usage minimization, will be detailed. Subsequently we will detail adaptation strategies that leverage the Model Predictive Control to fulfill these objectives. Therefore we have to derive cost models for the interesting objectives taking into account the structure of the exploited pattern. While in Chapter 4 we described the impact on performance parameters of the various parallel patterns in an intuitive way, now we need formal and parametric models to quantitatively measure these impacts. Finally we will focus on reconfiguration mechanisms: we abstract from the target architecture in order to provide general description of useful and re-usable reconfiguration actions. In the next chapter a detailed experimental section will follow, evaluating the proposed solutions on shared and distributed memory architectures.

The contents of this chapter and related experiments have been partially published in [II,III].

6.1 A dynamic world

Real-world stream processing applications are characterized by highly variable execution scenarios. For a keyed DaSP operator we can recognize three different dynamicity factors that may affect its working behavior:

($D1$) *variability of the stream pressure*: input streams are generated by outside sources that can be highly variables (e.g. sensors, financial tickers and social networks

data). The input rate can exhibit large up/down fluctuations and/or burstly characteristics;

($D2$) *variability of the key distribution*: the frequency distribution of the keys can be time-varying making load balancing impossible to be achieved statically;

($D3$) *variable processing time per tuple*: processing time per tuple can be different across the keys and may change significantly during the execution. In some cases like *time-based sliding window* it can be dependent from the arrival rate.

Operators should be able to tolerate these variability issues in order to keep the operator QoS optimized according to some user's criteria. The simple solution of configuring a system to sustain the *peak-load* is not effective. In many case the "peak" is unknown, therefore the only solution is to use all (or a considerable part of) the available resources. This is usually too expensive. Moreover it can be ineffective if the operator is not able to balance the workload during the execution, e.g. if the key frequencies exhibit wide variations at runtime.

For these reasons elastic strategies and related mechanisms are mandatory in this kind of applications. They must change operators' configurations in a non intrusive way to face all the mentioned dynamicity challenges.

In Key Partitioning (KP) a partitioned stateful operator is parallelized by using a set of workers each one is responsible for a state partition (refer to Section 4.4). The emitter guarantees that *all* the tuples with the same partitioning key are routed to the same worker. In this way tuples within the same group are processed in the same order of appearance in the input stream. To do so, the emitter uses a *routing function* $m : \mathcal{K} \rightarrow [1, n]$, where n is the current number of workers and $\mathcal{K}$ is the set of keys.

Scaling strategies must be able to change the parallelism degree n (the current number of workers) and the distribution function m. In this thesis we will take into account also energy consumption reduction in modern CPUs with *Dynamic Voltage and Frequency Scaling* (DVFS) capabilities. Therefore the operating CPU frequency f could be another interesting decision variable.

In the following section we will detail control strategies to enhance the KP pattern with autonomic features in order to:

- guarantee a certain QoS level required by users. Most of the existing strategies in DaSP are throughput oriented (see Section 2.4), but we want to provide assurances also on the experienced latency. Achieving and guarantee low latency is a crucial factor in many DaSP domains (e.g. financial trading, surveillance systems) as late results are practically worthless;

- minimize the resources consumption. We will not focus only on a simplistic approach "use the minimum number of processing elements". In our opinion it is interesting to tackle the problem also from an energy awareness point of view. Reducing energy consumption is a big challenge in today's society and in DaSP applications deserves special attention due to their long running nature.

In the reminder of this chapter we will rely on the following assumptions:

- our target architectures will be a multicore CPUs (single node) or a cluster of multicore machines. The various entities of a parallel operator (i.e. emitter, workers and so on) will be in execution on distinct cores of the machine(s);

- *homogeneity*: the workers are executed on the homogeneous cores of a (set of) multicore machine(s). For multicore CPUs this implies that all the cores run at the same frequency. In the case of execution over a cluster we are assuming that all the machines involved have the same hardware and computational characteristics.

6.2 Adaptation strategies

Following the description given in Section 5.2, the Key Partitioning schema is now enhanced with a controller in charge of taking the reconfiguration decisions (see Figure 6.1).

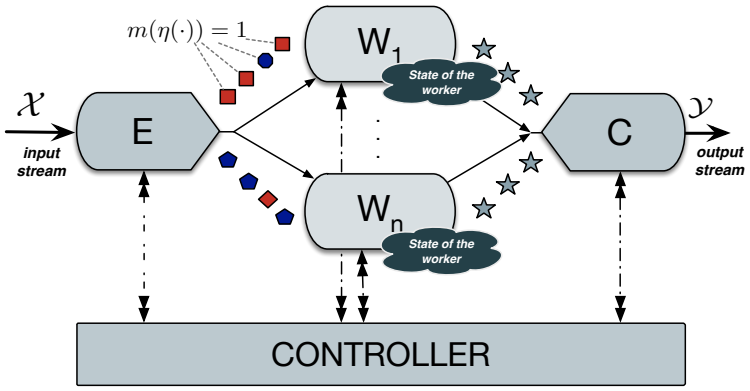


Figure 6.1: Adaptive KP operator. In the figure m is the routing function used by the emitter; η represent the key attribute of a tuple. Solid lines represent data flows. Dashed ones represent monitoring and reconfiguration messages.

The controller observes operator's execution and periodically acquires measurement data (past values of disturbances) from the computation functionalities. Table 6.1 summarizes the set of basic measurements gathered by the controller from the emitter and the workers. These metrics are relative to a control step (omitted to simplify the notation), i.e. at the beginning of control step τ the updated measurements related to the last step $\tau - 1$ are available.

Symbol	Description
T_A, σ_A	Mean and standard deviation of the inter-arrival time per triggering tuple (of any key). The arrival rate is $\lambda = T_A^{-1}$. These measurements are collected by the emitter.
$\{p_k\}_{k \in \mathcal{K}}$	Frequency distribution of the keys. They are measured by the emitter.
$\{\mathbf{t}_k\}_{k \in \mathcal{K}}$	Arrays of computation times of the keys during the last control step. Each $\mathbf{t}_k$ is an array of values for key k, collected by the worker owning the key k during the last step.

Table 6.1: Basic monitored *disturbance* metrics collected by the emitter and the worker functionalities and gathered at the beginning of each control step by the controller.

In the following discussion we will use symbols marked with a tilde on top to denoted forecasted values of disturbances or of derived metrics. Unmarked symbols refer to measured metrics.

Like in Section 4.9, the term *triggering tuple* denotes a tuple that triggers operator's internal processing logic. For stateless or stateful operators that are activated by each incoming data a triggering tuple is any input tuple. For window based stateful operators is any tuple that triggers a new window activation (according to the window triggering policy, see Section 2.1.2). Non triggering tuples are simply inserted into the corresponding window and it is reasonable to assume that they have a negligible computation cost.

The workers monitor the computation time per triggering tuple $\{\mathbf{t}_k\}_{k \in \mathcal{K}}$. To reduce the amount of measurements, it could be the case that each worker performs a sampling of the reported times.

6.2.1 Derived metrics

Starting from the reported basic disturbances, the controller can derive other additional metrics (reported in Table 6.2).

Symbol	Description
T_k	Mean computation time per triggering tuple with key k.
T_W^i	Mean computation time per triggering tuple (of any key) processed by worker i.
T	Mean computation time per triggering tuple (of any key) processed by any worker.
T_A^i	Mean inter-arrival time (of any key) to worker i.
T_S^{id}, σ_S	Mean and standard deviation of the ideal service time of the operator.
ρ_i	Utilization factor of worker i.
ρ	Utilization factor of the operator.
W_Q	Operator's mean waiting time per triggering tuple (of any key).
c_a, c_s	Coefficients of variation for operator's inter-arrival and ideal service time.

Table 6.2: Metrics derived by the controller from the basic ones.

From the arrays of measurement $\mathbf{t}_k$, for each $k \in \mathcal{K}$ the controller derives the average processing time per key that we denote with T_k. The mean computation time per triggering tuple of any key of the i-th worker is calculated as follows:

$$\widetilde{T}_W^i(\tau) = \frac{1}{\sum\limits_{k|m^\tau(k)=i} \widetilde{p}_k(\tau)} \sum_{k|m^\tau(k)=i} \widetilde{p}_k(\tau)\widetilde{T}_k(\tau) \tag{6.1}$$

It is essentially the weighted mean of the computation times of the keys assigned to worker i. In the definition of this quantity we used the values of the key frequencies p_k and computation time T_k for the next step. Since their current values cannot be measured until the next control step, they need to be forecasted by using predictive filters. In some cases it could be reasonable to use the last measured values as the next predicted ones, e.g. $\widetilde{p}_k(\tau) = p_k(\tau - 1)$.

More generally, the mean computation time per triggering tuple of the entire operator (for any key and any worker) can be defined as:

$$\widetilde{T}(\tau) = \sum_{k \in \mathcal{K}} \widetilde{p}_k(\tau)\widetilde{T}_k(\tau) \tag{6.2}$$

The mean inter-arrival time of triggering tuples to worker i is given by:

$$\widetilde{T}_A^i(\tau) = \frac{\widetilde{T}_A(\tau)}{\sum_{k|m^\tau(k)=i} \widetilde{p}_k(\tau)} \tag{6.3}$$

This follows from the observation that a worker i receives only a fraction of the input tuples transmitted by the emitter functionality, i.e. the ones whose key attribute value is assigned to that worker by the routing function. In the above formula the inter-arrival time value for the current step $\widetilde{T}_A(\tau)$ should be predicted using forecasting tools in order to track the future workload.
The *utilization factor* of worker i is derived as follows:

$$\widetilde{\rho}_i(\tau) = \frac{\widetilde{T}_W^i(\tau)}{\widetilde{T}_A^i(\tau)} \tag{6.4}$$

If it is greater or equal than one, the worker is a bottleneck. If no worker is a bottleneck, the emitter is able to route tuples to the workers without blocking on average. Otherwise, if at least one worker is a bottleneck, its input queue grows up to reaching its maximum capacity. At this point the backpressure throttles the emitter which is periodically blocked from sending new tuples to the worker. Therefore this is a situation that should be avoided on the long term.

6.2.2 Performance and energy models

The MPC relies on operator models to evaluate different operator configurations (Section 5.4.3). For the moment being we assume that the operator configuration for a given step τ is uniquely identified by the number of workers, the routing function used by the emitter and the operating CPU frequency (Table 6.3). The models return the predicted values of QoS variables. As pointed out, we are interested in the *throughput* sustained by the operator, its *latency* (or more formally the *response time*) and the *power consumption*. Table 6.4 reports the QoS variables which are the output of the models.

Symbol	Description
$n(\tau)$	Number of workers used during step τ.
m^τ:$\mathcal{K} \rightarrow [1, n(\tau)]$	Routing function used by the emitter during control step τ.
$f(\tau)$	The operating CPU frequency (GHz) used by the operator during control step τ.

Table 6.3: *Decision variables* selected by the controller

Symbol	Description
$T_S(\tau)$	Effective service time of the operator during step τ. It is the inverse of the throughput.
$\mathcal{R}_Q(\tau)$	Expected response time (*latency*) of the operator during step τ.
$\mathcal{P}(\tau)$	Power consumed by the operator during step τ (in Watts).

Table 6.4: *QoS variables* output of the models.

Throughput model

We define the *ideal service rate* of the operator as the average number of triggering tuples that the operator is able to serve per time unit provided that there are always new tuples to ingest. We are interested in the inverse quantity, i.e. the ideal service time T_S^{id}. To derive it we have to account for the slowest worker, i.e. the one with the highest utilization factor:

$$\widetilde{T}_S^{id}(\tau) = \widetilde{T}_W^b(\tau) \cdot \sum_{k|m^\tau(k)=b} \widetilde{p}_k(\tau) \qquad (6.5)$$

$$\textit{such that: } b \in \underset{i\in[1,\ldots,n(\tau)]}{\arg\max}\ \widetilde{\rho}_i(\tau)$$

The *effective service time* is given by the maximum between the ideal one and the inter-arrival time:

$$\widetilde{T}_S(\tau) = \max\left\{\widetilde{T}_A(\tau),\ \widetilde{T}_S^{id}(\tau)\right\} \qquad (6.6)$$

This essentially shows that to optimize the throughput it is not necessary to balance the workload between the workers, but it is sufficient that all the workers have utilization factor less than one. Under the assumption that the load is evenly distributed among the $n(\tau)$ workers, the ideal service time formula of Equation 6.5 can be further simplified as:

$$\widetilde{T}_S^{id}(\tau) = \frac{\sum_{k\in\mathcal{K}} \widetilde{p}_k(\tau)\widetilde{T}_k(\tau)}{n(\tau)} = \frac{\widetilde{T}(\tau)}{n(\tau)} \qquad (6.7)$$

Equation 6.7 requires that, given the current frequency distribution, there exists a routing function m^τ that allows the workload to be (quasi) evenly balanced among the workers. As stated in recent literature [Gedik et al., 2014], this assumption practically holds in many real-world applications, where skewed distributions are common but a well balanced distribution function can usually be found. This will clearly require

to properly balance the workload among the n workers by changing the distribution function when necessary.

Latency model

A very important QoS metric is the *latency* experienced by the users. More formally we are interested in the *response time*, i.e. the time elapsed from the reception of a tuple that triggers the operator internal processing logic and the production of the corresponding output.

Analogously to [Lohrmann et al., 2015] we use a Queueing Theory approach to derive a formal cost model for this important metric. The mean response time of the operator during a control step τ can be modeled as the sum of two quantities:

$$\mathcal{R}_Q(\tau) = W_Q(\tau) + T(\tau) \tag{6.8}$$

where W_Q is the *mean waiting time* that a triggering tuple spends from its arrival to the system to when the operator starts the execution related to it.

To find the mean waiting time per triggering tuple, our idea is to model the operator as a $G/G/1$ queueing system, i.e. a single server system with inter-arrival time and service times having general statistical distributions. An approximation of the mean waiting time for this system is given by the Kingman's formula [Kingman, 1962]:

$$\widetilde{W}_Q^K(\tau) \approx \left(\frac{\widetilde{\rho}(\tau)}{1-\widetilde{\rho}(\tau)}\right)\left(\frac{\tilde{c}_a^2(\tau)+\tilde{c}_s^2(\tau)}{2}\right)\widetilde{T}_S^{id}(\tau) \tag{6.9}$$

where the input parameters are the following:

- the utilization factor of the operator during step τ, defined as $\widetilde{\rho}(\tau) = \widetilde{T}_S^{id}(\tau)/\widetilde{T}_A(\tau)$;
- the coefficient of variation of the inter-arrival time $c_a = \sigma_A/T_A$;
- the coefficient of variation of the ideal service time $c_s = \sigma_S/T_S^{id}$.

Equation 6.9 can be used for stable queues only, i.e. such that $\widetilde{\rho}(\tau) < 1$. This condition is used in most of the Queueing Theory results, and implies that this model can be used for evaluating the response time for configurations in which the operator is not a bottleneck. The ideal service time can be determined according to Equation. 6.5 or Equation. 6.7 if the load is balanced among the workers.

We choose this model for its generality, since it does not need strict assumptions on the type of the arrival and service stochastic processes. Simpler formulas of the waiting time for other queueing systems like $M/M/1$, $M/D/1$ and $M/G/1$

exist and can be used in our strategies by assuming that the transmission rate of the stream source can be modeled as a Poisson process. However, in real situations arrival and service distributions are usually unknown and such strong assumptions usually do not hold. The case of Equation 6.9 is more general but also more challenging because several information (e.g. coefficients of variation) must be efficiently monitored by the runtime to apply it. We will use Equation 6.9 by making the following simplifications:

- we model the whole operator as a single queueing system with service time equal to the one of the operator with n workers (Equation 6.5 or Equation 6.7);
- although the mean inter-arrival time for the next step $\widetilde{T}_{\mathcal{A}}(\tau)$ is forecasted using statistical filters, the estimated coefficient of variation is kept equal to the last measured one, i.e. $\widetilde{c}_a(\tau) = c_a(\tau - 1)$;
- the coefficient of variation of the ideal service time is equal to $\widetilde{c}_s(\tau) = c_s(\tau-1)$. So doing, we suppose that c_s is unaffected by changes in the parallelism degree.

Kingman's model provides good accuracy especially for systems with utilization factor near to one (close to saturation) [Kingman, 1962; Gross et al., 2008], which is a good property since our goal is to avoid wasting resources. We will use this model to compare the latency of different operator configurations. To increase its precision we use a feedback mechanism in order to fit Kingman's approximation to the last measurements gathered by the controller. This mechanism is defined as follows:

$$\widetilde{W}_Q(\tau) = e(\tau) \cdot \widetilde{W}_Q^K(\tau) = \frac{W_Q(\tau - 1)}{\widetilde{W}_Q^K(\tau - 1)} \cdot \widetilde{W}_Q^K(\tau) \tag{6.10}$$

The parameter e is a corrective factor defined as the ratio between the measured mean waiting time during the past step $\tau - 1$ collected by the emitter functionality and the last prediction obtained by Kingman's formula. The idea is to adjust the next prediction according to the past error. A similar mechanism has been already applied with good results to the problem of estimating the response time of a chain of operators in [Lohrmann et al., 2015].

Power consumption model

Power efficient computing systems have drawn the attention in the last years, due to both environmental and economical reasons. The energy is defined as the power consumed on a given time interval. Owing the infinite nature of DaSP computations,

the minimization of the instant power (*power capping*) is the main solution to reduce energy consumption and cutting down operating costs. Therefore a desired control objective could be to find the operator configuration that minimizes the power consumption on the execution architecture. We will focus on the power consumption of the CPU, since it is in charge of the greatest percentage of power consumption when the system is under load [Feng et al., 2005]. In general, there are various factors that contribute to the CPU power consumption. We can recognize at least two of them, the static and the dynamic power parts [Kim et al., 2003]:

$$\widetilde{\mathcal{P}}(\tau) \approx \widetilde{\mathcal{P}}_{static}(\tau) + \widetilde{\mathcal{P}}_{dynamic}(\tau)$$

The *static power* part represents the power consumed from leakage mechanisms in the transistors. It depends from the number of active transistors and other technology dependent features. The *dynamic part* is essentially due the activity of the logic gates of the CPU and it is originated when the processor is active in executing computations. In this thesis we will use a basic power model, in which we do not consider the static power dissipation of the CPU. The reasons of this choice are twofold. First of all, for the moment being we are not interested in knowing the exact amount of energy consumed, but only a proportional estimation such that we can compare different operator's configurations and choose the most power efficient one that satisfies the QoS required. Secondly, the static power depends also on technological characteristics inherent to the CPU used, while the dynamic power is proportional to factors that we may actually control (e.g. number of active cores and frequency; see Equation 6.11). Clearly we recognize that a more precise power model could allow better results in terms of power-saving, but we left its evaluation for future works. Given this assumption, we model the power dissipation of the CPU as its dynamic part. This follows the underlying formula [Intel, 2004; Miyoshi et al., 2002]:

$$\widetilde{\mathcal{P}}(\tau) \sim C_{eff} \cdot n(\tau) \cdot f(\tau) \cdot \mathcal{V}^2 \tag{6.11}$$

where the power during step τ is proportional to the used number of cores (that is equal to the number of workers in our case), the CPU frequency and the square of the supply voltage $\mathcal{V}$, which in turn depends on the frequency of the processor. In the model C_{eff} represents the *effective capacitance* [Chandrakasan and Brodersen, 1995], a technological constant that depends on the hardware characteristics of the CPU.

From Equation 6.11 we deduce that the power consumption of the CPU can be reduced by reducing its frequency and voltage, by shutting down unused resources or reducing the number of cores used by the application. Modern CPUs have different frequency operating points (the so called *P-States*). The voltage required for stable

operation is determined by the frequency at which the circuit is clocked, and can be reduced if the frequency is also reduced. Therefore by lowering the operating frequency, the supply voltage will be lowered too, thus allowing a reduction of both components of Equation 6.11. Furthermore, the second component of the equation decreases more than linearly with respect to the frequency, since a relationship exists between these two quantities. This mechanism of frequency throttling is commonly known as *Dynamic Voltage and Frequency Scaling* (DVFS).

The rationale that we will adopt to bind the used frequency, power consumption and computation time is the following one: halving the frequency we will double the computation time (this is true for CPU bound computations [Miyoshi et al., 2002]), but on the other hand we will use less than half the power (thanks to the more than linear relation between frequency and voltage). Using Equation 6.11 we are able to estimate the power consumed by different operator configurations, expressed by all the feasible combinations of the number of used cores and the employed CPU frequency, and select the most power efficient one that meets the targeted QoS level. It is important to note that we are switching the CPU frequency for the whole system. In modern processors, it is possible to do this at a finer granularity, for example for each CPU socket (like in the architecture used in our experiments) or even for each different core in some cases. We do not consider explicitly this possibility: on the one hand this could allow to save more power, but on the other hand it is in contrast with our *core homogeneity* assumption (see Section 6.1). In particular this will complicate our performance models, optimization problem and load balancing heuristics (the latter aspect will be discussed in Section 6.4.3).

Finally, it is worth noting that by monitoring the system utilization factor $\rho(\tau)$ only, we are not able to achieve this goal. If we have a given utilization factor to guarantee (derived for example from Equation 6.9) several configurations can achieve the same or similar utilization factor, e.g. by using 5 workers with a frequency of 2GHz or 10 workers at 1 GHz.

6.3 Optimization phase

In Chapter 5 we have seen that at each step an MPC based controller resolves an optimization problem. The nature of our optimization problem is combinatorial, with the decision variables taking their values from discrete and finite sets. A simplification in the resolution of this problem consists in taking out the routing function m from the decision variables (Table 6.3). The idea is to assume that at each control

step the controller is always able to find a routing function that allows the workload to be (quasi) evenly balanced among the workers. In this way the controller can use Equation 6.7 to estimate operator's ideal service time instead of the more general Equation 6.5. This assumption is realistic in all the scenarios in which the *skewness* factor (ratio between the most frequent and the least frequent key) can be acceptably bounded to some relatively small constant. This conditions holds in many real-world applications and most of the recent research works assume this conditions to be true [Gedik et al., 2014; Gedik, 2014; Nasir et al., 2015]. Therefore the MPC controller is designed as in Figure 6.2 with a *rebalancer* component in charge of computing a new routing function that balances the workload among the workers. This compo-

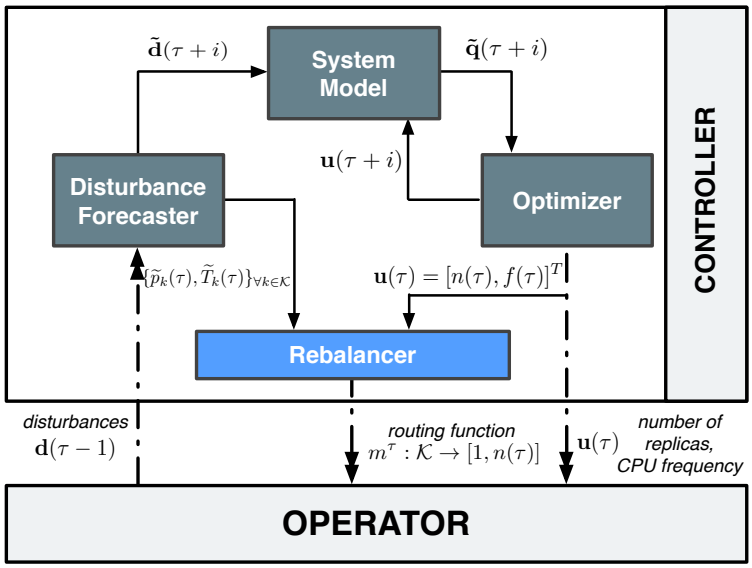


Figure 6.2: Internal logic structure of the MPC controller with the *rebalancer* component for generating the routing function

nent is executed at the beginning of each control steps, after that a new operator configuration has been chosen, for two reasons:

- if the MPC controller changes the number of workers, the rebalancer computes a new routing function to map the keys onto the new set of workers;
- even if the number of workers does not change, the rebalancer computes a new routing function if the difference in terms of load between the most loaded worker and the least loaded one is over a threshold. This allows the MPC controller to use Equation 6.7 to accurately estimate operator's ideal service time.

