

Parallel Patterns for Window-based Stateful Operators on Data Streams: an Algorithmic Skeleton Approach

Tiziano De Matteis · Gabriele Mencagli

Received: date / Accepted: date

Abstract The topic of *Data Stream Processing* is a recent and highly active research area dealing with the in-memory, tuple-by-tuple analysis of streaming data. *Continuous queries* typically consume huge volumes of data received at a great velocity. Solutions that persistently store all the input tuples and then perform off-line computation are impractical. Rather, queries must be executed continuously as data cross the streams. The goal of this paper is to present parallel patterns for *window-based stateful operators*, which are the most representative class of stateful data stream operators. Parallel patterns are presented “à la” *Algorithmic Skeleton*, by explaining the rationale of each pattern, the preconditions to safely apply it, and the outcome in terms of throughput, latency and memory consumption. The patterns have been implemented in the **FastFlow** framework targeting off-the-shelf multicores. To the best of our knowledge this is the first time that a similar effort to merge the Data Stream Processing domain and the field of Structured Parallelism has been made.

Keywords Parallel Patterns · Algorithmic Skeletons · Data Stream Processing · Multi-/Many-core Architectures

1 Introduction

The recent years have been characterized by an explosion of *data streams* generated by a variety of sources like social media, sensors, network and mobile devices, stock tickers, and many others. Technologies to extract greater value from the data have proliferated accordingly, by translating information into

T. De Matteis and G. Mencagli
Department of Computer Science, University of Pisa
Largo B. Pontecorvo 3, I-56127, Pisa, Italy
Phone: +39-050-221-3132
E-mail: dematteis@di.unipi.it, mencagli@di.unipi.it

decisions in real-time. Traditional store-and-process data management architectures are not capable of offering the combination of *latency* and *throughput* requirements needed by the new applications. Even map-reduce frameworks like Hadoop, in their original form, still do not provide the needed answers [13].

More adequate models have been proposed both in academic and commercial frameworks [3,2,1]. *Data Stream Processing* [9] (briefly, DaSP) is the paradigm enabling new ways to deal with and work with the streaming data. Data are seen as transient continuous streams rather than being modeled as traditional permanent, in-memory data structures or relations. Streams convey single *elements* or *segments* belonging to the same logical data structure (e.g. relational tables). The defined computation is a *continuous query* [8,9] whose *operators* must be executed “on the fly”. The goal is usually to extract actionable intelligence from raw data, and to react to operational exceptions through real-time alerts and automated actions in order to correct/solve problems.

We claim that in the DaSP literature there is a use of incoherent terminology in explaining *intra-operator parallelism* for continuous queries, which in some cases contrasts with the classic view of Algorithmic Skeletons [16]. Just to clarify this sentence, in the DaSP domain *data parallelism* is exemplified by splitting the input stream into multiple outbound streams routed to different replicas of the same operator that work on distinct input elements in parallel [17]. For researchers expert in algorithmic skeletons [16,12], this pattern is the classic *farm* skeleton belonging to the *task parallelism* paradigm.

The goal of this paper is to show how the parallelization issues of DaSP computations can be dealt with the algorithmic skeleton methodology based on the usage and composition of a limited set of parallel patterns. W.r.t the traditional patterns, the DaSP domain requires proper specializations and enhanced features in terms of *data distribution* and *management* policies and *windowing* methods. The contributions of this paper are the following:

- the identification of the features of parallel patterns in relation to the distribution policy, the presence of an internal state, and the role of parallel executors in the window management;
- the description of four patterns for window-based stateful operators that are the most representative class of stateful operators for continuous queries;
- our parallel patterns are implemented in the **FastFlow** framework [4] for high-level pattern-based parallel programming on multicores.

The paper is organized as follows. Sect. 2 briefly recalls the fundamental features of DaSP. Sect. 3 presents the parallel patterns which will be experimentally evaluated in Sect. 4. Sect. 5 summarizes related works and their binding with our work. Finally, Sect. 6 concludes this paper.

2 Data Stream Processing

In the last years data management architectures known as Data Stream Management Systems [13] (DSMSs) have been proposed as a response to the increasing volume and velocity of streaming data. Extensions of DaSP have been

developed in specific application domains. For example *Complex Event Processing Systems* [13] (CEPSs) focus on the detection of complex events through pattern matching algorithms applied on transient data. In this paper we use the general term *Stream Processing Engine* (SPE) to refer to the recent stream processing frameworks. A review of them is presented in Sect. 5.

Modern SPEs allow the programmer to express applications as compositions of core functionalities in directed flow graphs [13], where vertices are operators and arcs model streams, i.e. unbounded sequences of data items (*tuples*) sharing the same schema in terms of name and type of attributes. Flow graphs represent *continuous queries* [13], i.e. standing queries that run continuously on transient data. In the sequel, we discuss the nature of the operators and the parallelism recognized in the execution of continuous queries.

2.1 Operators and Windowing Mechanisms

The internal processing logic of an operator consumes input data and applies transformations on them. The number of output tuples produced per tuples consumed is called *selectivity* of the operator.

The nature of operators is wide and varied. DSMSs provide relational algebra operators such as *map*, *filters*, *aggregates* (sum, count, max), *sort*, *joins* and many others. Recently, they have been enhanced with preference operators like *skyline*, *top-k* and operators employing data mining and machine learning techniques [21]. In CEPSs the focus is on pattern-matching operators programmed by a set of reactive rules “à la” active databases [13]. Finally, the most recent SPEs like Storm [2] and IBM InfoSphere [17] support customizable operators implemented in general-purpose programming languages.

According to the literature [9] we distinguish between:

- *stateless operators* (e.g. selection, projection and filtering) work on a item-by-item basis without maintaining data structures created as a result of the computation on earlier data;
- *stateful operators* maintain and update 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, grouping, intersection as so forth.

In many application cases, the physical input stream conveys tuples belonging to multiple *logical substreams* multiplexed together. Stateful operators can require to maintain a separated data structure for each substream, and to update it on the basis of the computation applied only to the tuples belonging to that substream. The correspondence between tuples and substreams is usually made by applying a predicate on a partitioning attribute called *key*, e.g. the distinction is the result of a *partition-by* predicate. Examples are operators that process network traces partitioned by IP address, or trades and quotes from financial markets partitioned by stock symbol. We refer to the case of a partitionable state as a *multi-keyed stateful operator* with *cardinality* $|\mathcal{K}| > 1$. The case $|\mathcal{K}| = 1$ refers to the special case of a *single-keyed stateful operator*.

On data streams the semantics of stateful operators requires special attention. Due to the unbounded nature of the stream, it is not possible to apply the computation over the entire stream history. The literature suggests two possible solutions:

- the state can be implemented by succinct data structures such as *synopses*, *sketches*, *histograms* and *wavelets* [5] used to maintain aggregate statistics of the tuples received so far;
- in applications like pattern detection and time-series analysis the internal processing logic must inspect the input tuples that need to be maintained as a whole in the internal state. Fortunately, in realistic applications the significance of each tuple is time-decaying, and it is often possible to buffer only a subset of the input data by obtaining well approximate results with limited memory occupancy. A solution consists in implementing the state as a *window* buffer [9] in which only the most recent tuples are kept.

Windows are the predominant abstraction used to implement the internal state of operators in DaSP. The window semantics is specified by the *eviction policy*, i.e. when data in the window can be safely removed, and the *triggering policy*, i.e. when the internal logic can be applied on the actual content of the window buffer. Two parameters are important:

- the *window size* $|\mathcal{W}|$ is expressed in time units (seconds, minutes, hours) in *time-based windows* or in number of tuples in *count-based windows*;
- the *sliding factor* δ expresses how the window moves and its content gets processed by operator’s algorithm. Analogously to the window size, the sliding factor is expressed in time units or in number of tuples.

A crucial aspect for the discussion of Sect. 3 is that consecutive windows may have overlapping regions. This situation is true for *sliding windows* in which $\delta < |\mathcal{W}|$. However, a window at a given time instant contains also tuples not belonging to the preceding windows. In general, the window-based processing is *non-monotonic* [20], since the results can not be produced incrementally due to the expiration of old tuples exiting from the current window. Cases of *tumbling windows* (disjoint, i.e. $\delta = |\mathcal{W}|$) and *hopping windows* ($\delta > |\mathcal{W}|$) are studied in the literature for specific problems [9], but in general are less common than sliding windows.

2.2 Optimizations, Parallelism and our Vision

From the performance standpoint, SPEs are aimed at executing continuous queries submitted by the users in such a way as to maximize *throughput* (output rate), i.e. the speed at which results are delivered to the users, and minimize *latency* (response time), i.e. the time elapsed from the reception of a tuple triggering a new activation of query’s internal processing logic and the production of the first corresponding output.