We will see in Section 6.4 different rebalancing policies. From this follow that the decision variables of the models are now the number of workers and the CPU frequency. They are represented by the a vector $\mathbf{u}(\tau) = [n(\tau), f(\tau)]^T$.

6.3.1 The optimization problem

The MPC controller solves at each step the optimization problem defined in Equation 5.2:

$$\min_{U^h(\tau)} \mathcal{J} = \sum_{i=0}^{h-1} L\big(\widetilde{\mathbf{q}}(\tau+i), \mathbf{u}(\tau+i)\big)$$

The cost function is the sum of the step-wise costs L over a horizon of $h \geq 1$ future steps. A general form of the step-wise cost can be expressed as follows [Abdelwahed et al., 2009], with $i = 0, \ldots, h-1$:

$$\begin{aligned} L(\widetilde{\mathbf{q}}, \mathbf{u}, i) = Q_{cost}\big(\widetilde{\mathbf{q}}(\tau+i)\big) + & \qquad \textit{QoS cost} \\ + R_{cost}\big(\mathbf{u}(\tau+i)\big) + & \qquad \textit{Resource cost} \\ + S^w_{cost}\big(\Delta_u(\tau+i)\big) & \qquad \textit{Switching cost} \end{aligned} \tag{6.12}$$

The cost comprises three terms. The *QoS cost* represents the user degree of satisfaction with the actual QoS (i.e. throughput and latency). The *resource cost* models a penalty proportional to the amount of resources/power consumed. The two terms counterbalance each other: while a higher usage of resources guarantees better performance in general (hence a lower QoS cost), this also results in a higher resource cost.

The *switching cost* is a function of the vector $\Delta_u(\tau) = \mathbf{u}(\tau) - \mathbf{u}(\tau-1)$. It models the penalty incurred in changing the actual configuration and it will be used by our strategies to achieve a proper compromise between SASO properties (see Section 5.4). The role of the switching cost is also more subtle: it binds control decisions between consecutive steps. Consequently, the optimal reconfiguration trajectory cannot be determined by minimizing L at each step, but the controller needs to explore the space of all the possible reconfiguration trajectories $U^h(\tau)$.

Various MPC-based strategies can be designed by using different formulations of the three cost terms. We now introduce different variants of them.

QoS cost

With the intent of guarantee two different QoS requirements, we distinguish between two QoS cost definitions:

- *throughput-based cost*: the goal is to make the operator able to sustain the input rate. A possible way of modeling this requirement is to use the following formulation:

$$Q_{cost}\big(\widetilde{\mathbf{q}}(\tau+i)\big) = \alpha\,\widetilde{T}_S(\tau+i) \tag{6.13}$$

  That is a linear cost proportional to the effective service time, where $\alpha > 0$ is a unitary price per time instant. To minimize the QoS cost the controller chooses one of the configurations with minimum service time i.e. equal to the predicted inter-arrival time according to Equation 6.6. It is worth noting that the QoS cost for all the configurations that lead to an ideal service time lower than the arrival time is the same;

- *latency-based cost*: in latency-sensitive applications the response time needs to be bounded to some user defined thresholds. Exceeding those requirements may lead to a loss of revenue or to wrong results depending on the type of system controlled. We model this QoS requirement with a cost function defined as follows:

$$Q_{cost}\big(\widetilde{\mathbf{q}}(\tau+i)\big) = \alpha\,\exp\left(\frac{\widetilde{R}_{\mathcal{Q}}(\tau+i)}{\delta}\right) \tag{6.14}$$

  where $\alpha > 0$ is a positive cost factor. The cost lies in the interval $(\alpha, e\alpha]$ for latency values within the interval $(0, \delta]$, where $\delta > 0$ is a maximum threshold for the response time. The idea is that such kind of cost heavily penalizes configurations with a response time greater than the threshold as the cost increases exponentially.

Resource cost

The resource cost is defined as a cost proportional to the number of used workers or to the overall power consumed, which in turn depends both on the number of used cores and the CPU frequency. In the two cases we use the following cost definitions:

$$R_{cost}\big(\mathbf{u}(\tau+i)\big) = \beta\,n(\tau+i) \qquad \textit{per-core cost} \tag{6.15}$$

$$= \beta\,\widetilde{\mathcal{P}}(\tau+i) \qquad \textit{power cost} \tag{6.16}$$

where $\beta > 0$ is a unitary price per unit of resources used (per-core cost) or per watt (power cost).

Switching cost

We use the following switching cost definition in the MPC optimization problem:

$$S^{w}_{cost}\big(\Delta_u(\tau+i)\big) = \gamma\left(\|\Delta_u(\tau+i)\|_2\right)^2 \tag{6.17}$$

where $\gamma > 0$ is a unitary price factor. The cost includes the euclidean norm of the difference vector between the decision vectors used at two consecutive control steps. Quadratic switching cost functions are common in the control theory literature [Camacho and Bordons Alba, 2007], as they penalize frequent reconfigurations by avoiding the controller oscillating the configuration used. With the euclidean norm both dimensions (i.e. number of workers and operating CPU frequency) are considered equally significant. Our first intent is to show that a similar switching cost definition has two important effects:

- it incentives the controller to change the configuration as less as possible;
- owing to the quadratic non-linearity, it favors small modifications of the actual configuration.

Both the aspects will be discussed in Chapter 7 and play a crucial role in designing control strategies able to reach desirable SASO trade-offs. Other switching cost could be easily defined, for example to give more weight to (and therefore avoid) changes in the number of workers with respect to changes in the CPU frequency. They will not be discussed in the experimental chapter and left to future investigations.

6.3.2 Search space reduction

The MPC optimization phase might need to explore the combinatorial set of all the feasible combinations of the decision variables. Being $\mathcal{N}$ the maximum number of available workers and $\mathcal{F}$ the set of feasible CPU frequencies, the number of explored configurations is $\mathcal{O}((\mathcal{N} \times |\mathcal{F}|)^h)$. Thus we have exponential increase in worst-case complexity with an increasing number of reconfiguration options and longer prediction horizons. The computational overhead of the controller could become a major concern, as the strategy needs to occupy a small (negligible) fraction of the control step interval. Therefore, methods to reduce the computational overhead are mandatory for real-time execution.

The optimization is essentially a search problem over a tree structure called *evolution tree* [Mencagli and Vanneschi, 2014], whose height corresponds to the horizon

length h and the arity to the total number of configuration options $\Omega = \mathcal{N} \times |\mathcal{F}|$, as shown in Figure 6.3. An approach to reduce the explored search space consists in

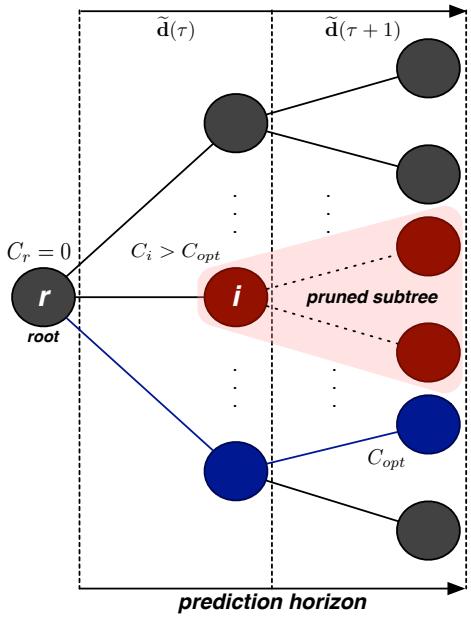


Figure 6.3: Evolution tree ($h = 2$) and B&B procedure to reduce the search space of the MPC optimization process.

Branch & Bound methods (B&B). We will use the following procedure [Mencagli and Vanneschi, 2014]:

- we assign to each explored node i of the tree a variable C_i which represents the cost spent to reach that node from the root. The cost of the root is zero;
- for each node we have Ω possible branches. The subtree rooted at node i is explored if and only if $C_i < C_{opt}$, where C_{opt} is the minimum cost of all the root-to-leaf paths currently explored during the search process. If i is a leaf, we further set $C_{opt} = C_i$.

This procedure can be applied if the cost function $\mathcal{J}$ is monotonically increasing with the step of the horizon. According to Equation 6.12, $L > 0$ for each step, thus the cost function $\mathcal{J}$ satisfies this property.

The procedure does not affect the final choice, which is the same of the exhaustive search. However, the space reduction depends on the size of the pruned subtrees and we have no guarantee of its efficacy in general.

In conclusion, B&B techniques can mitigate the problem but not solve it at all. For very large configuration spaces the controller overhead could be unacceptable even using B&B. Possible solutions can be evolutional algorithms, AI methods or special heuristics [Abdelwahed et al., 2009]. The employment of such techniques will be evaluated in the future.

6.4 Reconfiguration mechanisms

The evaluation of the adaptation strategies results in a (possible) new configuration of the operator, in terms of number of workers, CPU frequency and routing function (computed by the rebalancer). Reconfiguration mechanisms enforce the new determined configuration. Various mechanisms are necessary and in the rest of this section we will describe them trying to abstract from the target architecture. In Chapter 7 we will further details their implementations depending if they are in execution on a shared memory architecture or on a distributed environment.

In every case we assumed that the execution platform is composed by one (or more) multicore CPUs. Therefore emitter, collector and the worker are executed by threads running on the cores of the underlying architecture. The controller is executed by a dedicated control thread. This choice is in line with other approaches proposed in the recent literature. In [Gedik et al., 2014] the proposed elastic support consists in a control algorithm performed by a centralized entity (at the emitter in this case). In our implementation we prefer to keep separated the strategy evaluation by the routing functionality in order to avoid the emitter being slowed. A similar solution has been adopted in [Lohrmann et al., 2015]. Solutions with distributed interconnected controllers [Scattolini, 2009] will be studied in the future.

Threads cooperate by exchanging messages along queues that interconnect them and are implemented accordingly to the architecture. In the same machine they are implemented using shared lock-free buffers offered by Fastflow. In a shared nothing environment we resort to traditional sockets (further details in Chapter 7). Apart from the tuples coming from the input stream and the produced results, emitter, workers and collector send measurements to the controller while the controller will send reconfiguration messages. Now we will describe in an architecture-agnostic way the basic reconfiguration mechanisms needed to support the dynamic adaptation of a DaSP operator parallelized using the KP pattern.

6.4.1 Increase/Decrease the number of workers

If it is required to increase the number of workers, we must create a set of new threads for the additional workers and the related queues used to interconnect them with the emitter, collector and the controller itself must be created. The controller sends special control messages to the emitter and the collector in order to notify them of the new configuration. The controller also computes a new routing function which is notified to the emitter functionality. Then, the emitter starts the migration protocol described in the next section.
Symmetric actions are taken in the case of a removal of a subset of the workers.

6.4.2 State migration

Critical for the correctness of the computation is the state migration actions needed at each change in the number of workers. Each time the rebalancer computes a new routing function, some of the data structures must be migrated. This requires to move data from the worker that was handling a given key k, to the worker that is now responsible of that key.

We are interested in low-latency streaming applications for which the state migration protocol must be very optimized. We identify four fundamental properties of a reconfiguration protocol, which represent general design principles for a low-latency reconfiguration protocol for adaptive operators

(R1) *gracefulness*: during the migration the emitter and the workers should avoid discarding input tuples, processing tuples with the same key out-of-order, and should prevent the generation of duplicated results (i.e. preserve *exactly-once* semantic);

(R2) *fluidness*: the emitter should never wait for the migration completion before starting to distribute new incoming tuples to the workers again;

(R3) *non-intrusiveness*: the migration should involve only the workers exchanging parts of their state. The workers not involved in the migration should be able to process the input tuples without interferences;

(R4) *fluentness*: the workers involved in the migration should not be blocked until the migration is complete. While a worker involved in the migration is waiting to acquire the state of an incoming key, it should be able to process all the input tuples with other keys for which the state is ready to be used.

We firstly provide a brief summary of the most recent solutions to this problem according to the mentioned properties. Then, we will show our approach.

Qualitative comparison

Table 6.5 shows a summary of some of the most recent elastic supports presented in the literature. The approach in [Gedik et al., 2014] does reconfigurations without generating duplicate results and processes all the input tuples without altering their arrival order, i.e. property *(R1)*. However, the migration activity executes coarse-grained synchronization barriers among the emitter and all the workers. Therefore, this support is neither able to achieve *(R2)*, i.e. the emitter is blocked waiting for the migration completion, nor properties *(R3)* and *(R4)* because all the workers are always blocked during the whole reconfiguration. Analogous is the case of the approach described in [Heinze et al., 2014a], where their operator movement mechanism has the same drawbacks.

Work	R1	R2	R3	R4	Platform
[Gedik et al., 2014]	Yes	No	No	No	Clusters
[Heinze et al., 2014a]	Yes	No	No	No	Clusters
[Wu and Tan, 2015]	No	Yes	Yes	Yes	Clusters/Clouds
[Castro Fernandez et al., 2013]	No	No	Yes	Yes	Clusters/Clouds
[Gulisano et al., 2012]	No	Yes	Yes	Yes	Clusters/Clouds
[Shah et al., 2003]	Yes	No	Yes	Yes	Clusters
Our	Yes	Yes	Yes	Yes	Multicore/Clusters

Table 6.5: Recent existing migration protocols compared with our solution.

Improvements have been developed in Chronostream [Wu and Tan, 2015] by providing a state migration protocol for latency-sensitive applications. The migration is less intrusive, as it involves only a subset of the workers that need to exchange state partitions *(R3)*. Furthermore, workers that wait for an incoming state acquisition can still process tuples for which the state is available and consistent *(R4)*, and the emitter is not required to block until the migration has finished *(R2)*. However, the mechanism introduces further complexity because the workers can generate duplicate results for the same tuple *(R1)* that must be properly filtered by the merger. Similar is the approach described in [Castro Fernandez et al., 2013], in which the emitter (or the upstream operator) is additionally involved in the re-generation of the input

tuples (of the migrated keys only) not processed during the reconfiguration (*(R2)* is not achieved). Those tuples will be processed in order to keep the state partition up-to-date. This re-generation action by the emitter is much more intrusive (it needs proper checkpointing techniques) than the transmission of our asynchronous control messages that do not block the input flow.

Two solutions are proposed in Streamcloud [Gulisano et al., 2012]. In the first one no state is transferred among workers, but new tuples corresponding to moving keys are multicasted to both the workers (the old and new ones) for a certain time interval until the window slides enough and can be processed by the new worker only. This approach can be used only for state partitions maintained as sliding windows and introduces a long reconfiguration delay, proportional to the window length, which is unacceptable for our online strategy (MPC needs that each reconfiguration completes within one control step). The second solution is based on state transfer among workers without blocking them and the emitter during the migration (*(R2)*, *(R3)* and *(R4)*). Although conceptually similar to our approach, this runtime needs sophisticated interactions between the emitter and the workers involved in the migration. All the tuples of a moved key received within the time interval of the migration activity, must be forwarded to both the workers that properly execute or discard them. Therefore, property *(R1)* is not satisfied.

Finally, the technique described in [Shah et al., 2003] is the most similar to our approach. Besides the achievement of properties *(R3)* and *(R4)* common to many other solutions, this approach gives a central role to the emitter, which is in charge of buffering all the tuples of the moved keys in a buffer. Such tuples will be delivered to the workers as soon as the migration is complete but quiesces the routing of new fresh input tuples (not involved in the migration) to the other workers in the meanwhile. Therefore, although no duplicated tuple/result is generated in this solution *(R1)*, property *(R2)* is not met.

In the next section we will describe our reconfiguration protocol able to achieve all the identified properties. Our approach never blocks the workers and the emitter entities, it does not generate duplicate results, and all the tuples are transmitted just one time to the corresponding worker that executes the computation on a given tuple exactly one time. As we will see, this approach is particularly effective in reducing latency spikes during the migration.

Fluent protocol

Our fluent migration achieves the four properties identified before and it is completely *lockless*. After a reconfiguration decision the controller transmits a *reconfiguration message* to the emitter containing a new routing function m^τ. The emitter receives data non-deterministically from the input stream (new tuples) and from the controller (reconfiguration messages), the latter with higher priority. Once received a reconfiguration message, the emitter recognizes the keys that must be migrated and transmits to the involved workers a sequence of *migration messages*:

- `move_out(k)` is sent to the worker w_i that held the data structures corresponding to the tuples with key $k \in \mathcal{K}$ before the reconfiguration but will not hold them after the reconfiguration, i.e. $m^{\tau-1}(k) = i \wedge m^\tau(k) \neq i$;

- `move_in(k)` is sent to the worker w_j that will hold the data structures associated with key k after the reconfiguration (and did not own them before), i.e. $m^{\tau-1}(k) \neq j \wedge m^\tau(k) = j$.

All the keys $k \in \mathcal{K}$ s.t. $m^{\tau-1}(k) = i \wedge m^\tau(k) = i$ are not involved in the migration and will be processed by the corresponding workers without interferences. It should be noted that such migration messages introduce a small delay in the transmission of new incoming tuples by the emitter. However, they are *asynchronous* notifications to a subset of the workers, and in any case the emitter does not need to wait the completion of the state migration activities in the worker *(R2)*.

The emitter routes input tuples using the new routing function m^τ. Let us suppose that, as a result of a reconfiguration chosen by the controller ①, a key $k \in \mathcal{K}$ must be migrated from worker w_i to w_j as depicted in Figure 6.4. At the reception of a `move_out(k)` message ②, the worker w_i knows that it will not receive tuples with key k anymore. In fact, the worker receives the `move_out(k)` message from the same queue e-w_i used for retrieving tuples distributed by the emitter (managed in a FIFO fashion). Therefore, w_i can safely save the state of that key (denoted by s_k) to a *backing store* (see Figure 6.4) used to collect the migrated state and to synchronize the pairs of workers involved in the migration.

The worker w_j, which receives the `move_in(k)` message ③, may receive new incoming tuples for that key before the state is acquired. Only when the worker w_i has properly saved the state s_k to the repository ④, it can be acquired by w_j ⑤. We devise two possible behaviors:

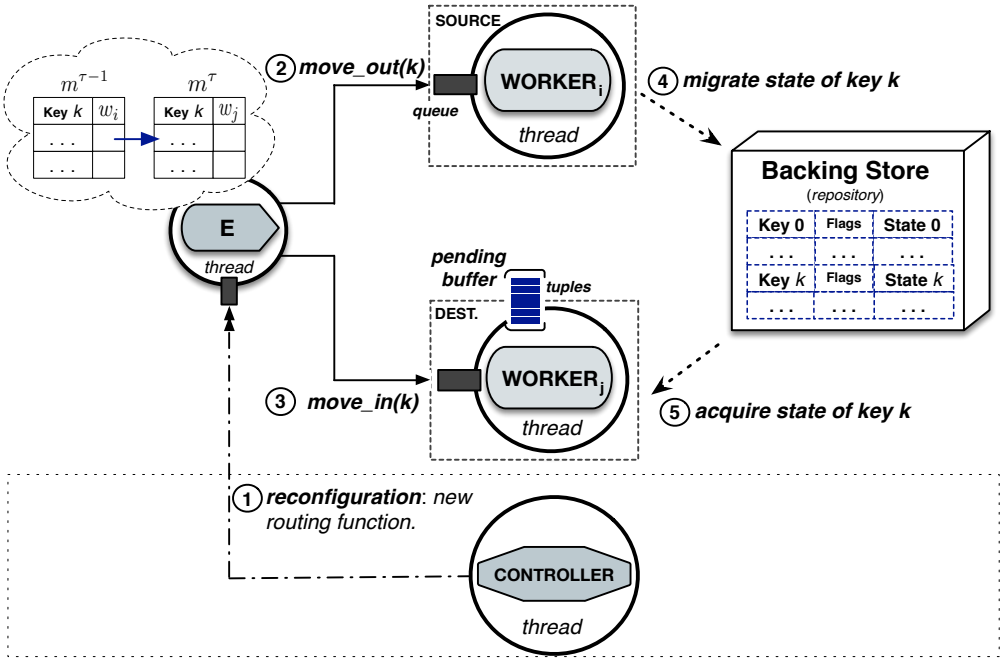


Figure 6.4: Example of state migration between worker w_i and worker w_j for key k.

- as soon as the worker w_j receives the first tuple with key k, it blocks until the state s_k is available in the backing store and can be safely acquired. This behavior does not provide property *(R4)*;
- w_j is not blocked but accepts new tuples. All the tuples with key k are enqueued in a temporary *pending buffer* until the state s_k is available. The availability of s_k in the backing store is periodically checked at each reception of a new tuple. In the meantime all the tuples with a different key can be processed. When the state becomes available, it is acquired by the worker and all the pending tuples of key k in the buffer are rolled out and processed in the same order in which they were sent by the emitter.

In both the cases the state migration for each moving key is performed in parallel by the workers without global barriers (*R3*) and without duplicates or out-of-order results (*R1*). The second solution does not stall the computation of input tuples with keys not involved in the migration (*R4*). Furthermore, pending buffers are maintained in the worker involved in the migration and not at the emitter level as in [Shah et al., 2003], therefore the emitter is able to route tuples to the workers without interruptions (*R2*). When the protocol is complete the workers and the emitter notify the controller of the end of the reconfiguration.

The backing store can be implemented in different ways accordingly to the execution architecture, possibly relying on external services. A possible alternative to the use of a backing store could require communications between workers. In our working scenario, this necessitates worker-to-worker channels and the cost of their management could sensibly increase with the increasing of used resources. We have therefore chosen to avoid a similar implementative choice.