To this end, parallelism has become an unavoidable opportunity to speedup the query execution by relying on underlying parallel architectures such as

multi/manycorers or clusters of multicorers. Parallelism in existing SPEs is expressed at two different levels:

- *inter-query parallelism* consists in supporting the execution of multiple flow graphs in parallel. It is used to increase the overall throughput;
- *intra-query parallelism* makes it possible to increase throughput and to reduce the response time. It is further classified into *inter-operator* parallelism, by exploiting the inherent parallelism between operators that run concurrently, and *intra-operator parallelism*, in which a single operator instance (generally a hotspot) can be internally parallelized if necessary.

We claim that *a methodology for intra-operator parallelism is still lacking*. As it will be described in Sect. 5, 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 and assigning input tuples to the replicas in a load balanced fashion. For multi-keyed stateful operators the parallel solution consists in using replicas each one working on a subset of the keys. Although recurrent [15] these two approaches are far from being exhaustive of all the possible parallel solutions.

We advocate that the algorithmic skeleton methodology [12, 16] is particularly suitable to be integrated in SPE environments. It is a solid methodology to develop parallel applications as compositions, interweaving and nesting of reusable parallel patterns parameterized by the programmer to generate specific parallel programs. Besides being a methodology to reduce the effort and complexity of parallel programming, algorithmic skeletons simplify the reasoning about the properties of a parallel solution in terms of throughput, latency and memory occupancy [11]. *Exactly what is needed by intra-operator parallelism in continuous queries*.

In the next section we get into the core of the discussion. We tackle the problem of operators working with windows that, as we have seen, are the most widely used abstraction to model the concept of internal state in DaSP.

3 Parallel Patterns

In the ensuing discussion we assume a generic window-based stateful operator working on a single input 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 available from *any* streams. With more than one output stream the results can be transmitted to one of them, selected according to a given predicate on the results' attributes, or to all the output streams through a multicast.

A window is a very special case of internal state, consisting in a subsequence of input tuples. In the context of window-based computations task parallelism assumes a special characterization. A task is not a single input element as in the traditional stream processing. Rather, a task is now a segment of the input stream corresponding to *all* the tuples belonging to the same window. According to this interpretation of task, we distinguish between:

- *window parallel* patterns are skeletons capable of executing in parallel multiple windows at the same time instant;
- *data parallel* patterns are skeletons in which the execution of each single window is parallelized by partitioning it among a set of identical executors.

All the relevant characteristics of a skeleton can be easily derived from its definition and structure: throughput, latency, utilization of resources and memory consumption. Due to the dynamic nature of the windows, the distribution phase is particularly important. We focus on two orthogonal aspects:

- the *granularity* at which input elements are distributed by an *Emitter* functionality to a set of parallel executors called *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 the 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 workers*: the executors are just in charge of applying the computation to the received data. All the processing/storing actions needed to build and to update the windows are performed by the distribution logic;
- *active workers*: the window management is partially or entirely delegated to the workers that receive elementary stream items from the emitter and manage the window boundaries by adding/removing tuples.

Tab. 1 shows the features of the patterns in terms of parallelism paradigms, window management and if they can be used in single-keyed or multi-keyed scenarios. Each pattern will be presented in a section by itself including different parts: a figure giving a quick intuition of its behavior, *applicability* lists the conditions to apply the pattern, *profitability* shows the advantages derived from the pattern usage, *issues* describes the drawbacks, and *variations* lists possible optimizations. The patterns will be exemplified in the case of count-based windows but they can be easily adapted to time-based windows.

Name	Paradigm	Single Keyed	Multi Keyed	Win. Man.
Window Farming	Window Parallelism	✓	✓	Agnostic/* Active
Key Partitioning	Window Parallelism	✗	✓	Agnostic/* Active
Pane Farming	Window Parallelism	✓	✓	Agnostic
Windows Partitioning	Data Parallelism	✓	✓	Active

Table 1: Parallel patterns. *the distinction between agnostic/active workers depends on the granularity of the distribution and the assignment policy performed by the emitter.

In this paper the patterns will be presented by abstracting the the target architecture, i.e. they can be instantiated both on shared-memory machines and on distributed-memory architectures provided that proper thread/process cooperation mechanisms are used. For the experiments the patterns have been implemented on multi-core architectures only, using the `FastFlow` runtime.

3.1 Window Farming

The first pattern exploits a simple intuition. Each activation of the computation (let say a function \mathcal{F}) is applied to a subset of the input stream called window. 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. Therefore, a simple solution is to adapt the classic *farm* skeleton to this domain, as sketched in Fig. 1.

The emitter is in charge of buffering tuples coming from the stream and building and updating the *canonical copy* of the windows, one for each logical substream. In the figure we show an example of two keys X and Y . Tuples and windows are marked with unique identifiers. Once a window has been completely buffered, it is transmitted (copied) to a worker. The assignment is aimed at balancing the workload, e.g. the *round-robin* policy can be adopted in the case of an internal function with low variance processing time depending on the data values, otherwise an *on-demand* assignment can be a better solution. In the figure we adopt a round-robin strategy: windows ω_i^x and ω_i^y are assigned to worker j s.t. $j = (i \bmod n) + 1$ where n is the number of workers.

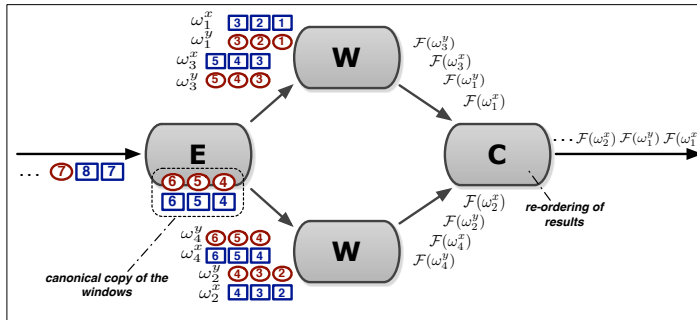
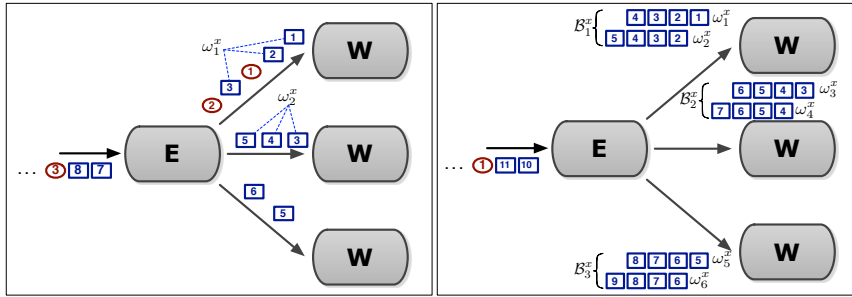


Fig. 1: Window farming with two workers, two substreams X (square) and Y (circle) and $|W|=3$ and $\delta=1$. ω_i^x the i -th window of substream X . \mathcal{F} is the processing function.

It is worth noting that multiple windows (of the same and of different substreams) are executed in parallel by different workers that are completely *agnostic* of the window management. Workers receive a bulk of data (three tuples per task in the example), apply the function \mathcal{F} , and discard the data. The emitter is responsible for receiving new tuples and removing expired ones

according to the window size and the sliding factor. A *Collector* functionality receives the results and may be responsible for reordering them. For brevity we denote window farming as follows: $\text{W-Farm}(\mathcal{F}, |\mathcal{W}|, \delta)$. An application of this pattern in the field of system biology has been implemented using **FastFlow** in [6], however without some of the optimizations discussed in this section.

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. Each tuple is routed to one or more workers depending on the values of the window size and slide parameters and the assignment policy of windows to workers. Fig. 2a shows an example with three workers, window size $|\mathcal{W}|=3$ and slide $\delta=2$. The distribution is performed on-the-fly by transmitting a tuple at a time.



(a) Fine-grained distribution ($|\mathcal{W}|=3, \delta=2$). (b) Batching ($|\mathcal{W}|=4, \delta=1, |\mathcal{B}|=2$).

Fig. 2: Optimizations of the window farming pattern (example with three workers).

Windows are assigned to the three workers in a round-robin fashion. Each tuple can be transmitted to *one* or *more* workers. 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 to windows ω_2^x and ω_3^x and thus it is multicasted to the second and the third worker. The window logic is still predominant in the emitter:

- it assigns windows to the workers;
- for each received tuple t , it determines the identifiers of the windows on which this tuple belongs. The tuple is transmitted to the worker j if there exists a window ω_i s.t. $t \in \omega_i$ and ω_i is assigned to that worker;
- when the last tuple of a window is received, it is marked with a special metadata to make aware the worker assigned to that window that the evaluation of \mathcal{F} can be triggered.

With this distribution the workers remove the tuples that are no longer needed, thus they become partially *active* in the window logic. Each worker removes the oldest $n\delta$ tuples before starting the computation on a new window.