6.4.3 Heuristics for load balancing

As highlighted in 4.4, the KP parallel pattern may suffer of load unbalancing, a problem that must be addressed especially in dynamic execution scenarios. In addition, to use Equation 6.7 it is required that at each control step the rebalancer is able to find a routing function that balance the load assigned to the workers.

The problem of finding the optimal routing function in terms of load balancing is a *NP-hard* problem equivalent to the *minimum makespan* [Vazirani, 2001]. Approximate solutions must be used, like the one in [Vazirani, 2001, Chapt. 10], see Algorithm1. In the following we will refer to this heuristic as *Balanced*, since it provide an almost optimal routing function.
Let $wt_k(\tau)$ be the "weight" of the key $k \in \mathcal{K}$ defined as:

$$wt_k(\tau) = \widetilde{p}_k(\tau) \times \widetilde{T}_k(\tau)$$

The keys are ordered by their relative weight. Starting from the key with the highest weight, each key is assigned to the worker with the actual least amount of load. We denote by $\mathcal{L}_i(\tau) = \sum_{k|m^\tau(k)=i} wt_k(\tau)$ the load of the i-th worker.

Algorithm 1 Computing the routing table.

Input: list of keys and their weight, number of workers N.
Output: a new routing table.

```
1: function RoutingTable(list of keys, N)
2:     Order list of keys by weight
3:     for i = 1 to N do
4:         L_i = 0
5:     for each k ∈ K in the list do
6:         Assign k to worker j s.t. L_j(τ) = min_{i=1}^{N} L_i(τ)
7:         Update load of worker j, i.e. L_j(τ) = L_j + wt_k(τ)
       return computed routing table
```

In the algorithm the function is represented by a live lookup table of entries $\langle key, worker\_id \rangle$. The *Balanced* solution computes ex-novo a new routing table,

not considering the current one and the number of moved keys. An alternative solution could be the one proposed in [Shah et al., 2003] (that we will call *Flux*) which tries to equalize the load of each of the workers as mush as possible while minimizing key movements. Starting from the most loaded worker (the *donor*) they pair it with the least utilized one (the *receiver)*. Then keys are exchanged between them if the difference between their loads is higher than an *imbalance threshold*. The algorithm walks down the list of keys assigned to the donor, ordered by weight in descending order, and exchange the first one that will reduce the utilization between the donor and receiver. The workers load is recomputed accordingly to the weight of exchanged keys . The procedure is re-iterated until the imbalance between donor and receiver is over the threshold. Low thresholds will result in quasi balanced load distribution.

The impact of these two different solutions will be evaluated in Chapter 7. In general they are fast and easy to understand solutions, but requires to store explicitly all the possible entries. Alternatively, a hash function can be used. Typically, uniform hash functions can be designed in order to keep the load of the workers as similar as possible. However, they do not work well if the number of workers changes at runtime. More advanced solutions, such as the one used in [Gedik et al., 2014] are based on *consistent hashing* [Karger et al., 1997], a technique to design hash functions able to balance the load with minimal number of migrated keys. Consistent hashing schemes have been extended in [Gedik, 2014] in order to balance the memory required. These alternative solutions will be considered in the future.

6.5 Summary

In this chapter we have introduced strategies and mechanisms for adaptive DaSP operators. Even if the focus is on the KP pattern, it should be noticed that the proposed solutions can be easily adapted to the other presented patterns. In addition, this pattern is the only one that manages generic state and not only windows. In the next chapter we will complete this description evaluating the effectiveness of the proposed approach on a realistic application targeting shared and distributed architectures.

7 Adaptation strategies and mechanisms evaluations

In this chapter we evaluate the reconfiguration mechanisms and control strategies developed in Chapter 6 on a DaSP operator operating in the *High Frequency Trading* (HFT) domain. HFT applications are good candidates for the evaluation, since they have usually stringent QoS requirements in terms of throughput and latency bounds as well as a very dynamic execution scenario.

The chapter is organized as follows. In the first section we will describe the application used to evaluate our solutions. Then the first part of the discussion will focus on the evaluation over a shared memory (multicore CPU) architecture. The reconfiguration mechanisms will be specialized for these environmentss and their impacts on performance carefully evaluated. Then we provide a comprehensive analysis of the proposed strategies and a comparison with similar approaches. The MPC-based strategies will be evaluated accordingly to the SASO properties described in Section 5.4. To recall, they are *Stability (P1)*, *Accuracy (P2)*, *Settling time (P3)* and *Overshoot (P4)*. Then the second part of the chapter is the extension of the proposed approach to shared-noting machines (a cluster of multicore machine). The experiments performed are similar to the ones of the shared memory platform. The intent is to show the effectiveness of the mechanisms and MPC based strategies also in this execution scenario.

7.1 The application

The evaluation of the proposed control strategies is performed on a kernel of a data stream processing application operating in the High Frequency Trading (HFT) domain. HFT computations ingest huge volume of data at a great velocity and process the market feeds with stringent QoS requirements to discover fresh trading oppor-

tunities. In this section we present a kernel of an application inspired by the work in [Andrade et al., 2011]. The kernel is sketched in Figure 7.1.

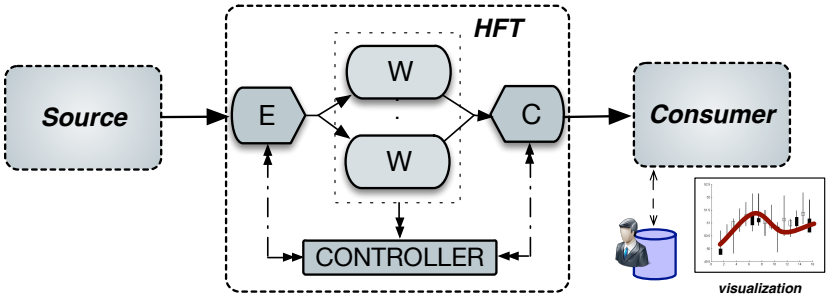


Figure 7.1: Kernel of a high-frequency trading application used to discover trading opportunities in real-time.

The *source* operator represents a stock market that generates a stream of financial quotes, i.e. buy and sell proposals (bid and ask) represented by a record of attributes like the proposed price, volume and the stock symbol (64 bytes in total). The HFT operator processes bids and asks grouped by the stock symbol. A count-based window of size $|\mathcal{W}|$ and slide δ is maintained for each group. The operator applies a prediction model aimed at estimating the future volume and price for the quotes of each symbol using the historical data. After receiving δ new tuples of the same stock symbol, the computation goes over the tuples buffered in the actual window and processes them. The processing logic consists of two main phases:

- *aggregation*: the quotes with a timestamp within the same *resolution interval* (1 ms) are transformed into a single tuple by averaging the values of the attributes of the original quotes;

- *regression*: the quotes (one per resolution interval) are used as input of the Levenberg-Marquardt regression algorithm that produces a polynomial fitting the aggregated quotes. For the regression algorithm we use the implementation offered by the `C++` library `lmfit` [Wuttke, 2015].

The results are transmitted to a *consumer* operator that produces a graphical representation in the form of candlestick charts.

The parameters $|\mathcal{W}|$ and δ can be tuned to provide different levels of accuracy and will be changed accordingly to the considered execution architecture. The values used in the experiments are typical examples. We recall from Section 6.1 the common dynamicity factors that these kinds of computations should face: we have variability in

the arrival rate *(D1)*, variability in the keys frequency distribution *(D2)* and variability in the processing time per key *(D3)*. This application presents all these variability issues. The source generates inputs with a variable arrival rate and frequency distribution of stock symbols (*D1* and *D2*). Although the windows are count-based, the aggregation phase changes the number of quotes to use for the regression. Therefore, also the computation cost per window is variable *(D3)*.

7.2 Experiments on shared memory architecture

For the evaluation of the mechanisms and control strategies on shared memory architecture, the implementation uses the Fastflow framework basic mechanisms [1]. As already indicated in Section 6.4 the various entities (emitter, workers, collector, controller and source) are implemented as separated threads that, in this case, exchange messages through lock-free shared queues of the Fastflow framework.

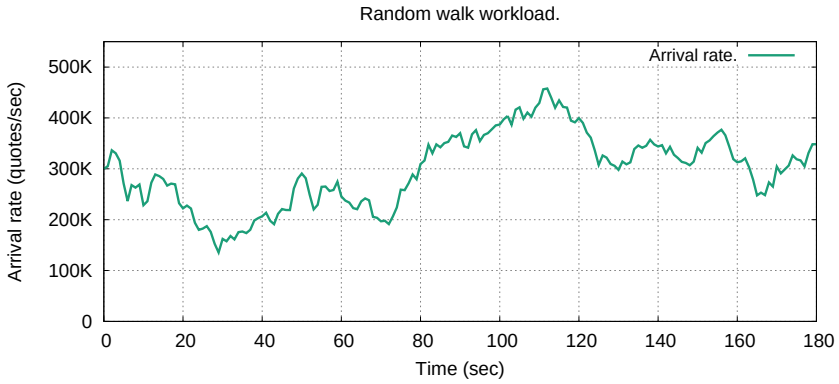
The emitter of the HFT operator is interfaced with the source by means of a TCP/IP socket. The source process is executed on the same machine. The consumer functionality is executed by the collector thread and it is in charge of saving the computation results (in a human readable format) into a file.

The target architecture is the same one of Section 4.9. The CPUs support DVFS with a frequency ranging from 1.2 GHz to 2 GHz in steps of 0.1 GHz (that results in 9 possible frequency configurations). In addition, the processor can exploit the TurboBoost feature. This architectural mechanism, presents in modern Intel CPU, dynamically overclocks the CPU frequency when higher performance is required. The amount of increased clock rate depends from the CPU current status (e.g. current dissipated power and temperature) and the number of active cores. This makes very difficult to estimate its effects on both performance and power dissipation. For these reasons, in the experiments this feature has been turned off.

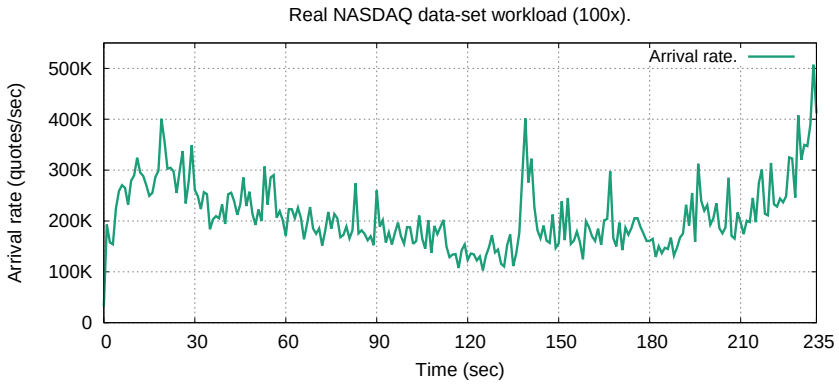
Each thread of the implementation has been pinned on a distinct physical core and hyperthreading CPU facility is not exploited. Thus the maximum number of workers is 12. The used compiler is `gcc` (version `4.8.1`), programs compiled with the `-O3` compiler flag. The used Fastflow version is the `2.0.5`.

[1] Parts of the experiments here reported are contained in the article presented at the ACM Sigplan PPoPP 2016 conference [II] that has passed the Artifact Evaluation. The paper does not contain the throughput based strategy and other minor experiments and it threats only the case of shared memory architecture. The code used for the experiments and the related documentation are available at: `https://github.com/tizianodem/elastic-hft`

For the evaluation on the target architecture, the operator uses count based windows of size $|\mathcal{W}| = 1000$ tuples and slide $\delta = 25$ tuples. This means that on average we have a *triggering tuple* each 25 incoming tuples. The source uses two different datasets: we will refer to a *synthetic workload* and a *real data-set* scenario. In the latter the quotes and their timestamps are the ones of a trading day of the NASDAQ market (daily TaQ of 30 Oct 2014[2]) with $2,836$ traded stock symbols. The peak rate observed is near to $60,000$ quotes per second, with an average rate well below this figure. Recent estimates for market data rates [Andrade et al., 2011] are near to 1 million of transactions per seconds (especially if we consider options data and not stock quotes). To model this future scenario, we accelerate 100 times the original timestamps to reproduce throttled input rates.



(a) Random walk.



(b) Real data-set.

Figure 7.2: Arrival rate: synthetic (random walk) and a real throttled ($100\times$) data-set of a NASDAQ trading day.

[2]The used dataset is freely available at the website: http://www.nyxdata.com.

In the synthetic workload (Figure 7.2a) the arrival rate follows a random walk model. The key frequency distribution is fixed and equal to a random time-instant of the real NASDAQ data-set. The skew factor, i.e. the ratio between the most probable key and the less probable one, is equal to 9.5×10^4. In this scenario we take into account the variabilities *(D1)* and *(D3)* only. Figure 7.2b shows the real workload. In this case also the key frequency distribution changes over the execution *(D2)*. The execution of the synthetic workload consists in 180 seconds while the one of the real data-set of 235 seconds, equal to about 6 hours and half in the original non-throttled data-set.

7.2.1 Reconfiguration mechanisms over a shared memory architecture

Being executed on a shared memory architecture, the messages exchanged by the various entities (e.g. data inside the operator, monitoring messages sent to controller, reconfiguration messages from the controller, ...) are memory pointers to shared data in order to avoid the overhead of extra copies. The reconfiguration mechanisms described in Section 6.4 are implemented taking into account this execution scenario.

Changes in the number of workers: when a new configuration implies the creation of workers, the controller is in charge of instantiating the set of threads for the additional workers and the Fastflow queues to interconnect them with the emitter, collector and controller itself. The controller sends special control messages to the emitter and the collector in order to notify them of the new workers and to pass a reference to the queues needed to interconnect with them. If some workers have to be removed, the controller notifies the number of workers to eliminate to the emitter. The emitter, after that the state migration messages are sent, notifies the termination to the interested workers. The controller is in charge of *joining* the respective terminating threads.

State migration: in this case the backing stores consists in a shared memory area in which workers exchange memory references to the data structures exchanged (windows in this case). This avoids the copy overhead and the point-to-point synchronizations between workers and controller, required to notify the end of the reconfiguration. Pending buffer, in which tuples are accumulated if the correspondent state is not available, are implemented as standard `C++` vectors (`std:vector<tuple_t>`).

Frequency scaling and energy measurements: modifications of the CPU frequency are performed using the `C++ MAMMUT` library[3] (MAchine Micro Management UTilities) targeting off-the-shelf multi-core CPUs. The API allows the controller to change the operating frequency by writing on the `sysfs` files. The controller uses the library also to collect energy statistics. On Intel Sandy-Bridge CPUs this is performed by reading the Running Average Power Limit (RAPL) energy sensors [Hähnel et al., 2012]. On the same way, voltage values are read through model-specific registers (MSR). The library requires administrative privileges to be used. It should be noticed that a change in the operating frequency does not affect the structure of the parallel implementation and can be performed by the controller transparently to the operator execution.

7.2.2 Mechanisms evaluations

In this first set of experiments we analyze different aspects of the proposed mechanisms in order to prove their effectiveness and low performance impacts. Each aspect is studied by means of ad hoc experimentations and it is presented in a separated section.

Overhead of the elastic support

The first aspect that deserves attention is the performance impairment due to the use of the elastic support i.e. the presence of the controller and monitoring activities by the emitter, workers and collector.

We measure the maximum input rate that the HFT operator sustains with the highest CPU frequency. The input tuples are generated with a constant rate while the $2,836$ keys are uniformly distributed. Figure 7.3 compares the elastic implementation, with the controller functionality and all the monitoring activities performed with a sampling of 1 second, and the implementation without the elastic support.

In this setting no reconfiguration is taken by the controller. We observe a modest overhead bounded by $3 - 4\%$ on average. The speedup with 12 workers is 12.91 and 12.56 for the unelastic and the elastic case. The slight hyper scalability is due to the increasing temporal locality of window data structures in the private cache of the cores by using more workers. This first experiment demonstrates that the monitoring activity and the asynchronous interaction with the controller have a negligible effect on the computation performance.

[3]The Mammut library is open source and freely available at `https://github.com/DanieleDeSensi/Mammut`.

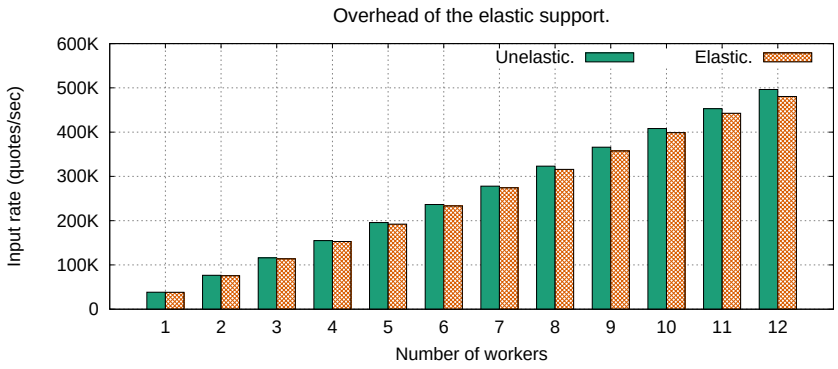


Figure 7.3: Maximum sustainable input rate: comparison between the elastic and the unelastic implementations.

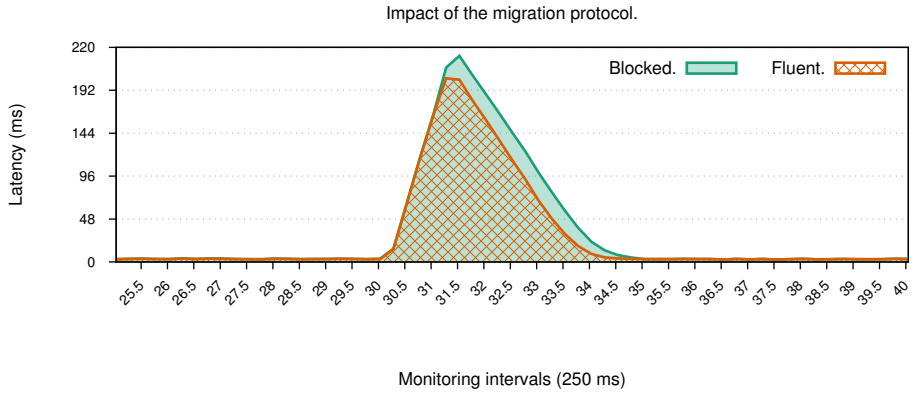
Analysis of migration protocols

This experiment is devoted to comparing different migration protocols according to the properties detailed in 6.4. In the analysis we consider two protocols:

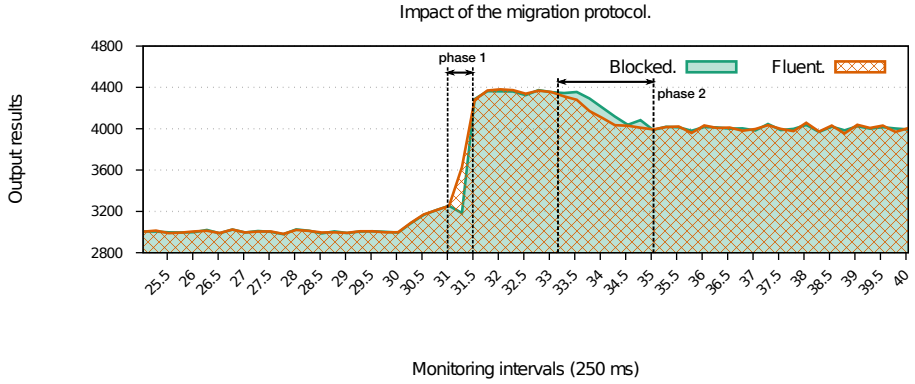
- our *fluent protocol* satisfying all the properties *(R1)*, *(R2)*, *(R3)* and *(R4)*;
- a *blocked protocol* which satisfies properties *(R1)*, *(R2)* and *(R3)* but not *(R4)*. Accordingly, all the workers involved in the reconfiguration are blocked until the reconfiguration is completed. In that case no tuples are stored in the pending buffers.

Other more intrusive protocols that block the emitter [Gedik et al., 2014] or all the workers produce worse results in terms of latency peaks and are not considered in this experiment. We analyze a scenario in which the operator is not a bottleneck and the workload is perfectly balanced until timestamp 30. Then, we force the input rate to abruptly change from $300K$ tuples/sec to $400K$ tuples/sec. The strategy detects the new rate and triggers a reconfiguration at timestamp 31 by changing the number of workers from 6 to 8 with the same CPU frequency. Figure 7.4a reports average latency measured using a monitoring interval of 250 ms (1/4 of the control step equal to one second). As depicted, during the transient phase (from timestamp 30 to 35) we observe some latency peaks. In the fluent protocol the operator is still able to process tuples not belonging to the moving keys, however additional tasks are performed such as the enqueuing of some tuples in the pending buffers, and the unrolling of them when the corresponding state becomes available. In contrast, the blocked protocol blocks the involved workers as soon as the first tuple of any migrated key is received.

This second solution is the worst one, producing higher latency peaks. Furthermore, the fluent protocol is able to reach the steady state earlier.



(a) Latency per monitoring interval.



(b) Number of results per monitoring interval.

Figure 7.4: Impact of the migration protocol: latency and number of results of the HFT operator in the time instants before, during and after a reconfiguration. Results are mediated over multiple runs.

Figure 7.4b reports the output results produced. During phase one the fluent protocol is able to produce more results. The reason is that it never blocks the workers involved in the migration, which are able to process tuples not belonging to the migrated keys. In contrast, the second protocol blocks the involved workers until the migration is complete. During this blocking phase, several tuples (of any key) accumulate in the input socket between the source operator and the HFT. These tuples are rolled out and processed after the migration; this happens during phase two in which the blocked protocol apparently produces more results. However, these results are in part the ones already computed by the fluent protocol and thus have a higher latency as confirmed in Figure 7.4a.

Impact of the migration overheads

The third set of experiments studies the impact of the migration cost. Although on multicores we can avoid transferring the whole state by-value, we have all the synchronization issues needed to preserve correctness and result ordering. We study: *i*) the *Balanced* policy shown in Algorithm 1, where a new routing function balancing at best the workload is used at each reconfiguration; *ii)* the *Flux* policy discussed in [Shah et al., 2003] and in Section 6.4 for minimizing the number of migrated keys by keeping the imbalance under a threshold (we use 10%).

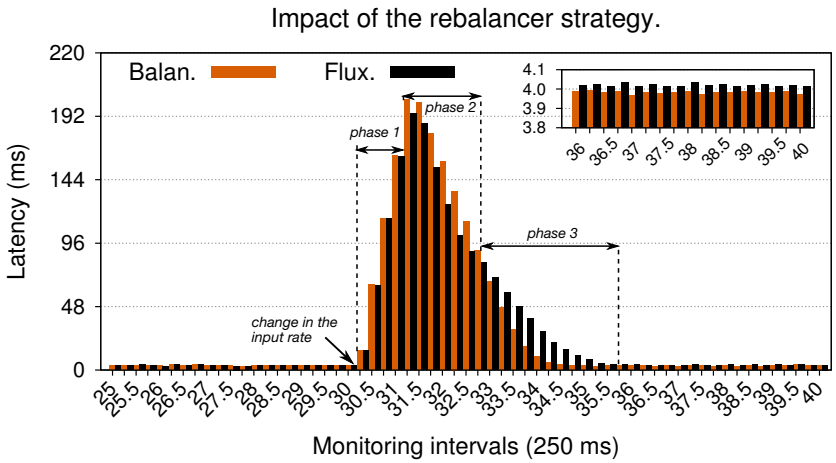
The testing scenario is the same one of the previous evaluation, with a change in the input rate at timestamp 30 that triggers a change in the number of workers at timestamp 31. From timestamp 31 to 36 we are in a transient phase where the keys are moved among the workers, tuples enqueued in the operator's input buffer are processed and we reach the new steady-state behavior.

Figure 7.5a shows the average latency measured using a monitoring interval of 250 ms (1/4 of the control step). Three phases can be identified:

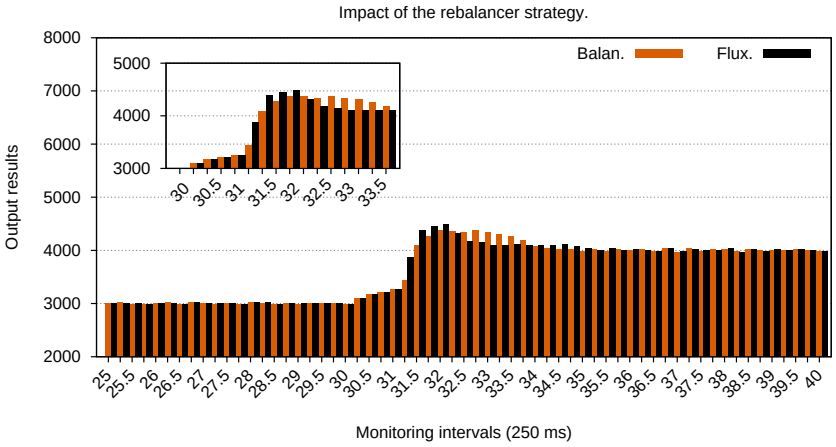
- a *first phase* in which the rate changes and the controller detects it at the next control step. During this phase both the policies produce the same latency results. The latency grows as the rate increases and the operator becomes a bottleneck;
- a *second phase* in which the controller triggers the reconfiguration and some keys are migrated (2,460 and 68 in the Balanced and the Flux policies). The Flux policy produces lower latency because fewer keys are migrated and the pending buffers are mostly empty. It usually takes a hundred of milliseconds to migrate the state in the Balanced case, while it requires tens of milliseconds with the Flux policy;
- in the *third phase* (after timestamp 33) the workers process the tuples that were waiting. Due to a better load balancing, the Balanced policy approaches the steady state faster than Flux. During this phase the measured latency with the Flux policy is about the double of the latency with the Balanced policy.

Figure. 7.5b shows the number of results produced (one per triggering tuple). The subplot shows a zoom of the central part of figure. The Flux policy outperforms the Balanced one only during the transient migration phase, while the Balanced one approaches the steady state faster owing to better service time.

In conclusion, there is no apparently winner. The slight advantage of the Flux policy is only noticeable during the migration phase, which takes a small fraction of



(a) Latency per monitoring interval.



(b) Number of results per monitoring interval.

Figure 7.5: Impact of the rebalancer strategies: latency and number of results of the HFT operator in the time instants before, during and after a reconfiguration.

the reconfiguration time. Furthermore, the steady-state behavior after timestamp 36 shows a slight advantage ($1 \div 2\%$) of the Balanced policy (see subplots in Figures 7.5a and 7.5b). We can conclude that on multicores the Balanced policy should be preferred in latency-sensitive applications since the latency peaks during the migration are not so high and the new steady state can be reached faster.

Controller complexity reduction

The MPC controller explores a potentially huge number of states which grow exponentially with the horizon length. This can rapidly make the computational burden

excessive for real-time adaptation. Table 7.1 shows the theoretical number of states that the control strategy should explore with an exhaustive research and the ones explored with the B&B solution described in Section 6.3.2. We recall that we are taking into account as decision variables the number of possible workers and the CPU frequency steps (respectively 12 and 9 in the target architecture).

	States	**Explored**	**%**	**Time**
$h = 1$	108	108	100	45.77 usec
$h = 2$	11,664	2,537	21.75	608 usec
$h = 3$	1,259,712	72,055	5.72	17 msec

Table 7.1: Explored states with respect to the total number.

The execution of the MPC procedure should cover a small fraction of the control step. According to the results of the table, the B&B solution is capable of reducing the number of explored states in our experiments. The reduction is of several orders of magnitudes and makes it possible to complete the resolution of the optimization problem in few milliseconds even with the longest horizon ($h = 3$) that we use in these experiments.

If needed, other solutions and heuristics can be studied in the future for cases with more reconfiguration options (e.g. evolutionary algorithms).

7.2.3 Adaptation strategies evaluation

At this point, we can study different MPC-based strategies according to the formulations of the optimization problem presented in Section 6.3.1. Table 7.2 shows the different choices and the name used to refer them in the experiments. Each strategy is evaluated without switching cost (that is $\gamma = 0$; they will be referred as *NoSw* strategies) or with the switching cost term and different lengths $h \geq 1$ of the prediction horizon (*Sw* strategies). Horizons longer than one step are meaningful only with the switching cost enabled. Therefore, $h = 1$ is implicit in any NoSw strategy. When we use as resource cost the per node cost (i.e. we try to minimize the number of used cores), the CPU frequency is always set at its maximum. The cost parameters α, β and γ require careful tuning in order to properly normalize the various cost components (i.e. QoS, Resource and Switching costs) and give them the desired weights. We studied the best settings of these parameters in our workload scenarios. The resource cost and the switching cost parameters are $\beta = 0.5$ and $\gamma = 0.4$ for all the

QoS cost	Resource cost	Name	alpha
throughput-based (Equation 6.13)	per node (Equation 6.15)	`Th-Node`	rw: 200 real: 200
throughput-based (Equation 6.13)	power cost (Equation 6.16)	`Th-Power`	rw: 200 real: 300
latency-based (Equation 6.14)	per node (Equation 6.15)	`Lat-Node`	rw: 2 real: 3
latency-based (Equation 6.14)	power cost (Equation 6.16)	`Lat-Power`	rw: 2 real: 4

Table 7.2: MPC-based strategies studied in the experiments. In all the cases we have $\beta = 0.5$ and $\gamma = 0.4$.

strategies, while we need a different value for the first parameter α to tune properly the weight of the QoS cost. In all the strategies we have chosen to give more priority to the QoS cost and lower priority to the resource cost and to the switching cost term. We model in this way an important case in which the controller tries to reduce the QoS violations by using minimal resources. The α values used in the experiments are shown in the table.

The derivation process of these cost parameters could be possibly automated by maintaining the same rationale. In general, the parameters values can be calculated as the ratios between a real *weight* (a dimensionless priority) and a *scale factor*. The scale factors are used to map the values in the same numerical interval/order of magnitude. To set the value of the scale factors we require knowing the range of the cost extremes (maximum and minimum values) that have been estimated as follow:

- for the resource/power cost, we know the maximum number of cores that can be used on a given architecture and the maximum power per control step (using the maximum frequency with all the cores active);
- for the switching cost it is easy to find the maximum "variation" between two configurations;
- more complex is the case of the QoS cost, which needs to determine some bounds of the QoS variables of the problem (e.g., the operator response time). These bounds could be asked to an experienced programmer or could be reasonably estimated by running preliminary experiments and monitoring the QoS results.

At this point the real weights can be chosen in order to reflect a dimensionless relative priority between the cost terms, without being aware of their order of magnitude.

In all the presented experiments, unless otherwise specified, the control step is of 1 second. In the experiments all the MPC-based strategies perform statistical predictions of measured disturbances. We assume that:

- the arrival rate will be predicted according to a Holt Winters filter [Fried and George, 2014] able to give h-step ahead forecasts by taking into account trends and cyclic non stationarities in the underlying time-series;
- the frequencies and the computation times per key will be estimated using the last measured values, i.e. $\widetilde{p}_k(\tau+i) = p_k(\tau-1)$ and $\widetilde{\mathcal{C}}_k(\tau+i) = \mathcal{C}_k(\tau-1)$ for any $k \in \mathcal{K}$ and every step $i = 0, 1, \ldots, h-1$.

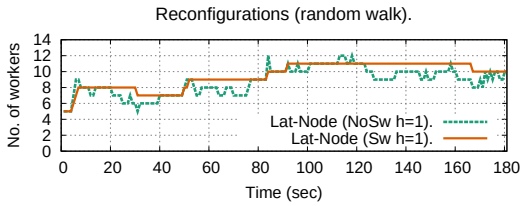
In addition we will also assume that: *i)* the rebalancer functionality adopts the Balanced policy, *ii)* we adopt the fluent migration protocol. All the experiments have been repeated 25 times by collecting the average measurements. The variance is very small: in some cases we will show the error bars to further prove the reliability of our results.

Reconfigurations and effect of the switching cost

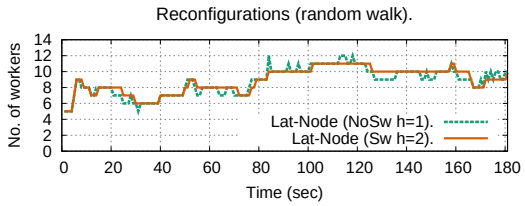
In this section we analyze the effect of the switching cost. We show the results of the `Lat-Node` strategy with the random walk and the real data-set workload. Qualitatively similar results are achieved with the other strategies of Table 7.2. Figure 7.6 shows the number of workers used by the HFT operator. The reconfiguration sequence follows the workload variability. In this setting the strategy changes only the number of workers (the CPU frequency is fixed to 2 Ghz). Phases with higher arrival rate correspond to greater number of used workers and vice-versa.

The combined effect of the switching cost and the horizon length is evident from the figure. The dashed green line corresponds to the reconfiguration sequences without switching cost ($\gamma = 0$). At each step, the MPC controller selects the number of workers that optimizes the trade-off between performance and resource cost. The orange solid line corresponds to the strategy with the switching cost which acts as a *stabilizer* of the sequence by smoothing reconfigurations. In other words, it is a brake that slows the acquisition/release of workers.

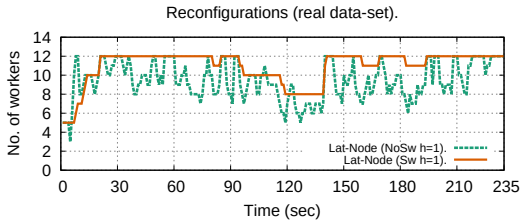
By increasing the foresight of the controller the reconfigurations with the switching cost better approximate the sequence obtained without the switching cost. The



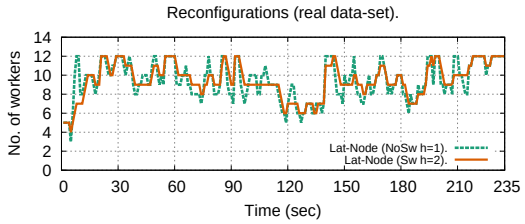
(a) Random walk `Lat-Node` NoSw ($h = 1$) vs. Sw ($h = 1$).



(b) Random walk `Lat-Node` NoSw ($h = 1$) vs. Sw ($h = 2$).



(c) Real data-set `Lat-Node` NoSw ($h = 1$) vs. Sw ($h = 1$).



(d) Real data-set `Lat-Node` NoSw ($h = 1$) vs. Sw ($h = 2$).

Figure 7.6: Number of used workers per control step (1 sec). `Lat-Node` strategy without and with the switching cost and prediction horizons $h = 1, 2$.

reason is that our multiple-step ahead forecasts are able to capture future increasing/decreasing trends in the arrival rate. During increasing trends, longer horizons allow the controller to anticipate the acquisition of new workers. The opposite characterizes decreasing trend phases. Therefore, by increasing the horizon length the effect of the stabilizer is less intensive and a faster adaptation to the workload variability can be observed in general.

Figure 7.7 summarizes the total number of reconfigurations performed by the different MPC strategies. More reconfigurations are performed in the real workload, due to a higher variability of the arrival rate from the stock market. Furthermore, more reconfigurations are performed with the strategies `Th-Power` and `Lat-Power` with respect to their counterparts `Th-Node` and `Lat-Node`. In fact, with the power-aware strategies the space of possible reconfiguration options is larger, owing to the possibility to change the CPU frequency in addition to the number of workers. In the number of reconfigurations we count any change in the number of workers and/or in the CPU frequency. We do not consider the changes (only) in the routing function taken by the rebalancer component.

Figure 7.8 shows the types of reconfigurations performed. The `Th-Power` strategy performs more changes in the number of workers instead of changes in the frequency. The opposite holds for the `Lat-Power` strategy. Reconfigurations that change both

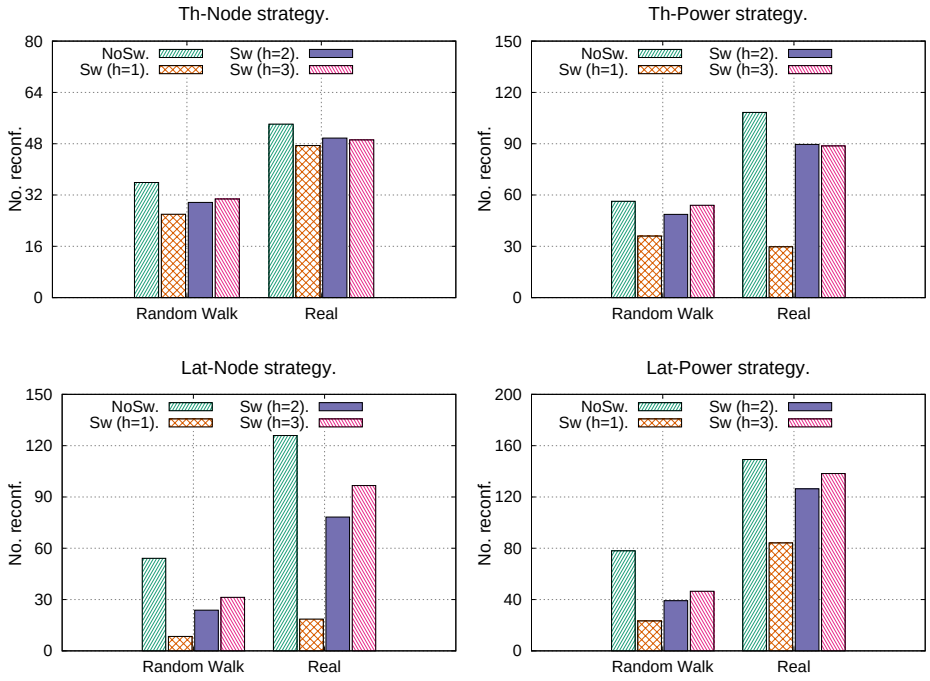


Figure 7.7: Number of reconfigurations per strategy: random walk and real workload.

the aspects are a small fraction of the total amount. In summary, we can derive the following result:

Result 1. The switching cost allows the strategy to reach better *stability* (P1), i.e. to reduce the number and frequency of reconfigurations. This effect is partially mitigated by increasing the horizon length.

QoS violations

The strategies have important effects in the accuracy *(P2)* achieved by the elastic support. A QoS violation is a deviation from the expected behavior defined as follows:

- for throughput-based strategies we measure the ratio between the number of results produced per control step N_{out} and the number of received triggering tuples N_{in}. We detect a QoS violation if the ratio computed over the horizon length is lower than a specified threshold θ. In the experiments we use a threshold $\theta = 0.95$: this implies that for each control step the operator must be able to handle at least the 95% of the incoming tuples, i.e. it is not a bottleneck;

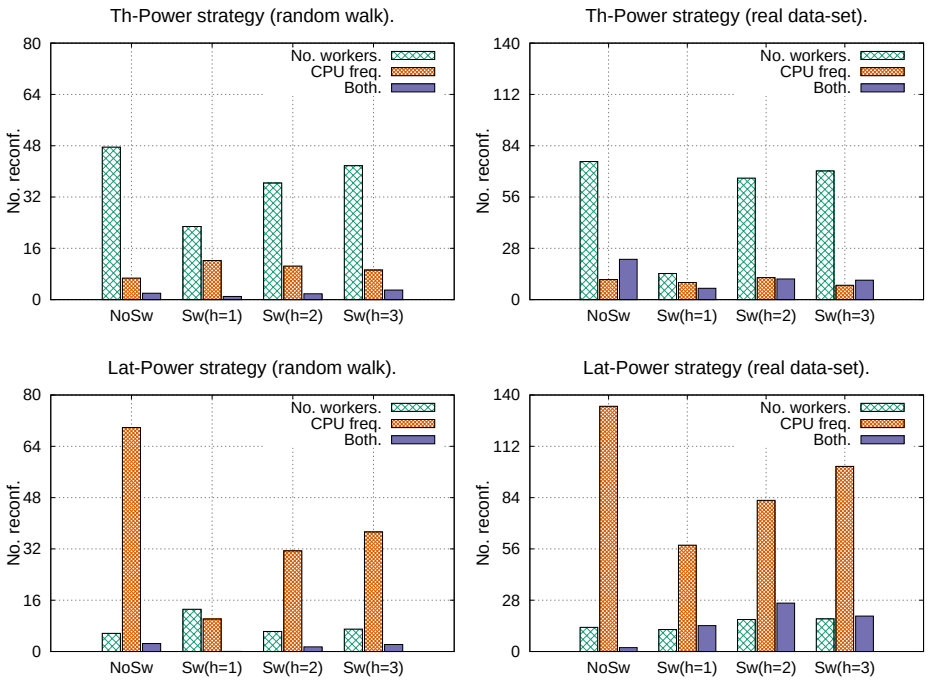


Figure 7.8: Types of reconfigurations: random walk and real workload.

- for latency-based strategies we detect a QoS violation each time the average latency measured during a control step is higher than a threshold δ. This value clearly depends on the workload and execution platform characteristics. Our intent is to show the effect of the switching cost on the accuracy achieved by the various strategies. Therefore if the threshold is too low, this will result in too many QoS violations since the performance constraints are excessive for the underlying hardware. If it is too high, the application is able to sustain the peak rate with few workers and the difference between different strategy configurations becomes minimal. Therefore, finding a good threshold is fundamental to increase the quality of the results and to make the previously described qualitative behavior more evident. Heuristically, we found that a good threshold must be able to produce few violations, e.g. < 10, with the most performing `Lat-Node` strategy (that as we will see is the one with Sw h=1) and the random walk workload.

Figure. 7.9 shows the QoS violations measured in the random walk workload with the `Th-Power` strategy. In the figure we detect a QoS violation each time the green line crosses the red region in the plots (we use a threshold $\theta = 0.95$). The strategy without

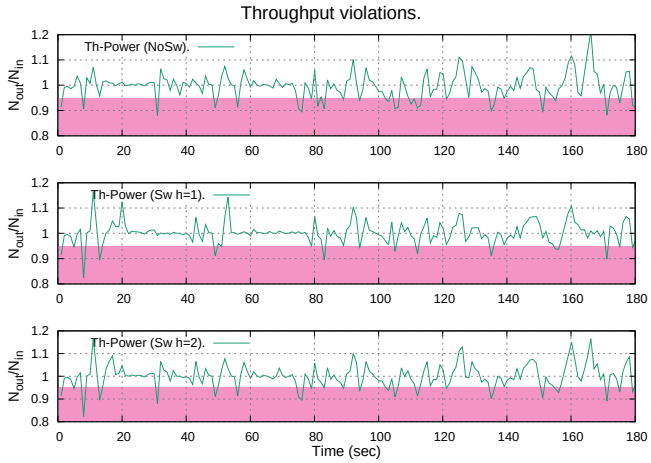
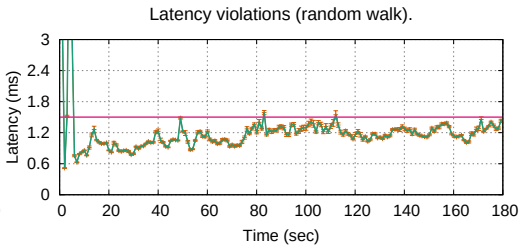


Figure 7.9: Number of throughput violations: strategy `Th-Power` (NoSw $h = 1$ and Sw $h = 1, 2$) and random walk.

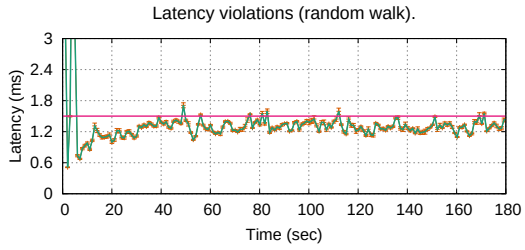
switching cost chooses at each step the minimal configuration such that the operator is not a bottleneck. If the arrival rate predictions are underestimated, the operator may become likely a bottleneck during some control steps and the ratio N_{out}/N_{in} assumes values lower than the threshold. Typically, the controller reacts by changing the configuration in the future steps. The effect is that the input tuples enqueued during the time periods in which the operator was bottleneck are consumed in the next control steps and the ratio N_{out}/N_{in} exhibits peaks greater than 1. This is the reason for the zig-zag pattern of the green line in the figure. The best accuracy is obtained by the strategy with switching cost and horizon $h = 1$. This is an expected result, as this strategy overshoots the configuration to use *(P4)*. This is evident in Figure 7.6c, where the reconfiguration sequence with switching cost is on top the one without it. By overshooting the configuration, the strategy is more capable of dealing with potential underestimations of the workload, by producing fewer QoS violations. By increasing the horizon length the accuracy worsens (26 violations with $h = 2$). Therefore, a proper length of the horizon allows to reach a trade-off between overshoot and accuracy.

Figures 7.10a, 7.10b and 7.10c show the latency violations achieved with `Lat-Power` in the random walk scenario. The threshold is set to 1.5 ms and it is represented by a red line in the plots. The figure reports the average latency without switching cost and with switching cost ($h = 1, 2$). Each experiment has been run 25 times. We measure the 95% confidence intervals (orange intervals in the figure) which are very small, demonstrating the small variance of the obtained results. The results confirm the same behavior seen before. Without switching cost we have more violations, with

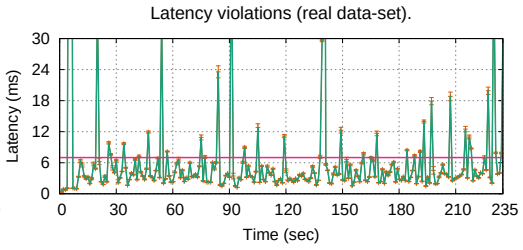
(a) `Lat-Power` NoSw ($h = 1$) - 18 violations.