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. Furthermore, fine-grained distributions can be useful to reduce latency and improve throughput if \mathcal{F} is *incrementally computable* (e.g. distributive or algebraic aggregates). For example a worker performs the following steps: `while (EOW) { receive t; s.update(t); s.output(); }`, where EOW denotes the end of a window and s is a data structure maintained incrementally. The workers can start the computation as new tuples arrive without needing to have the whole window. Complex statistics and processing like interpolation, regression or sorting may need the entire window in the worst case, and do not benefit from this optimization.

It is important to observe that tuples are replicated¹ in the workers due to the fact that consecutive windows overlap. A further optimization consists in assigning *batches* to workers, i.e. a set of $\mathcal{B} \geq 1$ consecutive windows of the same substream. A tuple present in more than one window in the same batch is transmitted just one time to the corresponding worker. The goal is to reduce data replication. Fig. 2b 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 the standard assignment of single windows. Batching can increase latency and the buffering space in the emitter. This can still be mitigated by using fine-grained distributions. Tab. 2 summarizes the main features of window farming.

Window Farming	
Applicability	The pattern can be applied to any window-based stateful operator. No particular property is required for the function \mathcal{F} .
Profitability	The pattern is able to optimize <i>throughput</i> by using a number of workers such that the stream speed can be sustained without being bottleneck. <i>Load balancing</i> can be easily obtained through proper assignment policies of windows to workers.
Issues	The emitter manages the window logic and may be bottleneck for very fine-grained \mathcal{F} . Based on the sliding factor, the same tuple can be replicated in several workers by potentially increasing the overall <i>memory consumption</i> . Results may arrive to the collector potentially <i>unordered</i> . This pattern does not optimize <i>latency</i> since the function \mathcal{F} is applied serially on each window.
Variations	Distributions with finer granularity (<i>on-the-fly</i>) reduce the buffering space in the emitter and improve its service time. This can be also profitable to <i>reduce latency</i> and <i>improve throughput</i> if the function \mathcal{F} is <i>incrementally computable</i> . <i>Batch-based</i> assignments can be used to reduce data replication.

Table 2: Summary of the features and properties of window farming.

¹ replicated in a hypothetical message-passing abstract model. On multicores, based on the used run-time support, tuples replication can be avoided by sharing data, i.e. by passing memory pointers to the input tuples.

3.2 Key Partitioning

Key partitioning is a variant of window farming with a constrained assignment policy. This pattern expresses a limited parallelism: only the windows belonging to different substreams can be executed in parallel, while the windows of the same substream are processed serially by the same worker. The idea is to split the set of keys \mathcal{K} into n partitions, where n is the number of workers. An emitter assigns windows to the workers based on the value of the key attribute. If windows are distributed as a whole, workers are *agnostic* of the window management. With fine-grained distributions the workers become *active*, i.e they manage the window boundaries for the assigned keys. We denote the pattern as $\text{KP}(\mathcal{F}, \mathcal{K}, |\mathcal{W}|, \delta)$. This variant deserves to be considered a pattern *per se* due to its wide diffusion in the literature [15].

Intuitively, load balancing becomes a problem when there is skew in the distribution of the keys. If p^{max} denotes the highest frequency of a key, the parallel pattern can scale up to $1/p^{max}$. Only with a uniform distribution of keys the maximum scalability is equal to the number of distinct keys $|\mathcal{K}|$, which is also the highest parallelism degree exploitable by this pattern. Tab. 3 summarizes the pros and cons of key partitioning.

Key Partitioning	
Applicability	The pattern can be applied to any <i>multi-keyed</i> stateful operator (based on windows or not). Multiple logical substreams must exist.
Profitability	The pattern improves <i>throughput</i> . Windows are assigned to workers such that no <i>data replication</i> is necessary. Results with the same key arrive to the collector <i>ordered</i> .
Issues	<i>Load balancing</i> can be difficult or even impossible if the key distribution is very skewed. The pattern does not optimize <i>latency</i> .
Variations	As for window farming, distributions with finer granularity (<i>on-the-fly</i>) reduce the buffering space in the emitter and improve its service time. This can be also profitable to <i>reduce latency</i> and <i>improve throughput</i> if the function \mathcal{F} is <i>incrementally computable</i> .

Table 3: Summary of the features and properties of key partitioning.

This pattern is useful also when the data structure for each substream is not a window. An example are *synopsis-based operators*, where a synopsis is used to make statistics over the the entire history of a substream. In that 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.

3.3 Pane Farming

The pane-based approach has been proposed for the centralized processing of sliding window aggregates in [18]. Its generalization identifies a parallel pat-

tern with interesting properties in terms of throughput, latency and memory occupancy. The idea is to divide each window into non-overlapping contiguous partitions called *panes* of size $\sigma_p = \text{gcd}(|\mathcal{W}|, \delta)$. Each window ω is composed of r panes $\omega = \{\mathcal{P}_1, \dots, \mathcal{P}_r\}$ with $r = |\mathcal{W}|/\sigma_p$. This pattern can be applied if the internal processing function \mathcal{F} can be decomposed into two functions \mathcal{G} and \mathcal{H} used as follows: $\mathcal{F}(\omega) = \mathcal{H}(\mathcal{G}(\mathcal{P}_1), \dots, \mathcal{G}(\mathcal{P}_r))$, i.e. \mathcal{G} is applied to each pane and \mathcal{H} is computed by combining the 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), complex *differential* and *pseudo-differential* aggregates, and many others [18, 10]. The idea of this pattern is sketched in Fig. 3 exemplified in a single-keyed scenario.

The application of \mathcal{G} and \mathcal{H} can be viewed as a two-staged pipeline where each stage can be parallelized using window farming. Panes are *tumbling sub-windows* distributed to a set of *agnostic* workers applying the function \mathcal{G} on each received pane. Panes execution can be assigned in a round-robin fashion or using an on-demand policy to achieve load balancing. The second stage receives pane results and applies the function \mathcal{H} on windows of pane results with 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 defined as $\text{Pipe}(\text{W-Farm}(\mathcal{G}, \sigma_p, \sigma_p), \text{W-Farm}(\mathcal{H}, r, \delta_p))$. In multi-keyed scenarios key partitioning can be applied in the second stage too. We can observe that, belonging to the window parallel paradigm, more than one window is executed in parallel in the workers of the pane farming pattern.

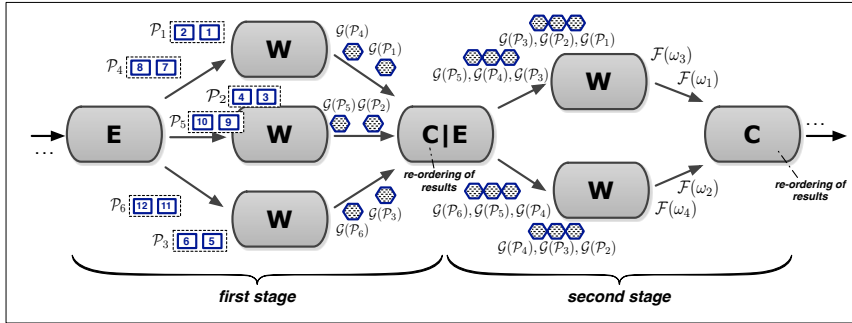


Fig. 3: Pane farming with one key. Window with $|\mathcal{W}|=6$, $\delta=2$, $\sigma_p=2$. The collector of the first stage and the emitter of the second one have been merged in a single C|E functionality.

In the figure the first stage has been parallelized with three workers while the second with two. For fine-grained \mathcal{H} the second stage can be sequential, i.e. $\text{Pipe}(\text{W-Farm}(\mathcal{G}, \sigma_p, \sigma_p), \text{Seq}(\mathcal{H}, r, \delta_p))$, and eventually collapsed in C/E.

Pane farming improves throughput. It also reduces latency by sharing overlapping pane results between consecutive windows. The latency reduction factor is given by the ratio $\mathcal{L}^{seq}/\mathcal{L}^{pane}$, where $\mathcal{L}^{seq} \sim rT_{\mathcal{G}} + T_{\mathcal{H}}$ is the latency of the sequential version and $\mathcal{L}^{pane} \sim T_{\mathcal{G}} + T_{\mathcal{H}}$ the one with the pane ap-

proach ($T_{\mathcal{H}}$ and $T_{\mathcal{G}}$ are the processing times of the two functions). The ratio approaches r as $T_{\mathcal{H}} \rightarrow 0$. Pane farming can also be used with multiple keys: panes of different substreams are dispatched by **E** to the workers of the first stage, while the corresponding windows are calculated in the second one.

The functionality **C/E** can be critical for latency and throughput. It merges the pane results coming from the workers of the first stage and assigns windows of pane results to the workers of the second stage. *Shuffling* can be used to remove this potential bottleneck. Rather than merging back 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 Fig. 4a.