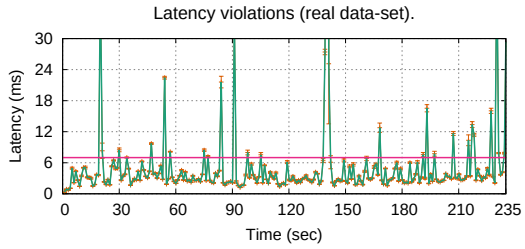
(b) `Lat-Power` Sw ($h = 1$) - 6 violations.



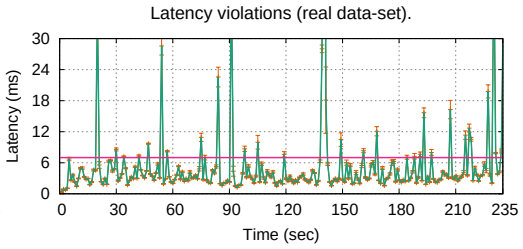
(c) `Lat-Power` Sw ($h = 2$) - 11 violations.



(d) `Lat-Power` NoSw ($h = 1$) - 38 violations.



(e) `Lat-Power` Sw ($h = 1$) - 29 violations.



(f) `Lat-Power` Sw ($h = 2$) - 33 violations.

Figure 7.10: Latency violations. Strategy `Lat-Power` with random walk workload (a, b, c) and real data-set (d, e, f).

the latency sometimes slightly higher than the threshold. We obtain fewer violations by using the switching cost and the minimum horizon. Longer horizons provide intermediate results. Figures 7.10d, 7.10e and 7.10f show the results with the real-data set in which, due to the high variability of the input rate, we use a threshold of 7 ms and more violations are observed.

Figure 7.11 shows a summary of the QoS violations. In conclusion we can state the following property:

Result 2. The switching cost allows the MPC strategy to reach better accuracy (P2). This positive effect is partially offset by increasing the horizon length.

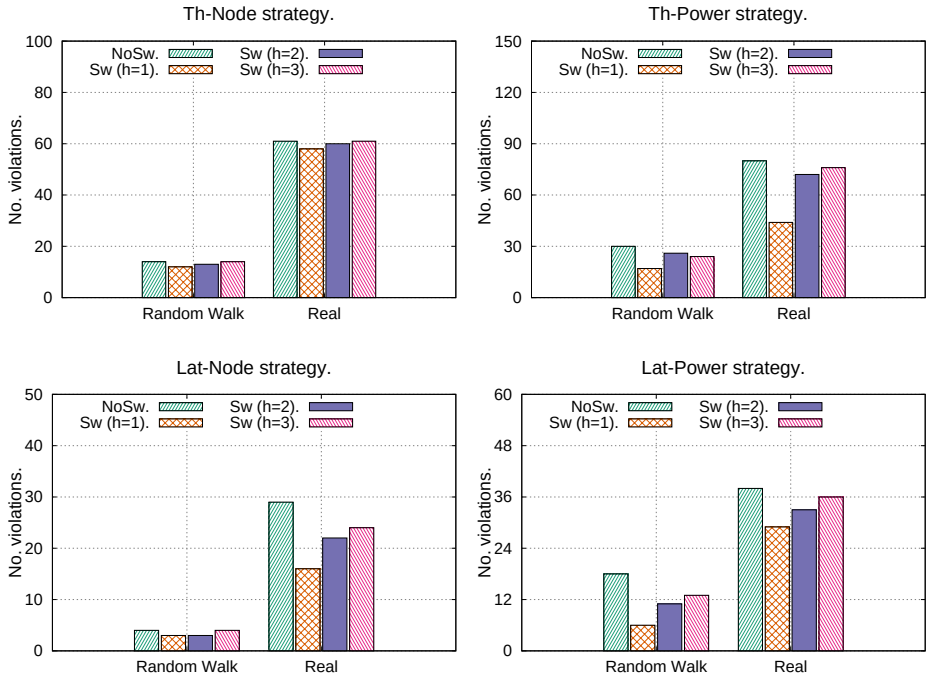


Figure 7.11: Number of QoS violations per strategy: random walk and real workload.

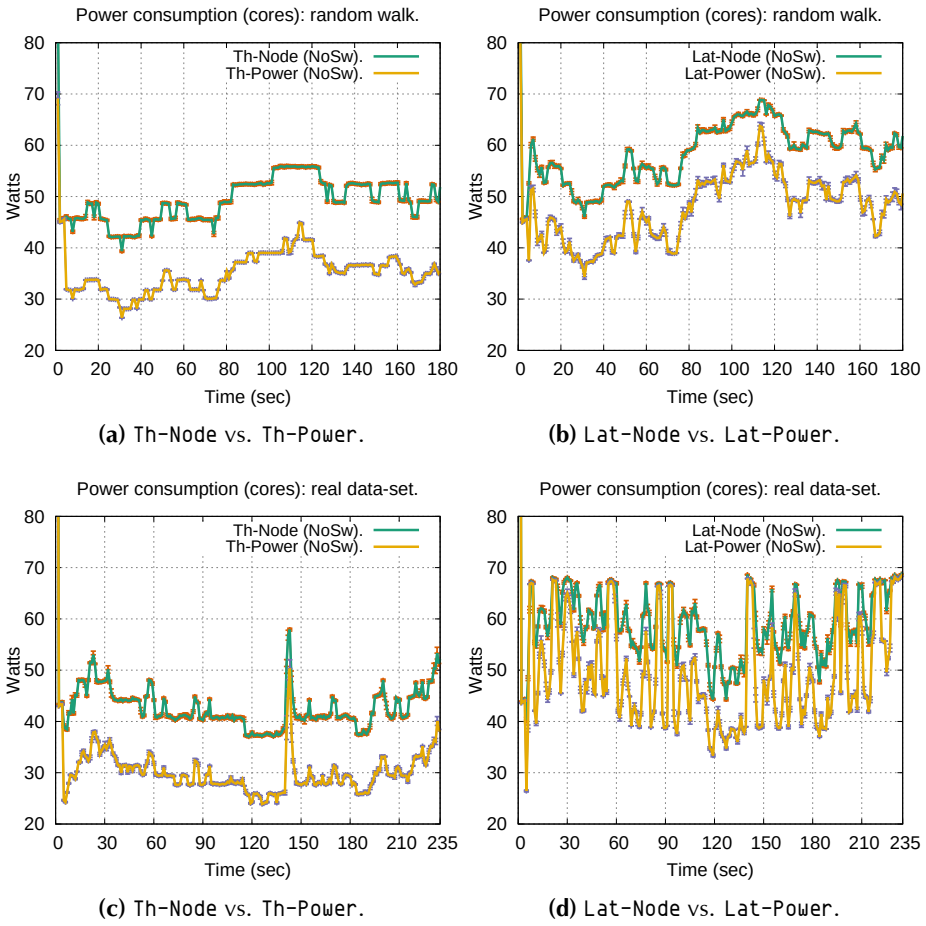
Resource and power consumption

We study the resource consumption achieved by our strategies. For the `*-Node` ones we measure the average number of workers used. For the `*-Power` strategies we consider the average power consumption in watts.

The two chips of our multicore always use the same frequency. Figure 7.12 shows the watts consumed by the strategies without switching cost[4]. Each plot compares the strategies in which the controller only changes the number of workers fixing the CPU frequency to 2 Ghz (`Th-Node` and `Lat-Node`) with the corresponding strategy with frequency scaling (`Th-Power` and `Lat-Power`). Figures 7.12a and 7.12b show the results in the random walk scenario. The watts measured with the power-efficient strategies always stay below the consumption without frequency scaling. This results in an average power saving of $14 \div 15$ watts (28%) and 11 watts (18.3%) for the `Th-Power` and `Lat-Power` strategies respectively.

A similar behavior can be observed in Figures 7.12c and 7.12d for the real data-set where we have more variability due to more reconfigurations. In summary, the `Th-`

[4]We measure the core energy counter. The overall socket consumption has additional $25 \div 30$ watts per step.



(a) `Th-Node` vs. `Th-Power`. **(b)** `Lat-Node` vs. `Lat-Power`.

(c) `Th-Node` vs. `Th-Power`. **(d)** `Lat-Node` vs. `Lat-Power`.

Figure 7.12: Power consumption (watts) of the non-switching cost strategies: random walk and real data-set workload.

`Power` saves $13 \div 14$ watts per second on average compared with `Th-Node`, while `Lat-Power` saves $10 \div 11$ watts with respect to `Lat-Node`. This results on an average power savings of 18.2% and 16.5% respectively. These reductions are significant especially owing to the long-running nature of DaSP computations. Very similar results are obtained considering strategies with the switching cost and different horizon lengths.

Figure 7.13 shows the global results. The strategy with switching cost and $h = 1$ consumes more resources/power. This is expected, since as discussed above this strategy tends to use more resources and to release them slower. The effect is that in some time periods this strategy overshoots the configuration, i.e. it uses more resources/a higher frequency than necessary. Therefore, we have:

Result 3. The switching cost causes overshoot *(P4)*. This can be mitigated by using longer horizon lengths.

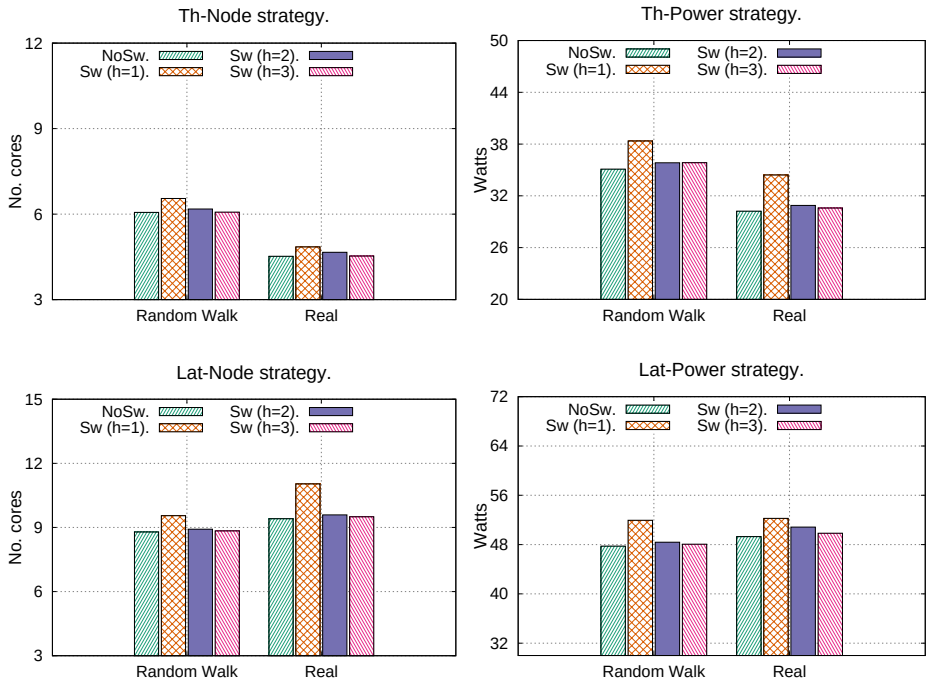


Figure 7.13: Resources/power consumed per strategy: random walk and real workload.

Reconfiguration amplitude

When the workload suddenly changes an effective strategy should be able to reach rapidly the new configuration that meets the QoS requirements. If the strategy takes small reconfigurations (e.g. few workers are added/removed each time), this negatively impacts the settling time property. Figure 7.14 shows for each strategy the *reconfiguration amplitude* measured over the entire execution. It consists in the average euclidean distance between the vector $\mathbf{u}(\tau)$ and the vector $\mathbf{u}(\tau - 1)$ for each τ. The frequency values (from 1.2 Ghz to 2 Ghz with steps of 0.1) have been normalized using the rule $(f(\tau) - 1.2) * 10 + 1$, thus obtaining the integers from 1 to 9.

As shown in Figure 7.14, the strategy with switching cost and $h = 1$ performs smaller reconfigurations. The highest amplitude is achieved by the strategy without the switching cost that follows rapidly the workload variation.

Result 4. The switching cost reduces the average reconfiguration amplitude. Better settling time *(P3)* can be achieved by using longer prediction horizons.

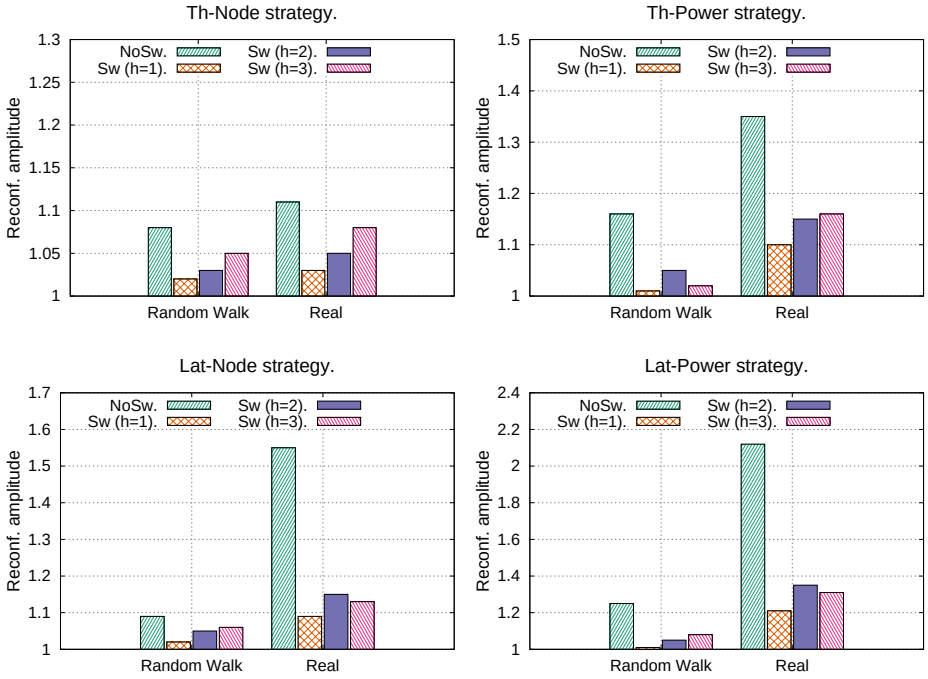


Figure 7.14: Settling time per strategy: random walk and real workload.

7.2.4 Comparison with peak-load overprovisioning

In this section we compare the best results achieved by our strategies against a static configuration in which the HFT operator is configured to sustain the peak load. In this scenario state migrations are eventually performed at each step to maintain the workload balanced among workers. However, the number of workers and the CPU frequency are statically set to the maximum value throughout the execution. The results are depicted in Figures 7.15 for both throughput and latency oriented strategies.

The peak-load configuration allows to achieve the minimum number of QoS violations both in the synthetic and the real workload traces. Just considering the `Lat-*` strategies, with the real workload we have 7 and 18 more violations achieved by `Lat-Node` and `Lat-Power` respectively. However, this static configuration has the highest power consumption. The relative power savings for the synthetic and real workload

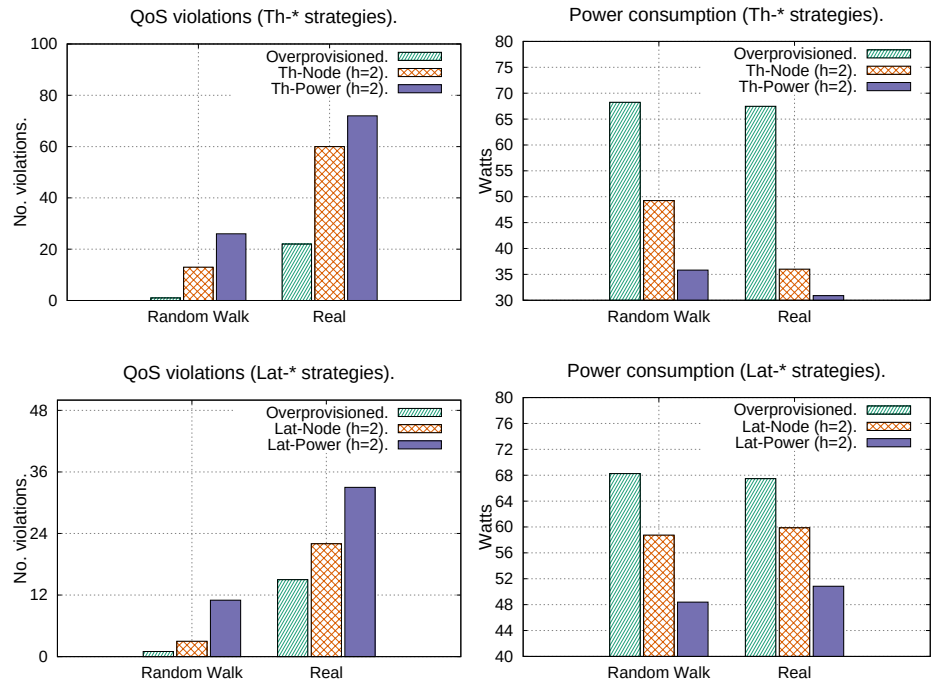


Figure 7.15: QoS violations and power consumption: comparison with the static peak-load overprovisioning strategy.

are respectively of 14% and 12% with `Lat-Node` (simply thanks to the reduction in number of used workers) and 29% and 25% with `Lat-Power`.

7.2.5 Comparison with similar approaches

Finally, we show a comparison with other reactive scaling strategies, considering throughput and latency oriented ones.

For what concern throughput based strategies, we have implemented a reactive policy based on *event-condition-action* rules widely adopted in Autonomic Computing (see Section 5.4.1). The strategy increases/decreases the number of workers if operator's utilization factor is over/under a maximum/minimum threshold (θ_{max} and θ_{min}). We refer to this strategy as `Th-Rule`.

In addition, we have implemented the control algorithm described in [Gedik et al., 2014] developed for the IBM IIS/SPL framework (`SPL-strategy` in the following), which is the only one addressing the SASO properties in the DaSP domain. That strategy measures a congestion index, i.e. the fraction of time the emitter is blocked in sending new tuples to the workers. A congestion is detected if the con-

gestion index is over a threshold. The strategy is throughput oriented and monitors the number of input tuples served by the operator in the last adaptation period (our control step). It changes the number of workers based on the congestion index and the recent history of the past actions. If an action taken in the recent past did not improve throughput, the control algorithm avoids executing it again. To adapt to fluctuating workload (as our real data-set), the authors introduce specific mechanisms to forget the recent history if the congestion index or the throughput change significantly. The algorithm uses a *sensitivity* parameter to determine what a significant change means.

Table 7.3 shows a summary of the results for the real data-set scenario (we omit the random walk one for brevity). Since both the rule-based strategy and the heuristic one target throughput optimization without frequency scaling, we compare them with our `Th-Node` strategy. For our strategy we use a horizon of 2 steps which gives the best trade-off between SASO properties. For the comparison we change the control step length (adaptation period) to 4 seconds because the `SPL-strategy` approach performs very poorly with too frequent steps. This is a shortcoming of this approach, which is unable to track the workload with a fine-grained sampling. The best values for the congestion threshold and sensitivity parameter are 0.1 and 0.9 in our scenario. In these experiments, the number of violations has been measured on a "second basis", rather than considering the control step. For the interested reader this is useful to compare these results with the previously reported ones.

	No. reconf.	**QoS viol.**	**Ampl.**	**No. workers**
Th-Rule*	39.17	62	1.07	4.63
Th-Rule**	29	59	1.06	4.53
SPL-strategy	40.18	58	1	4.63
Th-Node (4s)	11	56	1	4.51
Th-Node (2s)	24.77	54	1.04	4.50

Table 7.3: Comparison with similar throughput-oriented existing works . * $\theta_{max} = 0.9$ and $\theta_{min} = 0.8$. ** $\theta_{max} = 0.95$ and $\theta_{max} = 0.8$.

Table 7.4 shows a summary of the results for the real dataset scenario considering latency oriented strategies. The rule based policy (`Lat-Rule`), changes the number of workers if the relative difference between the measured latency and the required one is over/under a maximum/minimum threshold (ξ_{max} and ξ_{min}). For the sake of completeness, we report also the latency violations obtained with the SPL strategy. In all the aforementioned cases, our approach is the winner. Fewer reconfigurations

	No. reconf.	**QoS viol.**	Ampl.	**No. workers**
Lat-Rule	47.42	76	1	6.89
Lat-Node (4s)	11.0	30	1.2	9.97
SPL-strategy	40.18	230	1	4.63

Table 7.4: Comparison: average values with 25 tests per strategy. For the Lat-Rule strategy we have $\xi_{max} = 1.2$ and $\xi_{min} = 0.8$.

are performed (*stability*) with slightly fewer violations (*accuracy*). Furthermore, our approach uses a smaller number of workers on average (*overshoot*) with a comparable amplitude (*settling time*). This is a confirmation of the effectiveness of our predictive model-based approach.

7.3 Experiments on Distributed Memory

In this section we describe a proof of concept implementation of the HFT operator parallelization, reconfiguration mechanisms and strategies for a distributed memory architecture. The current implementation works with basic mechanisms (i.e. TCP/IP socket) and external services/libraries (i.e. Memcached, Google Protocol Buffer) whose use will be detailed in the ensuing discussion. The goal is to evaluate the effectiveness of the proposed solutions also in this execution scenario. Further refinements of the implementation or its porting in existing distributed and parallel frameworks is left as future work since do not directly impact with our objectives.

Given a homogeneous cluster i.e. composed by similar machines, we distinguish between two types of cluster *nodes*, over which we run the distributed operator (see Figure 7.16):

- a *master node*, which is responsible of executing the emitter, collector, controller and other services (i.e. the data source and services for the management of the remote repository);
- a set of *executor nodes*, which are the machines in charge of actually executing the workers of the operator.