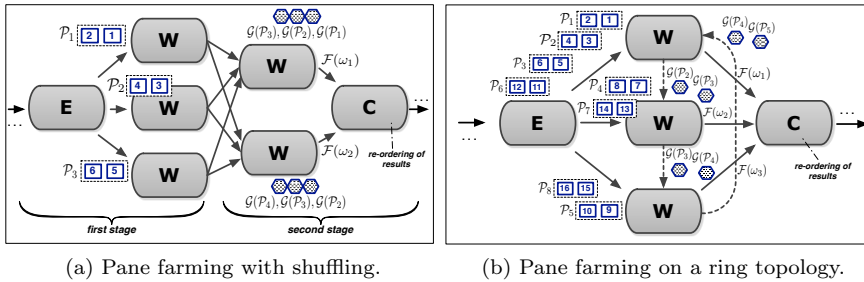


Fig. 4: Variations of the pane farming pattern.

Alternatively, the two stages can be merged by organizing the workers on a *ring* topology as suggested in [10] (see Fig. 4b). Workers now 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 worker on the ring. As explained in [10], 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. However, this static assignment can prevent load balancing if \mathcal{G} has a high variance processing time. Although elegant and interesting, this variant can not always be applicable in algorithmic skeleton frameworks [16], where it is not always possible to express explicit communications between workers by using predefined skeletons. Tab. 4 summarizes the properties of pane farming.

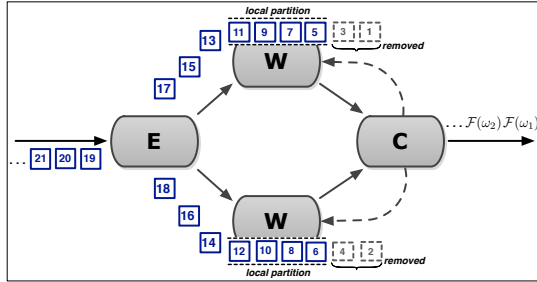
3.4 Window Partitioning

Window partitioning is an adaptation of the *map-reduce* skeleton on data streams. The current window is partitioned among n workers responsible for computing the internal processing function on their partitions. A *reduce* phase can be necessary to aggregate/combine the final results of the window. According to the data parallelism paradigm, exactly *one window at a time is in execution* on the parallel pattern. The emitter can buffer and scatter each window as a whole or, more conveniently, single tuples are distributed to the workers

Pane Farming	
Applicability	The pattern can be applied if the function \mathcal{F} can be expressed as $\mathcal{F}(\omega) = \mathcal{H}(\mathcal{G}(\mathcal{P}_1), \dots, \mathcal{G}(\mathcal{P}_n))$ on each window ω .
Profitability	We have no <i>data replication</i> in the first stage because panes are disjoint (<i>batching</i> can be applied in the second stage). The pattern improves <i>throughput</i> and <i>latency</i> . <i>Load balancing</i> can be easily achieved through proper assignment policies of panes to workers.
Issues	The functionalities E and C/E are aware of the decomposition in panes and how pane results must be combined to obtain the final results of the windows. This pattern is not useful if the sliding factor δ is equal to one tuple, called <i>slide-by-tuple</i> queries. E.g. “return the max price in the last 100 trades”.
Variations	<i>Shuffling</i> can be adopted to remove the C/E functionality. This can be useful if C/E is bottleneck or it is hard to implement. Shuffling comes at the expense of additional communication cost in the workers of the first stage. The <i>ring</i> variant makes it possible to merge the two stages into a unique $[\mathbb{E}, \{\mathbb{W}\}, \mathbb{C}]$ structure if communications between workers can be expressed.

Table 4: Summary of the features and properties of pane farming.

one at a time. In this case the workers 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. An example of this pattern is shown in Fig. 5. In the example the 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. Otherwise, other distributions preserving the contiguity of data can be used if necessary. In a multi-keyed scenario the workers maintain a window partition per logical substream.

Fig. 5: Window partitioning with two workers and one key. $|\mathcal{W}|=8$ 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 (let say \mathcal{F}) on the partitions. The workers

execute the local reduce phase on their partitions and, in the implementation depicted in Fig. 5, they communicate the local reduce results to the collector which is in charge of computing the global reduce result. Depending on the computation semantics the reduce phase (with an associative operator \oplus_r) can be performed in two ways:

- *asynchronously* w.r.t the computation of the workers. In this case the global reduce result is forwarded to the output stream and it 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 and \oplus_r is the `sum`;
- *synchronously* w.r.t the computation of the workers, which need explicitly to receive the global reduce result from the collector through the dashed arrows in Fig. 5. Usually, the global reduce result needs to be multicasted to all the workers for reasons related to the computation semantics, e.g. a second map(-reduce) phase must be executed as in the query “finding the tuples in the last 1000 tuples such that the price attribute is higher than the average price in the window”.