We decided to use this distinction for easiness of implementation and for experimental purposes, to clearly separate the machines that are in charge of executing the business logic of the operator (i.e. the workers) from the one that runs the other

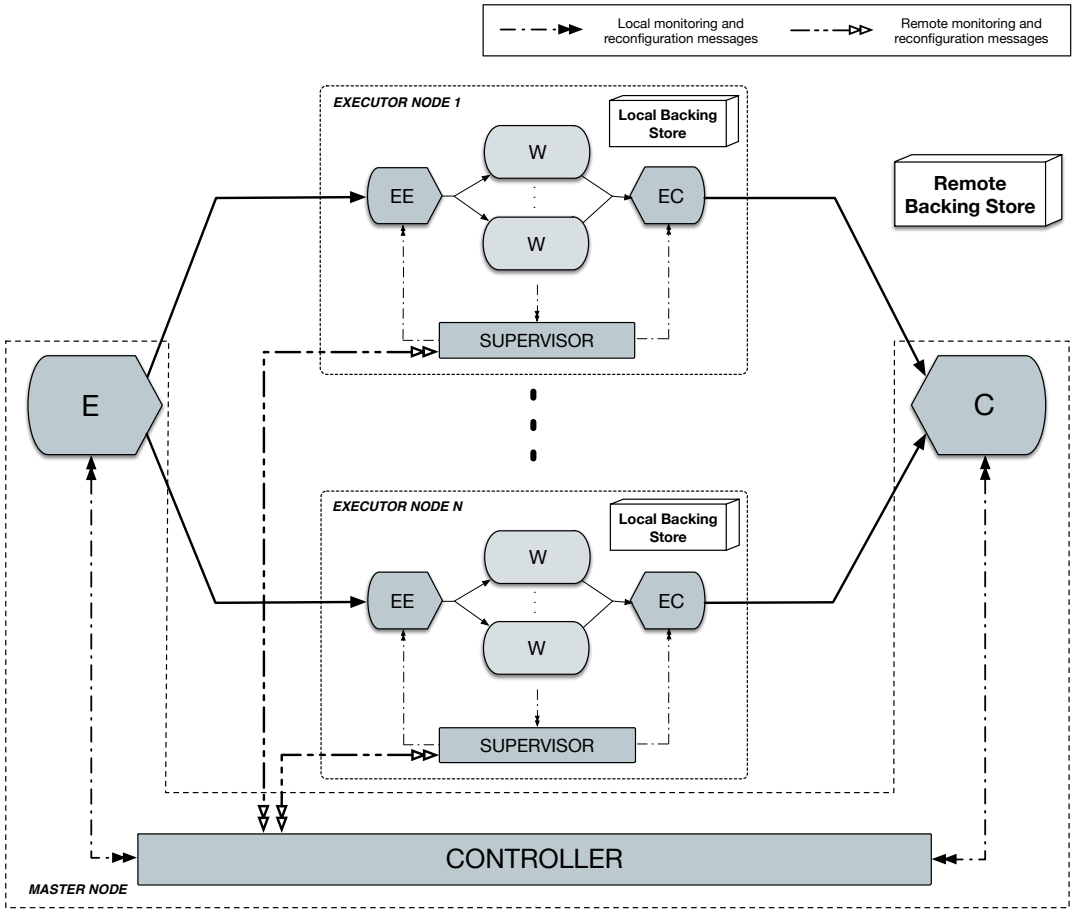


Figure 7.16: Schema of the HFT distributed operator.

functionality (i.e. emitter, collector, controller and backing stores). In a production deployment, we could use an appropriately downscaled node (not available in our execution environment) to execute the master's entities and backing stores.

As depicted in Figure 7.16, each executor node run a sort of replica of the HFT parallelized operator of Section 7.2. Apart from the workers, in each execution nodes are present additional entities that essentially act as interfaces with the master node:

- for each executor node we have an executor's emitter (*EE* in figure) and an executor's collector (*EC*). They are connected with the global emitter and collector and are respectively in charge of dispatching input tuples to the destination workers and gather the produced results and send them to the collector;
- an additional entity (called *supervisor*), is in charge of collecting all the measurements performed by the workers in a single monitoring message and send

it toward the controller at the beginning of each control step. On the other side, it will be also responsible for the creation/deletion of the workers inside the node.

These three entities are always in execution over the execution node: they are in a sleeping state if no worker is currently running on the node, but they are ready to create them when required. Thanks to their presence we are able to avoid too many connections between workers and emitter and collector (and the related cost of opening, handling and closing them). Communication channels from/to the master node to/from an executor node are implemented on TCP/IP socket. Communications that occur internally to each node are implemented using Fastflow shared queues.

As pointed out in Section 6.4, for the moment being, the controller is implemented as a centralized entity that is in charge of observing the whole operator execution and takes the adaptation decisions accordingly to the used strategy. Even if this is not the case, we recognize that a centralized controller could become a bottleneck and constitute a problem. A solution with decentralized or hierarchical controllers will be studied in the future. In figure is reported also the presence of Local (to each node) and Remote (shared between nodes) backing stores: their functionality will be detailed in Section 7.3.1.

The execution environment [5] is composed of 4 identical nodes interconnected through an Infiniband networks working at 40Gb/s. Each node is a dual CPU Intel Xeon E5-2699, for a total of 36 physical cores running at 2.30GHz (HyperThreading is turned off). Each core has a private L1 (32KB) and L2 (256KB) cache. Each CPU is equipped with a shared L3 cache of 45MB. Every machine has 192 GB of RAM. It runs a Linux based operating system. The used compiler is `gcc` (version `4.8.1`), programs are compiled with the `-O3` compiler flag. For the additional services/libraries, we have used: Fastflow (version `2.0.5`), Memcached (version `1.4.25`), Libmemcached (version `1.0.18`) and Google Protocol Buffer (version `proto2`).

For this experimental evaluation we do not take into account the power cost of the used reconfiguration. The reason is simply due to the fact that we do not have administrative privileges on the running machines in order to take the power consumption measurements and change the running CPU frequency. In any case, we already provided a detailed and accurate experimental discussion on the subject in Section 7.2 and it is reasonable to assume that similar results and conclusions hold also in this case.

[5] We would like to thank Centro di Calcolo Scientifico - INFN Pisa (Director: Dott. Alberto Ciampa) for their willingness and kindness in let us use these machines.

All the entities in Figure 7.16 are always implemented as threads in execution over the cores of the different nodes. Given this configuration, we use a node as master and three nodes as executors. Due to the presence of the executor's emitter, collector and supervisor, for each node we have a maximum of 33 allowable workers. Therefore, the total maximum number of workers is equal to 99. The source, responsible for generating the input data, is in execution on the master node.

These machines are in general more powerful with respect to one used in Section 7.2 (also considering the single core). Therefore the workload is heavier in order to better exploit and saturate all the available resources. The operator uses count based windows of size $|\mathcal{W}| = 2000$ tuples and slide $\delta = 10$ tuples. With respect to the previous configuration ($|\mathcal{W}| = 2000, \delta = 25$) this clearly result in coarser grain and more frequent windows activation.

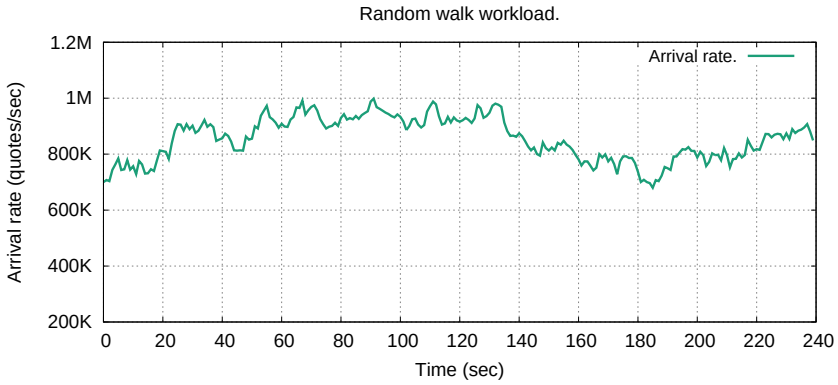
Also in this case we have a *synthetic* and a *real* data-set, shown in Figures 7.17a and 7.17b respectively. In the latter the quotes and their timestamps are the ones of a trading day of various stock markets (i.e. NASDAQ, NYSE, ...). This is a superset of dataset used in the shared memory evaluations. In this case we have $8,163$ traded stocks and the most frequent one constitute the 0.81% of the whole generated quotes. The dataset is accelerated 50 times to reproduce throttled input rates.

In the synthetic workload (Figure 7.2a) the arrival rate follows a random walk model. The key frequency distribution is fixed and equal to a random time-instant of the real data-set. The skew factor, i.e. the ratio between the most probable key and the less probable one, is equal to 21.6×10^4. The execution of the synthetic workload consists in 240 seconds while the one of the real data-set of 477 seconds, equal to about 6 hours and half in the original non-throttled data-set.

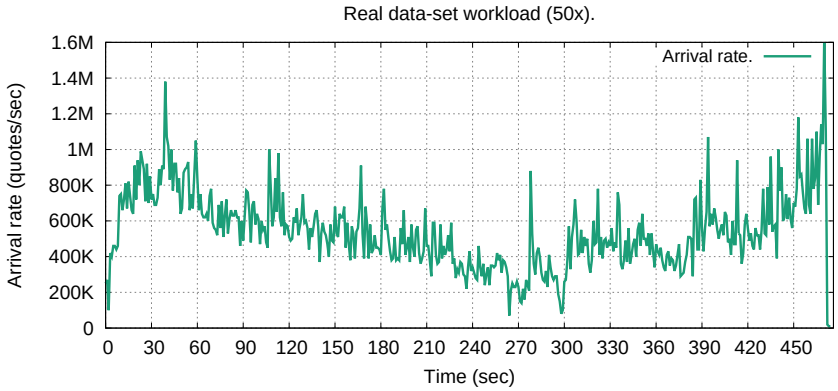
7.3.1 Reconfiguration mechanisms for a distributed memory architecture

The reconfiguration mechanisms presented in Section 6.4 have to be specialized for this execution platform. In this case we have messages that are internally exchanged between the entities of the various nodes, but also *remote* messages exchanged between nodes. We can recognize essentially two types of remote messages:

- *monitoring* messages: contain all the monitoring data gathered by the nodes supervisors. They are sent to the controller, one message per executor node at each control step;
- *reconfiguration* messages: sent by the controller to supervisors in order to implement a reconfiguration.



(a) Random walk.



(b) Real data-set.

Figure 7.17: Arrival rate: synthetic (random walk) and a real throttled ($50\times$) data-set.

In the current implementation all the data/messages remotely exchanged between data are serialized using the Google Protocol Buffer (`protobuf`) library [Google, 2016] and flow along the TCP/IP socket connections established between nodes.

At the program launch, on each executor nodes the emitter, collector and supervisors threads are created and respectively connected (through a socket) to the main emitter, collector and controller. They remain in a quiescence status, until it is required to create at least a worker on the node. Workers are carefully *packaged* in the executor nodes: only when there is no room for an additional worker in an executor (i.e. we used all the available cores) we start using the next executor. In this way, the number of used nodes is minimized. As usually, a routing table is used by the emitter to send the tuples to the right executor node. The emitter can easily derive the correspondence between the $worker\_id$ (Section 6.4.3) and the node in which it is in execution. It will be the executor's emitter that will further internally route the

tuple to the right worker.

Changes in the number of workers: once the controller decides a change in the number of workers, it notifies the decision to the emitter (with a new updated routing table) and to the supervisor of the involved nodes (more than one if workers have to be created/deleted on multiple nodes). In case of an increment in the number of workers, the supervisor of the executor node in which the workers must be created is in charge of instantiating the respective threads and queues for interconnection with the executor's emitter, collector and itself. Similarly, when a worker has to be removed, is the supervisor of the node in which it is in execution that handles its termination.

State migration: once the controller devises a new routing table, some data structures must be migrated in order to preserve the correctness of the computation. In the case of execution over a shared nothing cluster, we need a *remote* backing store in order to implement such data movements accordingly to the Fluent protocol described in Section 6.4.2. The repository can be implemented in various ways: for example by back-end databases or using socket-based or MPI-based implementations [Gedik et al., 2014]. We decided to implement it as a *distributed shared memory* area, in which workers can exchange data (windows in this case) and synchronize. Such repository has been implemented using the *Memcached* service [Fitzpatrick, 2004]. Memcached is essentially an in-memory key-value store: objects (i.e. data) are stored and retrieved by referring them by mean of a unique key. It is used widely in the data-center environment for caching results of database calls and in large organizations (such as Facebook, Wikipedia, Flickr) as back-end for fast replies to objects look-up. To implements the repository and the required interactions with the nodes, we need to distinguish between:

- the memcached server (`memcached`): is an active entity that is in charge of maintaining the repository. It will receive requests for saving an object with a given key, retrieving data by passing the key or deleting a key and its related data;
- a client that interacts with the server in order to require all the aforementioned operations. In the current implementation we use `libmemcached` [Data Differential, 2016], which is a C++ client library for `memcached` servers.

In our case, we have a single `memcached` server that is in execution on the master node. It is itself a parallel program, composed of 8 threads in charge of handling the

requests arriving from the various clients. For the HFT operator the object saved in the repository will be the windows migrating from a worker to another. The key used will simply be the stock symbol to which the window refers. The memcached server does not understand data: it simply accepts arrays of byte. Therefore windows (and all their content) are serialized using `protobuf`. Workers in executor nodes will use the `libmemcached` client to save windows to and import them from the `memcached` server.

As an optimization, we decided to use two levels of backing repository (see Figure 7.16):

- a *local* repository for each executor node is used for state migration among the workers in execution on the same node. In this case we can use a shared memory area (like in Section 7.2.1) avoiding the cost of serializing and copying the whole window;
- the *remote* repository based on `memcached` for data exchanges between nodes.

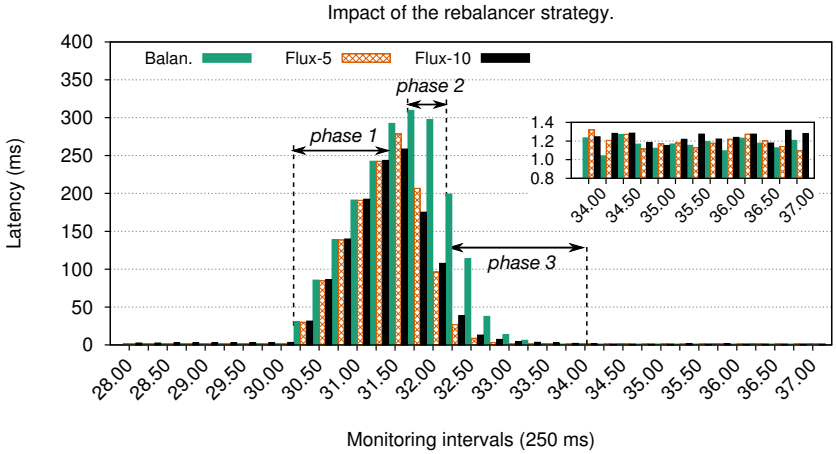
To handle this situation the only change in the Fluent protocol regards the type of messages sent by the emitter to the workers: we can have local movement messages (`move_out`/`move_in`) and remote movements (`rmove_out`/`rmove_in`) that are handled by workers accordingly. The emitter knows the topology and the deployment of the application over the different nodes and can easily distinguish between these two cases. If the state movement regards workers allocated on the same node it will send local movement messages, otherwise it will use the remote ones.

7.3.2 Mechanisms evaluation

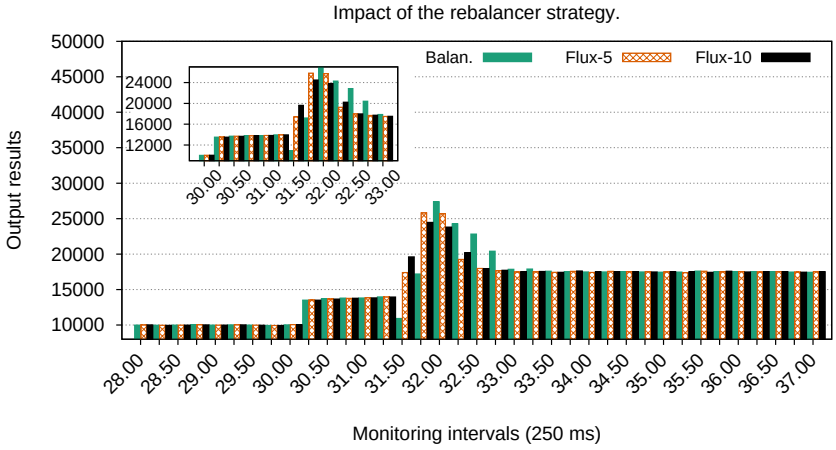
For this analysis we do not take into account aspects such as the overhead of the elastic support or the analysis of different migration protocols, that have been already exhaustively studied for the shared memory architecture. On the other hand, the main difference with respect to the shared memory implementation is the presence of the remote repository. Therefore it is interesting to evaluate its impact on the state migration costs.

On the same line to what done in Section 7.2.2, we study the *Balanced* and the *Flux* heuristics. For the latter we use two imbalance thresholds of 5% and 10% for the derivation of the routing table. We refer to these two situations as *Flux-5* and *Flux-10* respectively. We recall from Section 6.4.3 that the imbalance threshold represents an upper limit to the relative difference in the load between the most used worker and less used one. In the testing scenario, the operator is not a bottleneck and the

workload is balanced until timestamp 30. Then we force the input rate to change from $400K$ tuples/sec to $700K$ tuples/sec. At timestamp 31 the number of workers is changed from 30 to 60. In this way we pass from using only one executor node to use two nodes, involving in the migration also the remote repository. Results are reported on Figure 7.18.



(a) Latency per monitoring interval.



(b) Number of results per monitoring interval.

Figure 7.18: Impact of the rebalancer strategies for the distributed memory execution scenario: latency and number of results of the HFT operator in the time instants before, during and after a reconfiguration. Three different heuristics are evaluated: a *Balanced* one and two based on the *Flux* strategy, with an imbalance factor of 5% (Flux-5) and 10% (Flux-10).

Figure 7.18a shows the average latency measured each 250 ms. We can still identify three different phases:

- a *first phase* in which the rate changes and latency grows as the operator becomes a bottleneck. All the heuristics produce the same latency results;

- a *second phase* starts at timestamp 31: the controller triggers the reconfiguration, workers are created and keys are migrated. In this scenario we have that on average the Balanced strategy moves $8,061$ keys ($2,690$ of which are moved using the remote repository), Flux-5 moves 431 keys (389 remotely) and Flux-10 moves 418 keys (376 remotely). Clearly the fewer keys are moved (in particular remotely) the faster is the operator to complete the reconfiguration and the lower are the latency peaks in this phase. In fact, we have that the Balanced policy produces higher latency because more keys are migrated. Then we have Flux-5 and finally Flux-10 with the lowest latency peaks;

- in the *third phase*, workers process tuples have been accumulated in the pending buffers. Due to a better load balancing, Flux-5 approaches the steady state faster with respect Flux-10. Although it had the highest latency peak, even the Balanced strategy is able to approach the steady state in similar time to Flux-10 but later than Flux-5. During the steady state phase, as we can expect, the Balanced strategy has a slightly lower latency compared to the other two heuristics.

Figure 7.18b shows the number of results produced. The subplot shows a zoom of the central part of the figure. Immediately after timestamp 31 we have that the Balanced strategy produces fewer results with respect to others. This is due to the higher number of keys that must be migrated. Once the state migration is completed, it outperforms the other strategies: the higher number of produced results is due to a higher number of tuples enqueued in the various pending buffers during the migration phase.

Concluding we can state that the final considerations are reasonable different with respect to the case of the shared memory architecture. A small advantage of the Balanced strategy over the Flux ones is only noticeable in the steady state phase. In contrast, it produces the highest latency peaks and longest time needed to reach the steady state. This is due to the fact that it moves almost all the keys involved. Therefore, we can conclude that, on distributed memory architectures, a Flux policy with a low imbalance threshold is the just compromise between latency peaks and time to reach the steady state phase.

7.3.3 Adaptation strategies evaluation

Like in 7.2.3, we evaluate different MPC-based strategies. In the following we will assume that the rebalancer functionality adopts the Flux-5 policy, due to its advantages over the Balanced one. Since the imbalance threshold is very low, we still use the performance models that assume a balanced load: the discrepancy is very limited (5% among the most and less loaded workers) and does not justify the use of more complex cost model (like the ones introduced in Section 6.2.2).

For what concern the throughput based strategy, we have experimentally noticed that a formulation that uses as QoS cost the Equation 6.13 does not work well for this execution scenario. The strategy is able to resolve bottleneck situations slowly and requires long time to finish the transient phase after a configuration and reach a steady state. In our opinion, the reason is mainly due to the fact that we are in a much more complex execution scenario with respect to the case of a single isolated multicore machine: here the computation spans over multiple nodes, we have remote exchanges of data through an interconnection network with several mechanisms (e.g. socket and network buffers) whose behavior can impact on the transient phase if the right decision is not taken very rapidly. Moreover, such formulation returns the same QoS cost for all the configurations that result in an operator's utilization factor $\rho \leq 1$. That is, even if we increase the cost parameter α we are not able to take decision faster or in a more "performance-oriented" way.

For these reasons we decided to take into account a formulation of the throughput QoS cost slightly different and that recalls the one used for the latency strategy. It is defined as:

$$Q_{cost}\big(\widetilde{\mathbf{q}}(\tau+i)\big) = \alpha\, exp\big(\widetilde{\rho}(\tau+i)\big) \tag{7.1}$$

where $\widetilde{\rho}$ is the utilization factor of the operator, defined as $\widetilde{\rho}(\tau) = \widetilde{T}_S^{id}(\tau)/\widetilde{T}_A(\tau)$. The cost lies in the interval $(\alpha, \alpha e]$ for utilization factor in the interval $(0, 1]$. In this way, given two configurations $\mathbf{u}$ and $\mathbf{u}'$, if $\widetilde{\rho}_{\mathbf{u}} < \widetilde{\rho}_{\mathbf{u}'}$ we have that the QoS cost of the former will be slightly minor of the cost of the latter. On the other hand the cost heavily penalizes configurations with an expected utilization factor greater than one.

Table 7.5 shows the different tested strategies and the name used to refer to them. Again we evaluate the strategies without and with the switching cost. The QoS costs used for the `Lat-Node` strategy remains exactly the same of the shared memory architecture. As resource cost we use only the *per-node* cost since we do not have the administrative privileges to exploit the DVFS facility of the used CPUs. Clearly we have to adjust the cost parameters α, β and γ to the new execution scenarios (both machines and workloads) as well as the used latency thresholds. The resource cost

QoS cost	Resource cost	Name	alpha
throughput-based (Equation 7.1)	per node (Equation 6.15)	`Th-Node`	rw: 1 real: 2
latency-based (Equation 6.14)	per node (Equation 6.15)	`Lat-Node`	rw: 4 real: 6

Table 7.5: MPC-based strategies studied in the experiments. In all the cases we have $\beta = 0.5$. $\gamma = 0.2$ for `Th-Node` and $\gamma = 0.4$ for `Lat-Node`.

is $\beta = 0.5$ for both the strategies and workload, while the switching cost is set at $\gamma = 0.2$ for the `Th-Node` strategy and $\gamma = 0.4$ for the `Lat-Node` one. In all the presented experiments the control step is of 3 seconds: we have seen experimentally that for this application such time interval always guarantees the completion of the reconfiguration phase (e.g. state movements). For what concern the statistical predictions of disturbances, the same assumptions of the shared memory case hold also in this case: we predict the arrival rate using a Holt Winters filter, while frequencies and computation times per key will be estimated using the last measured values. All the experiments have been repeated 20 times by collecting the average measurements.

Evaluation of the SASO properties

For the sake of conciseness, the evaluation of the SASO properties achieved by our MPC-based strategies is shown in a more compact way with respect to the shared memory case. Figure 7.19 shows the results obtained. Each of the radar plots displays the averaged values of the measured metrics for the different strategies and workloads. For completeness the numeric results are reported also in Table 7.6.

All the metrics are measured in a similar way of the ones reported in Section 7.2.3. For what concern the accuracy we count the violations as the number of control steps in which the results obtained violate the QoS requirements. For `Th-Node` we look at the ratio between the number of results produced N_{out} and the number of received triggering tuples N_{in}, while for the `Lat-Node` strategy we consider the average latency measured over the control step. In these experiments, the control step has length equal to 3 seconds. The thresholds used for the `Th-Node` is $\theta = 0.95$; for `Lat-Node` we used a threshold of 5 ms and 25 ms respectively for the random walk and real workloads.

In general, the results are qualitatively the same of the ones experienced for the shared memory architectures. In particular, the switching cost:

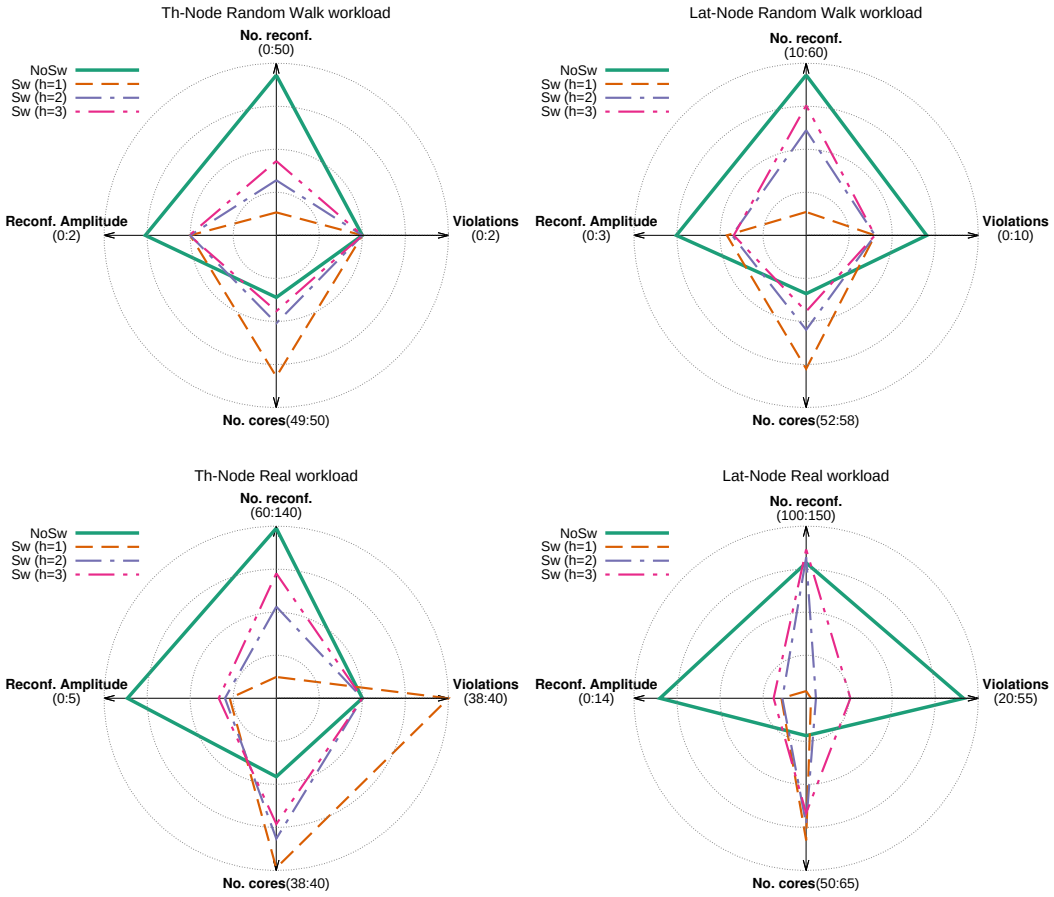


Figure 7.19: Radar plots for the SASO properties. Each plot shows using different radii the number of reconfigurations, violations, used cores and average reconfiguration amplitude for a different strategy and workload. For each spoke, it is indicated (in brackets) the minimum and maximum values that correspond respectively to the center and external edge of the plot. Results for different configuration of the switching cost are displayed in different colors and dashed lines.

- allows the strategy to reach a better stability (*Result 1*) and reduces the number of violations (*Result 2*);
- on the other hand, it overshoots the configuration (uses more resources, *Result 3*) and results in smaller reconfiguration amplitude (*Result 4*).

Like in the shared memory architecture, these effects are more pronounced with $h = 1$ and can be partially mitigated by increasing the horizon length. However in this case, as the interested reader can see from the results reported in Table 7.6, there are situations in which the effects of the switching cost and prediction horizon are not so evident like in the shared memory evaluations. For example consider the case

of the number of reconfigurations in the `Lat-Node` strategy with real workload: with switching cost and $h = 1$ we reach the lower number of reconfigurations, but this number is almost identically for all the other configurations (i.e. NoSw, Sw h=2 and Sw h=3). This could be due to a variety of reasons. Among the others, it should be considered the inaccuracy of predictions in this very dynamic scenario: as can be noticed in Figure 7.17b, the real workload is very noisy and irregular, making it very difficult for the used filter to obtain long term precise forecasts especially considering that the control step is of 3 seconds.

Strategy	Workload	Sw. Cost	Reconf.	Violat.	Cores	Amplit.
Th-Node	Rand. Walk	NoSw	46.5	1	49.36	1.52
		Sw h=1	6.67	1	49.82	1
		Sw h=2	16	1	49.51	1
		Sw h=3	21.58	1	49.44	1.01
Lat-Node	Rand. Walk	NoSw	56.53	7	54	2.26
		Sw h=1	16.79	4	56.65	1.38
		Sw h=2	40.53	4	55.29	1.29
		Sw h=3	47.73	4	54.65	1.26
Th-Node	Real	NoSw	139	39	38.91	4.32
		Sw h=1	69.92	40	39.97	1.35
		Sw h=2	102.58	39	39.63	1.49
		Sw h=3	118	39	39.46	1.67
Lat-Node	Real	NoSw	139.33	52	53.25	11.88
		Sw h=1	102.09	21	62.37	2
		Sw h=2	140.93	22	60.98	1.93
		Sw h=3	143.27	29	60	2.64

Table 7.6: Results for MPC-based strategies studied in the experiments.

7.3.4 Comparison with other approaches

To conclude this experimental section, we show a comparison of the results obtained with two reactive strategies and a peak load configuration that uses all the

available resources. Both the reactive strategies are based on policy rules and are the same used for the shared memory case. The first one (to which we refer as `Th-Rule`), increases/decreases the number of workers if the operator's utilization factor is over/under a maximum/minimum threshold (θ_{max} and θ_{min}). The second one takes into account the difference between the measured average latency and the required threshold, changing the number of workers if it is over/under a maximum/minimum threshold (ξ_{max} and ξ_{min}).

The results for the real workload are shown in Table 7.7, in which we consider throughput oriented strategies, and in Table 7.8, for latency based ones. In both cases we report the Peak Load configuration counting the QoS violations accordingly. In this configuration the state migrations are eventually performed at each step to maintain the workload quasi-balanced among the workers. For the MPC-Based strategies the results obtained with $h = 2$ are reported. Control steps have length equal to 3 seconds.

	No. reconf.	**QoS viol.**	**Ampl.**	**No. workers**
Th-Rule	134.83	45	2	36.84
Peak Load	-	35	-	99
Th-Node	102.58	39	1.49	39.63

Table 7.7: Comparison with throughput oriented strategy and peak load configuration. For `Th-Rule` we have $\theta_{max} = 0.9$ and $\theta_{min} = 0.8$.

	No. reconf.	**QoS viol.**	**Ampl.**	**No. workers**
Lat-Rule	147.2	43	3.09	46.36
Peak Load	-	19	-	99
Lat-Node	140.93	22	1.93	60.98

Table 7.8: Comparison with latency oriented strategy and peak load configuration. For the `Lat-Rule` strategy we have $\xi_{min} = 0.8$ and $\xi_{max} = 1.1$.

These comparisons show the effectiveness of our approach also in the case of a shared memory architecture. We have that fewer reconfigurations are performed with fewer violations with respect the rule based strategies. Compared with the peak load strategy, we obtain a slightly higher number of violations but using less resources.

7.4 Summary

In this chapter we evaluated our strategies in a high-frequency trading application in both shared and distributed memory context. In general the proposed solutions are able to minimize QoS violations, in terms of throughput or average latency, minimizing at the same time the resource used, in terms of used cores or consumed power. We compared the proposed MPC-based strategies in terms of the SASO properties, i.e. Stability, Accuracy, Settling time and Overshoot. The use of the switching cost and prediction horizons play an important role in achieving a good trade off in these properties. This effect is clearly visible in the shared memory case. The idea is graphically illustrated in Table 7.9. The best results correspond to three stars, whereas the worst ones are denoted by a single star. Two stars represent intermediate results (a good trade-off).

Strategy	Stability	Accuracy	Settling Time	Overshoot
NoSw	★	★	★★★	★★★
Sw h=1	★★★	★★★	★	★
Sw h=2	★★	★★	★★	★★

Table 7.9: Qualitative analysis: ★★★ denotes the best results, ★ the worst ones.

As experimentally evaluated, the configuration without switching cost is the worst in terms of stability (many reconfigurations are needed), the settling time is the best, the accuracy is the worst (many QoS violations) and, finally, the overshoot is the best, i.e. with this strategy configuration we always use the minimum number of workers or the minimum power to meet the QoS requirements. The configuration with switching cost and minimal horizon is the best in terms of stability (few reconfigurations) with low settling time (small reconfigurations on average). This configuration has a high overshoot, therefore the accuracy is high because we usually oversize the number of workers/CPU frequency. Finally, the strategy configuration with switching cost and a sufficiently long horizon reaches a good trade-off between the four SASO properties.

In the case of distributed memory architecture, these results are still visible. However there are situations in which the effect of the switching cost is not so prominent. This can be due to a variety of reasons: for example the fact that we are in a much more complex execution architecture with respect to the shared memory case or the difficulties in obtaining precise forecasting for the real workload. Further investigation of these issues is left as future works, due to the deep performance debugging

that is required to understand and resolve these minor problems. In a similar way, it is left to future works the possibility to incorporate performance cost models that take into account the presence of imbalance in the workload assigned to the various workers. In our opinion, this possibility is really meaningful when this imbalance has a certain relevance (e.g. $10 \div 20\%$ or higher). It should be however noticed that, unless this situation is explicitly imposed by using a heuristic with such imbalance degree, this could happen only in situations in which:

- we have few keys with respect to the number of available workers;
- or we have a highly unbalanced keys frequency distribution.

We recall from Chapter 4 that in similar cases the KP pattern is not the best solution to adopt, due to its well known load balancing problem. Therefore, in analogous situations, other parallel patterns should be used to parallelize the operator.

Finally, even if the proposed solutions have been tested on an HFT application, they are clearly beneficial also in all the applications in which QoS requirements and resources consumption are major concerns. Other examples could be healthcare diagnostic systems that process sensor data in real time to anticipate urgent medical interventions, network security systems that prevent intrusions, transportation monitoring systems in which sensor data is analyzed to detect anomalous behaviors and prevent catastrophic scenarios. In these application contexts performance guarantees are fundamental and a proactive strategy enabling adaptive processing is of great importance to meet the performance requirements with high probability by reducing the operating costs.

8 Conclusions

Data Stream Processing is one of the IT trending topics today. It follows the interest created by the so called "Big Data" in the scientific communities as well as in businesses. This thesis gives some insights on the fascinating challenges that DaSP applications pose and suggests some initials solutions to the problems of parallelism exploitation and autonomic behavior. Due to stringent QoS requirements and very dynamic execution scenarios, these applications must:

- be able to efficiently exploit intra-operator parallelism to obtain the desired performance;
- be able to adapt to dynamic workloads and changing conditions by automatically adjusting the used resources, possibly in a cost effective manner.

This thesis runs along these two directions that, in our opinion, are not completely and exhaustively covered by existing frameworks and literature. The leitmotif of the dissertation has been the exploitation of a structured approach to solve the problem of achieving intra-operator parallelism. This simplifies the programming by raising the level of abstraction presented to application programmer: when facing the problem of parallelizing a DaSP operator the programmer can instantiate a pattern available among a set of re-usable parallel solutions, taking into account the problem characteristics. For these patterns ready-to-use parametric implementations can be provided by high level programming tools. In this way the programmer has to specify only the functional details of the operator (e.g. type and characteristics of the used window and the function to compute) while all the implementation details are hidden and completely encapsulated in the used programming tool and runtime support. Furthermore, the knowledge of the communication/computation pattern implied by the use of well know parallelization schemes, allows the development of reconfiguration

mechanisms that can be provided directly by the runtime support and of adaptation strategies that may benefit from the presence of performance models to achieve the desired QoS requirements.

We started from the study of recurrent computations that arise in *window based stateful operators*, which are the most representative class of stateful data stream operators. In Chapter 4 we proposed parallel exploitation patterns: a set of four different re-usable parallelism schemes are identified, each one targeting particular computations and exhibiting its own peculiarities in terms of applicability, impact on performance and issues. Due to the vast range of computations that they can cover, the possibility to integrate them in existing DaSP or Skeletal parallel framework should deserve special consideration from these communities.

The problem of the autonomic behavior of a DaSP operator has been addressed in the second part of the thesis, taking into account one of the previously studied patterns. The deep knowledge of its parallelization schema allows us to define optimized reconfiguration mechanisms with low performance impacts. A set of predictive scaling strategies has been devised. They are based on the Model Predictive Control approach and present important aspects that are missing or not covered by existing frameworks/literature:

- they maintain desired QoS levels in terms of throughput or average latency;
- they deal with resource/power consumption issues;
- they are predictive and try to anticipate corrective actions by using forecasting tools.

The proposed approach has been evaluated in both shared and distributed memory architectures. Interestingly enough, we were able to obtain qualitatively similar behaviors in both cases, highlighting the effectiveness of our solutions. The application of MPC as well as the devising of strategies that take into account latency constraints and energy consumptions are quite new subjects for the DaSP community. The application of the studied methods can be clearly extended to the other proposed patterns.

The methods and solutions presented in this thesis are just first steps towards the resolution of problems related to the high performance exploitation and adaptive management of DaSP applications. Serial research directions can be traced starting from the current work: some of these aim at completing the picture partially drawn up to here. Others go beyond and look towards other interesting and related issues.

From a parallelism point of view, windows are not the only state representation in DaSP applications. Another representative class of DaSP computations are *sketch* based ones ([Aggarwal and Yu, 2007]), in which operators maintain a summary of the stream in order to compute the results. Parallel pattern solutions can be beneficial also in the parallelization of those cases. Another interesting feature is the ability to handle stream imperfections, such as delayed or out-of-order data [Stonebraker et al., 2005]. Tuples may exhibit this characteristic due to the fact that data sources produces data in a disordered way or simply because data arrives from multiple sources multiplexed together. Due to the existence of late arrivals, it is difficult to determine when a window has received all tuples falling into the window scope. This clearly has an impact on how parallelism can be exploited in operators that should deal with similar problems. Typical solutions resort in using *punctuations*, i.e. special tuples that indicate particular properties of the input data (e.g. "from now on you will receive tuple with timestamps higher than 10am"). This kind of problem is partially treated in existing literature (like in [Ji et al., 2015]) and framework ([Apache Flink, 2016]), typically without taking into account parallelism.

For what concern adaptivity, it is meaningful to extend the proposed mechanisms and scaling strategies to the other proposed patterns (not only Key Partitioning), provided that these solution can be effectively encapsulated in production frameworks, and to enhance the power consumption model. More scientifically appealing could be the study of other issues related with the autonomic behavior. Just to mention two possibilities:

- concerning the single operator, the space of all admissible configurations that must be explored by the controller grows exponentially with the configuration options and horizon length. In the thesis we partially address the problem of complexity reduction using B&B techniques. However, a similar solution requires that the cost function is monotonically increasing with the prediction steps over the horizon. Possible alternative solutions could be found in *greedy search algorithms*, to obtain sub-optimal solutions, or in *evolutionary* algorithms. Moreover, in a distributed environment *hierarchical* controllers could be beneficial to solve the computational problems as well as scalability and reliability issues;

- in this thesis we focus on the management of the single operators. Operators are the building blocks of more complex DaSP applications, organized in computation graph. Integrating MPC strategies also in a complete graph context is an interesting future work direction. In this case, the decisions taken by an op-

erator controller may influence the behavior of other parts of the computation. For example a bottleneck operator slows down the execution of its preceding operators due to backpressure phenomena. Therefore, control operators need to coordinate to find agreements in the scaling decisions. Similar problems have been already investigated in the research group (e.g. in [Mencagli, 2015]) and already proposed solution can find applicability also in this case.

Bibliography

Abadi, D. J., Ahmad, Y., Balazinska, M., Cherniack, M., hyon Hwang, J., Lindner, W., Maskey, A. S., Rasin, E., Ryvkina, E., Tatbul, N., Xing, Y., and Zdonik, S. (2005). The design of the borealis stream processing engine. In *In CIDR*, pages 277–289.

Abadi, D. J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., and Zdonik, S. (2003). Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139.

Abdelwahed, S., Bai, J., Su, R., and Kandasamy, N. (2009). On the application of predictive control techniques for adaptive performance management of computing systems. *Network and Service Management, IEEE Transactions on*, 6(4):212–225.

Abdelwahed, S., Kandasamy, N., and Neema, S. (2004). Online control for self-management in computing systems. In *Real-Time and Embedded Technology and Applications Symposium, 2004. Proceedings. RTAS 2004. 10th IEEE*, pages 368–375.

Aggarwal, C. C. and Yu, P. S. (2007). *Data Streams: Models and Algorithms*, chapter A Survey of Synopsis Construction in Data Streams, pages 169–207. Springer US, Boston, MA.

Akidau, T., Balikov, A., Bekiroglu, K., Chernyak, S., Haberman, J., Lax, R., McVeety, S., Mills, D., Nordstrom, P., and Whittle, S. (2013). Millwheel: Fault-tolerant stream processing at internet scale. In *Very Large Data Bases*, pages 734–746.

Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernandez-Moctezuma, R. J., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E., and Whittle, S. (2015). The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8:1792–1803.

Aldinucci, M., Campa, S., Danelutto, M., and Vanneschi, M. (2008). Behavioural skeletons in gcm: Autonomic management of grid components. In *Parallel, Distributed and Network-Based Processing, 2008. PDP 2008. 16th Euromicro Conference on*, pages 54–63.

Aldinucci, M., Coppola, M., Vanneschi, M., Zoccolo, C., and Danelutto, M. (2006a). *Grid Computing: Software Environments and Tools*, chapter ASSIST As a Research Framework for High-Performance Grid Programming Environments, pages 230–256. Springer London, London.

Aldinucci, M., Danelutto, M., and Kilpatrick, P. (2009). Autonomic management of non-functional concerns in distributed parallel application programming. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12.

Aldinucci, M., Danelutto, M., Kilpatrick, P., and Torquati, M. (2014a). Fastflow: high-level and efficient streaming on multi-core. In Pllana, S. and Xhafa, F., editors, *Programming Multi-core and Many-core Computing Systems*, Parallel and Distributed Computing, chapter 13. Wiley.

Aldinucci, M., Danelutto, M., and Teti, P. (2003). An advanced environment supporting structured parallel programming in java. *Future Gener. Comput. Syst.*, 19(5):611–626.

Aldinucci, M., Danelutto, M., and Vanneschi, M. (2006b). Autonomic QoS in ASSIST grid-aware components. In *Proc. of Intl. Euromicro PDP 2006: Parallel Distributed and network-based Processing*, pages 221–230, Montbéliard, France. IEEE.

Aldinucci, M., Torquati, M., Spampinato, C., Drocco, M., Misale, C., Calcagno, C., and Coppo, M. (2014b). Parallel stochastic systems biology in the cloud. *Briefings in Bioinformatics*, 15(5):798–813.

Alexandrov, A., Bergmann, R., Ewen, S., Freytag, J.-C., Hueske, F., Heise, A., Kao, O., Leich, M., Leser, U., Markl, V., Naumann, F., Peters, M., Rheinländer, A., Sax, M. J., Schelter, S., Höger, M., Tzoumas, K., and Warneke, D. (2014). The stratosphere platform for big data analytics. *The VLDB Journal*, 23(6):939–964.

Aly, A. M., Sallam, A., Gnanasekaran, B. M., Nguyen-Dinh, L.-V., Aref, W. G., Ouzzani, M., and Ghafoor, A. (2012). M3: Stream processing on main-memory

mapreduce. In Kementsietsidis, A. and Salles, M. A. V., editors, *ICDE*, pages 1253–1256. IEEE Computer Society.

Andrade, H., Gedik, B., Wu, K.-L., and Yu, P. (2009). Scale-up strategies for processing high-rate data streams in system s. In *Data Engineering, 2009. ICDE '09. IEEE 25th International Conference on*, pages 1375–1378.

Andrade, H., Gedik, B., Wu, K. L., and Yu, P. S. (2011). Processing high data rate streams in system s. *J. Parallel Distrib. Comput.*, 71(2):145–156.

Andrade, H. C., Gedik, B., and Turaga, D. S. (2014). *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*. Cambridge University Press.

Apache (2016). Apache spark streaming documentation. `http://spark.apache.org/docs/1.6.0/streaming-programming-guide.html`.

Apache Flink (2016). `http://flink.apache.org/`.

Apache Hadoop (2016). `http://hadoop.apache.org/`.

Apache Samoa (2016). `https://samoa.incubator.apache.org/`.

Apache Samza (2016). `http://samza.apache.org/`.

Apache Spark (2016). `http://spark.apache.org/`.

Apache Storm (2016). `http://storm.apache.org/`.

Arasu, A., Babcock, B., Babu, S., Datar, M., Ito, K., Nishizawa, I., Rosenstein, J., and Widom, J. (2003). Stream: The stanford stream data manager. *IEEE Data Engineering Bulletin*, 26.

Babcock, B., Babu, S., Datar, M., Motwani, R., and Widom, J. (2002). Models and issues in data stream systems. In *Proceedings of the twenty-first ACM symposium on Principles of database systems*, PODS '02, pages 1–16, New York, NY, USA. ACM.

Babcock, B., Datar, M., and Motwani, R. (2004). Load shedding for aggregation queries over data streams. In *Data Engineering, 2004. Proceedings. 20th International Conference on*, pages 350–361. IEEE.

Bacci, B., Danelutto, M., Orlando, S., Pelagatti, S., and Vanneschi, M. (1995). P3l: A structured high-level parallel language, and its structured support. *Concurrency: Practice and Experience*, 7(3):225–255.

Balkesen, C., Dindar, N., Wetter, M., and Tatbul, N. (2013a). Rip: Run-based intra-query parallelism for scalable complex event processing. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 3–14, New York, NY, USA. ACM.

Balkesen, C. and Tatbul, N. (2011). Scalable Data Partitioning Techniques for Parallel Sliding Window Processing over Data Streams. In *VLDB International Workshop on Data Management for Sensor Networks (DMSN'11)*, Seattle, WA, USA.

Balkesen, C., Tatbul, N., and Özsu, M. T. (2013b). Adaptive input admission and management for parallel stream processing. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 15–26, New York, NY, USA. ACM.

Ballard, C., Foster, K., Frenkiel, A., Gedik, B., Koranda, M. P., Nathan, S., Rajan, D., Rea, R., Spicer, M., Williams, B., et al. (2012). *IBM Infosphere Streams: Assembling Continuous Insight in the Information Revolution*. IBM Redbooks.

Bertolli, C. (2008). *Fault tolerance for high-performance applications using structured parallelism models*. PhD thesis, University of Pisa.

Bertolli, C., Buono, D., Mencagli, G., and Vanneschi, M. (2010). *Autonomic Computing and Communications Systems: Third International ICST Conference, Autonomics 2009, Limassol, Cyprus, September 9-11, 2009, Revised Selected Papers*, chapter Expressing Adaptivity and Context Awareness in the ASSISTANT Programming Model, pages 32–47. Springer Berlin Heidelberg, Berlin, Heidelberg.

Bloomberg (2016). The bloomberg market data feed (b-pipe). `http://www.bloomberg.com/enterprise/content-data/market-data/`.

Bollen, J., Mao, H., and Zeng, X. (2011). Twitter mood predicts the stock market. *Journal of Computational Science*, 2(1):1 – 8.

Brito, A., Martin, A., Knauth, T., Creutz, S., Becker, D., Weigert, S., and Fetzer, C. (2011). Scalable and Low-Latency Data Processing with StreamMapReduce. In *3rd IEEE International Conference on Cloud Computing Technology and Science*, pages 48 –58, Los Alamitos, CA, USA. IEEE Computer Society.

Brook, A. (2015). Low-latency distributed applications in finance. *Commun. ACM*, 58(7):42–50.

Buono, D., De Matteis, T., and Mencagli, G. (2014a). A high-throughput and low-latency parallelization of window-based stream joins on multicores. In *12th IEEE International Symposium on Parallel and Distributed Processing with Applications*, pages 117–126, Milano, Italy.

Buono, D., Mencagli, G., Pascucci, A., and Vanneschi, M. (2014b). Performance analysis and structured parallelisation of the space–time adaptive processing computational kernel on multi-core architectures. *Int. J. Parallel Emerg. Distrib. Syst.*, 29(5):460–498.

Butenhof, D. R. (1997). *Programming with POSIX Threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Camacho, E. F. and Bordons Alba, C. (2007). *Model Predictive Control*. Springer Berlin Heidelberg.

Castro Fernandez, R., Migliavacca, M., Kalyvianaki, E., and Pietzuch, P. (2013). Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 725–736, New York, NY, USA. ACM.

Chandrakasan, A. and Brodersen, R. (1995). Minimizing power consumption in digital cmos circuits. *Proceedings of the IEEE*, 83(4):498–523.

Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M. J., Hellerstein, J. M., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F., and Shah, M. A. (2003). Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*.

Ciechanowicz, P., Poldner, M., and Kuchen, H. (2009). The münster skeleton library muesli: A comprehensive overview. ERCIS Working Papers 7, University of Münster, European Research Center for Information Systems (ERCIS).

Cochran, R., Hankendi, C., Coskun, A., and Reda, S. (2011). Identifying the optimal energy-efficient operating points of parallel workloads. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '11, pages 608–615, Piscataway, NJ, USA. IEEE Press.

Cole, M. (1988). *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, Cambridge, MA, USA.

Cole, M. (2004). Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Comput.*, 30(3):389–406.

Condie, T., Conway, N., Alvaro, P., Hellerstein, J. M., Elmeleegy, K., and Sears, R. (2010). Mapreduce online. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 21–21, Berkeley, CA, USA. USENIX Association.

Coppola, M. and Vanneschi, M. (2002). High-performance data mining with skeleton-based structured parallel programming. *Parallel Computing*, 28(5):793–813.

Cranor, C., Johnson, T., Spataschek, O., and Shkapenyuk, V. (2003). Gigascope: a stream database for network applications. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, pages 647–651, New York, NY, USA. ACM.

Cugola, G. and Margara, A. (2012a). Complex event processing with t-rex. *J. Syst. Softw.*, 85(8):1709–1728.

Cugola, G. and Margara, A. (2012b). Low latency complex event processing on parallel hardware. *J. Parallel Distrib. Comput.*, 72(2):205–218.

Cugola, G. and Margara, A. (2012c). Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62.

Dagum, L. and Menon, R. (1998). Openmp: an industry standard api for shared-memory programming. *Computational Science Engineering, IEEE*, 5(1):46–55.

Danelutto, M. (2001). Efficient support for skeletons on workstation clusters. *Parallel Processing Letters*, 11(1):41–56.

Danelutto, M. and Dazzi, P. (2006). *Computational Science – ICCS 2006: 6th International Conference, Reading, UK, May 28-31, 2006. Proceedings, Part II*, chapter Joint Structured/Unstructured Parallelism Exploitation in muskel, pages 937–944. Springer Berlin Heidelberg, Berlin, Heidelberg.

Danelutto, M., De Sensi, D., and Torquati, M. (2015). Energy driven adaptivity in stream parallel computations. In *Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on*, pages 103–110.

Danelutto, M. and M., S. (2000). SKElib: parallel programming with skeletons in C. In A., B., T., L., W.l, K., and R., W., editors, *Euro-Par 2000 Parallel Processing*, number 1900 in Lecture Nodets in Computer Science, pages 1175–1184. Springer Berlin Heidelberg.

Darlington, J., Guo, Y.-k., To, H. W., and Yang, J. (1995). Parallel skeletons for structured composition. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 19–28, New York, NY, USA. ACM.

Das, T., Zhong, Y., Stoica, I., and Shenker, S. (2014). Adaptive stream processing using dynamic batch sizing. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 16:1–16:13, New York, NY, USA. ACM.

Data Differential (2016). Libmemcached. `http://libmemcached.org/`.

De Matteis, T., Di Girolamo, S., and Mencagli, G. (2015). A multicore parallelization of continuous skyline queries on data streams. In *Proceedings of the 2015 International Conference on Parallel Processing (Euro-Par)*, pages 402–413, Vienna, Austria.

Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113.

Enmyren, J. and Kessler, C. W. (2010). Skepu: A multi-backend skeleton programming library for multi-gpu systems. In *Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications*, HLPP '10, pages 5–14, New York, NY, USA. ACM.

EsperTech (2016). Esper. `http://www.espertech.com/esper/`.

Etzion, O. and Niblett, P. (2010). *Event Processing in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition.

Feng, X., Ge, R., and Cameron, K. W. (2005). Power and energy profiling of scientific applications on distributed systems. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 34–34.

Fitzpatrick, B. (2004). Distributed caching with memcached. *Linux J.*, 2004(124):5–.

Fried, R. and George, A. (2014). Exponential and holt-winters smoothing. In Lovric, M., editor, *International Encyclopedia of Statistical Science*, pages 488–490. Springer Berlin Heidelberg.

Gabriele (2012). *Adaptiveness in Structured Parallel Computations: a Control-theoretic Methodology*. PhD thesis, Universitá di Pisa.

Gedik, B. (2014). Partitioning functions for stateful data parallelism in stream processing. *The VLDB Journal*, 23(4):517–539.

Gedik, B., Schneider, S., Hirzel, M., and Wu, K.-L. (2014). Elastic scaling for data stream processing. *Parallel and Distributed Systems, IEEE Transactions on*, 25(6):1447–1463.

Gedik, B., Wu, K.-L., Yu, P. S., and Liu, L. (2007). Grubjoin: An adaptive, multiway, windowed stream join with time correlation-aware cpu load shedding. *IEEE Trans. on Knowl. and Data Eng.*, 19(10):1363–1380.

González-Vélez, H. and Leyton, M. (2010). A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers. *Softw. Pract. Exper.*, 40(12):1135–1160.

Google (2016). Protocol buffer. `https://developers.google.com/protocol-buffers/`.

Gropp, W., Lusk, E., Doss, N., and Skjellum, A. (1996). A high-performance, portable implementation of the {MPI} message passing interface standard. *Parallel Computing*, 22(6):789 – 828.

Gross, D., Shortle, J. F., Thompson, J. M., and Harris, C. M. (2008). *Fundamentals of Queueing Theory*. Wiley-Interscience, New York, NY, USA, 4th edition.

Gulisano, V., Jimenez-Peris, R., Patino-Martinez, M., Soriente, C., and Valduriez, P. (2012). Streamcloud: An elastic and scalable data streaming system. *IEEE Transactions on Parallel and Distributed Systems*, 23(12):2351–2365.

Gundencha, P. and Liu, H. (2012). *Mining Social Media: A Brief Introduction*, chapter 1, pages 1–17. Informs tutorial in operational research.

Hähnel, M., Döbel, B., Völp, M., and Härtig, H. (2012). Measuring energy consumption for short code paths using rapl. *SIGMETRICS Perform. Eval. Rev.*, 40(3):13–17.

Heinze, T., Jerzak, Z., Hackenbroich, G., and Fetzer, C. (2014a). Latency-aware elastic scaling for distributed data stream processing systems. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, pages 13–22, New York, NY, USA. ACM.

Heinze, T., Jerzak, Z., Hackenbroich, G., and Fetzer, C. (2014b). Latency-aware elastic scaling for distributed data stream processing systems. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, pages 13–22, New York, NY, USA. ACM.

Hellerstein, J. L., Diao, Y., Parekh, S., and Tilbury, D. M. (2004). *Feedback Control of Computing Systems*. John Wiley & Sons.

Herbst, N. R., Huber, N., Kounev, S., and Amrehn, E. (2013). Self-adaptive workload classification and forecasting for proactive resource provisioning. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ICPE '13, pages 187–198, New York, NY, USA. ACM.

Hirzel, M. (2012). Partition and compose: Parallel complex event processing. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, pages 191–200, New York, NY, USA. ACM.

Holmbacka, S., Nogues, E., Pelcat, M., Lafond, S., and Lilius, J. (2014). Energy efficiency and performance management of parallel dataflow applications. In *Design and Architectures for Signal and Image Processing (DASIP), 2014 Conference on*, pages 1–8.

IBM (2005). An architectural blueprint for autonomic computing. Technical report, IBM.

IBM (2016). Ibm infosphere stream. `http://www-03.ibm.com/software/products/en/ibm-streams`.

Intel (2004). Enhanced intel speedstep technology for the intel pentium m processor.

Investopedia (2016). Analyzing chart patterns: Head and shoulders. `http://www.investopedia.com/university/charts/charts2.asp`.

Ji, Y., Zhou, H., Jerzak, Z., Nica, A., Hackenbroich, G., and Fetzer, C. (2015). Quality-driven processing of sliding window aggregates over out-of-order data streams. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, DEBS '15, pages 68–79, New York, NY, USA. ACM.

Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., and Lewin, D. (1997). Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 654–663, New York, NY, USA. ACM.

Kephart, J. and Walsh, W. (2004). An artificial intelligence perspective on autonomic computing policies. In *Policies for Distributed Systems and Networks, 2004. POLICY 2004. Proceedings. Fifth IEEE International Workshop on*, pages 3–12.

Kephart, J. O. and Chess, D. M. (2003). The vision of autonomic computing. *IEEE Computer*, 36(1):41–50.

Kim, N. S., Austin, T., Baauw, D., Mudge, T., Flautner, K., Hu, J. S., Irwin, M. J., Kandemir, M., and Narayanan, V. (2003). Leakage current: Moore's law meets static power. *Computer*, 36(12):68–75.

Kingman, J. F. C. (1962). On queues in heavy traffic. *Journal of the Royal Statistical Society. Series B (Methodological)*, 24(2):pp. 383–392.

Kulkarni, S., Bhagat, N., Fu, M., Kedigehalli, V., Kellogg, C., Mittal, S., Patel, J. M., Ramasamy, K., and Taneja, S. (2015). Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 239–250, New York, NY, USA. ACM.

Kumbhare, A., Simmhan, Y., and Prasanna, V. (2014). Plasticc: Predictive look-ahead scheduling for continuous dataflows on clouds. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 344–353.

Kusic, D. and Kandasamy, N. (2006). Risk-aware limited lookahead control for dynamic resource provisioning in enterprise computing systems. In *Autonomic Computing, 2006. ICAC '06. IEEE International Conference on*, pages 74–83.

Kusic, D., Kephart, J. O., Hanson, J. E., Kandasamy, N., and Jiang, G. (2008). Power and performance management of virtualized computing environments via lookahead control. In *Proceedings of the 2008 International Conference on Autonomic Computing*, ICAC '08, pages 3–12, Washington, DC, USA. IEEE Computer Society.

Laney, D. (2001). 3D data management: Controlling data volume, velocity, and variety. Technical report, META Group.

Leyton, M. and Piquer, J. (2010). Skandium: Multi-core programming with algorithmic skeletons. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 289–296.

Li, J., Maier, D., Tufte, K., Papadimos, V., and Tucker, P. A. (2005). No pane, no gain: Efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Rec.*, 34(1):39–44.

Li, J. and Martinez, J. (2006). Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 77–87.

Liu, H. and Parashar, M. (2006). Accord: a programming framework for autonomic applications. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 36(3):341–352.

Lohrmann, B., Janacik, P., and Kao, O. (2015). Elastic stream processing with latency guarantees. In *The 35th International Conference on Distributed Computing Systems (ICDCS 2015)*, page to appear.

Lu, H., Zhou, Y., and Haustad, J. (2013). Efficient and scalable continuous skyline monitoring in two-tier streaming settings. *Information Systems*, 38(1):68 – 81.

Maggio, M., Hoffmann, H., Papadopoulos, A. V., Panerati, J., Santambrogio, M. D., Agarwal, A., and Leva, A. (2012). Comparison of decision-making strategies for self-optimization in autonomic computing systems. *ACM Trans. Auton. Adapt. Syst.*, 7(4):36:1–36:32.

Mayer, R., Koldehofe, B., and Rothermel, K. (2015). Predictable low-latency event detection with parallel complex event processing. *Internet of Things Journal, IEEE*, 2(4):274–286.

Mencagli, G. (2015). Adaptive model predictive control of autonomic distributed parallel computations with variable horizons and switching costs. *Concurrency and Computation: Practice and Experience*, pages n/a–n/a. cpe.3495.

Mencagli, G. and Vanneschi, M. (2014). Towards a systematic approach to the dynamic adaptation of structured parallel computations using model predictive control. *Cluster Computing*, 17(4):1443–1463.

Mencagli, G., Vanneschi, M., and Vespa, E. (2013). Control-theoretic adaptation strategies for autonomic reconfigurable parallel applications on cloud environments. In *High Performance Computing and Simulation (HPCS), 2013 International Conference on*, pages 11–18.

Miyoshi, A., Lefurgy, C., Van Hensbergen, E., Rajamony, R., and Rajkumar, R. (2002). Critical power slope: Understanding the runtime effects of frequency scaling. In *Proceedings of the 16th International Conference on Supercomputing*, ICS '02, pages 35–44, New York, NY, USA. ACM.

Murthy, S., Ng, T., Avalani, B., Wang, X., Wang, K., and Gangadharan, A. (2015). Pulsar – real-time analytics at scale. Technical report, Ebay Inc.

Nasir, M. A. U., De Francisci Morales, G., Garcia-Soriano, D., Kourtellis, N., and Serafini, M. (2015). The power of both choices: Practical load balancing for distributed stream processing engines. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 137–148.

Neumeyer, L., Robbins, B., Nair, A., and Kesari, A. (2010). S4: Distributed stream computing platform. In *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*, ICDMW '10, pages 170–177, Washington, DC, USA. IEEE Computer Society.

OPRA (2016). The options price reporting authority (opra). `http://www.opradata.com/`.

Oracle (2016). Cep. `http://www.oracle.com/technetwork/middleware/complex-event-processing/overview/index.html`.

Pelagatti, S. (1993). *A methodology for the development and the support of massively parallel programs*. PhD thesis, University of Pisa.

Plagemann, T., Goebel, V., Bergamini, A., Tolu, G., Urvoy-keller, G., and Biersack, E. W. (2004). Using data stream management systems for traffic analysis - a case study. In *In Passive and Active Measurements*.

Qian, Z., He, Y., Su, C., Wu, Z., Zhu, H., Zhang, T., Zhou, L., Yu, Y., and Zhang, Z. (2013). Timestream: Reliable stream computation in the cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 1–14, New York, NY, USA. ACM.

Rahman, M., Ranjan, R., Buyya, R., and Benatallah, B. (2011). A taxonomy and survey on autonomic management of applications in grid computing environments. *Concurrency and Computation: Practice and Experience*, 23(16):1990–2019.

Sakaki, T., Okazaki, M., and Matsuo, Y. (2010). Earthquake shakes twitter users: real-time event detection by social sensors. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 851–860, New York, NY, USA. ACM.

Sankaranarayanan, J., Samet, H., Teitler, B. E., Lieberman, M. D., and Sperling, J. (2009). Twitterstand: news in tweets. In *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '09, pages 42–51, New York, NY, USA. ACM.

Scarfone, K. A. and Mell, P. M. (2007). Sp 800-94. guide to intrusion detection and prevention systems (idps). Technical report, NIST, Gaithersburg, MD, United States.

Scattolini, R. (2009). Architectures for distributed and hierarchical model predictive control - a review. *Journal of Process Control*, 19(5):723–731.

Schneider, S., Hirzel, M., Gedik, B., and Wu, K.-L. (2015). Safe data parallelism for general streaming. *Computers, IEEE Transactions on*, 64(2):504–517.

Shafik, R. A., Das, A., Yang, S., Merrett, G., and Al-Hashimi, B. M. (2015). Adaptive energy minimization of openmp parallel applications on many-core systems. In *Proceedings of the 6th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures*, PARMA-DITAM '15, pages 19–24, New York, NY, USA. ACM.

Shah, M., Hellerstein, J., Chandrasekaran, S., and Franklin, M. (2003). Flux: an adaptive partitioning operator for continuous query systems. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 25–36.

Singh, S. and Chana, I. (2015). Qos-aware autonomic resource management in cloud computing: A systematic review. *ACM Comput. Surv.*, 48(3):42:1–42:46.

Skillicorn, D. B. and Talia, D. (1998). Models and languages for parallel computation. *ACM Comput. Surv.*, 30(2):123–169.

Stonebraker, M., Çetintemel, U., and Zdonik, S. (2005). The 8 requirements of real-time stream processing. *SIGMOD Rec.*, 34:42–47.

Stonebraker, M., Cherniack, M., Çetintemel, U., Balazinska, M., and Balakrishnan, H. (2003). The Aurora and Medusa Projects. *IEEE Data(base) Engineering Bulletin*, 26:3–10.

Sun, D., Zhang, G., Yang, S., Zheng, W., Khan, S. U., and Li, K. (2015). Re-stream: Real-time and energy-efficient resource scheduling in big data stream computing environments. *Information Sciences*, 319:92 – 112.

Tangwongsan, K., Hirzel, M., Schneider, S., and Wu, K.-L. (2015). General incremental sliding-window aggregation. *Proc. VLDB Endow.*, 8(7):702–713.

Tao, Y. and Papadias, D. (2006). Maintaining sliding window skylines on data streams. *IEEE Transactions on Knowledge and Data Engineering*, 18(3):377–391.

Tatbul, N., Çetintemel, U., Zdonik, S., Cherniack, M., and Stonebraker, M. (2003). Load shedding in a data stream manager. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 309–320. VLDB Endowment.

Teubner, J. and Mueller, R. (2011). How soccer players would do stream joins. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 625–636, New York, NY, USA. ACM.

Tibco (2016). Tibco streambase cep. `http://www.tibco.com/products/event-processing/complex-event-processing/streambase-complex-event-processing`.

Urbani, J., Margara, A., Jacobs, C., Voulgaris, S., and Bal, H. (2014). Ajira: A lightweight distributed middleware for mapreduce and stream processing. In *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, pages 545–554.

Vanneschi, M. (2002). The programming model of assist, an environment for parallel and distributed portable applications. *Parallel Computing*, 28(12):1709 – 1732.

Vanneschi, M. (2014). *High Performance Computing - Parallel Processing Model and Architectures*. Pisa University Press.

Vanneschi, M. and Veraldi, L. (2007). Dynamicity in distributed applications: issues, problems and the assist approach. *Parallel Computing*, 33(12):822 – 845.

Vazirani, V. V. (2001). *Approximation Algorithms*. Springer-Verlag New York, Inc., New York, NY, USA.

Weiler, A., Grossniklaus, M., and Scholl, M. H. (2016). An evaluation of the runtime and task-based performance of event detection techniques for twitter. *Information Systems*.

Wu, E., Diao, Y., and Rizvi, S. (2006). High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 407–418, New York, NY, USA. ACM.

Wu, S., Kumar, V., Wu, K.-L., and Ooi, B. C. (2012). Parallelizing stateful operators in a distributed stream processing system: How, should you and how much? In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, pages 278–289, New York, NY, USA. ACM.

Wu, Y. and Tan, K.-L. (2015). Chronostream: Elastic stateful stream computation in the cloud. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 723–734.

Wuttke, J. (2015). Lmfit – a c library for levenberg-marquardt least-squares minimization and curve fitting.

Yu, F., Chen, Z., Diao, Y., Lakshman, T. V., and Katz, R. H. (2006). Fast and memory-efficient regular expression matching for deep packet inspection. In *Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, ANCS '06, pages 93–102, New York, NY, USA. ACM.

Yuan, Q., Liu, Z., Peng, J., Wu, X., Li, J., Han, F., Li, Q., Zhang, W., Fan, X., and Kong, S. (2011). *Advances in Grid and Pervasive Computing: 6th International Conference, GPC 2011, Oulu, Finland, May 11-13, 2011. Proceedings*, chapter A Leasing Instances Based Billing Model for Cloud Computing, pages 33–41. Springer Berlin Heidelberg, Berlin, Heidelberg.

Zaharia, M., Das, T., Li, H., Shenker, S., and Stoica, I. (2012). Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, pages 10–10, Berkeley, CA, USA. USENIX Association.

Zhao, S., Chandrashekar, M., Lee, Y., and Medhi, D. (2015). Real-time network anomaly detection system using machine learning. In *11th International Conference*

on the Design of Reliable Communication Networks, DRCN 2015, Kansas City, MO, USA, March 24-27, 2015, pages 267–270.