In general the behavior of the pattern captures a computation defined as follows: $\text{Loop}_{v_i}(\text{Map}(\mathcal{F}^i, |\mathcal{W}|, \delta), \text{Reduce}(\oplus_r^i))$, i.e. one or more map-reduce computations applied repeatedly on the window data. 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 as well.

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. Tab. 5 summarizes the pros and cons of this pattern.

Window Partitioning	
Applicability	The pattern can be applied when the computation is expressed as a <i>map</i> followed by an optional <i>reduce</i> phase. The repetitive application of maps and reduces can be captured as well.
Profitability	The pattern improves <i>throughput</i> and optimizes <i>latency</i> . Tuples are partitioned perfectly without <i>data replication</i> .
Issues	<i>Load balancing</i> can be difficult to achieve if the internal processing function has a high variance processing time depending on the data values. Workers are fully active and need to know when the activation of the internal processing must be triggered.
Variations	Distribution can be performed with finer granularity (<i>on-the-fly</i>) by reducing the buffering space in the emitter and its service time. This can be also profitable to further <i>reduce latency</i> and <i>improve throughput</i> if the function \mathcal{F} is <i>incrementally computable</i> .

Table 5: Summary of the features and properties of window partitioning.

3.5 Nesting of Patterns

As for the classic algorithmic skeletons, nesting can be considered a potential solution to balance the pros and cons of the different patterns. Tab. 6 shows two notable examples of nesting. A more detailed analysis of the possible schemes and their advantages will be studied in our future work.

Notable cases of Pattern Nesting	
Outer-level: Window Farming Inner-level: Window Part.	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 <i>latency</i> than using window farming alone. Other properties are: <i>i</i>) it exploits the easier <i>load balancing</i> at the outermost level; <i>ii</i>) <i>batching</i> can be used at the outermost level to reduce <i>data replication</i> between macroworkers; <i>iii</i>) no data replication inside macroworkers; <i>iv</i>) the distribution (at both the levels) can be performed with finer granularity (<i>on-the-fly</i>).
Outer-level: Key Partitioning Inner-level: Window Part.	Key partitioning with macroworkers implemented using window partitioning. This solution can be applied in <i>multi-keyed</i> scenarios only. It increases throughput with lower <i>latency</i> and <i>optimal memory occupancy</i> (no tuple replication at any level). <i>Load balancing</i> can be hard as it is critical for both the parallel patterns.

Table 6: Notable nesting cases of parallel patterns for window-based stateful operators.

4 Experiments

This section describes a first evaluation of the patterns on shared-memory machines. We leave to our future work the analysis on shared-nothing architectures. The parallel patterns have been implemented in **FastFlow** [4], a C++ framework for skeleton-based parallel programming. Its design principle is to provide *high-level parallel patterns* to the programmers, implemented on top of *core skeletons* (*pipeline* and *farm*) easily composable in cyclic networks. **FastFlow** natively supports streaming contexts, thus it is a friendly environment on which provide a first implementation of our patterns.

As a *proof-of-concept* we have implemented our patterns in the **FastFlow** runtime, by directly modifying the core skeletons with the required distribution, collecting and windowing functionalities. Parallel entities (emitters, workers and collectors) have been implemented as **pthread**s with fixed affinity on the thread contexts of general-purpose multicores. According to the **FastFlow** model, the interactions between threads consist in **pop** and **push** operations on non-blocking lock-free shared queues [7].

The code of the experiments has been compiled with the **gcc** compiler (version 4.8.1) and the `-O3` optimization flag. The target architecture is a dual-CPU Intel Sandy-Bridge multicore with 16 hyperthreaded cores operating at 2GHz with 32GB of RAM. Each core has a private L1d (32KB) and L2

(256KB) cache. Each CPU is equipped with a shared L3 cache of 20MB. In the experiments threads have been fixed on the cores. The computation is interfaced with a *Generator* and *Consumer* thread, both executed on the same machine, through TCP/IP standard POSIX sockets. Thus, 12 is the max number of workers without hyperthreading.

4.1 Synthetic Benchmarks

The benchmark computes a suite of complex statistical aggregates used for algo trading [14], with $|\mathcal{K}| = 1000$ stock symbols and count-based windows of 1000 tuples and slide of 200 tuples. Each tuple is a *quote* from the market and is stored in a record of 64 bytes. To use pane farming (PF) the synthesized computation is composed of a function \mathcal{G} with $T_{\mathcal{G}} \approx 1500\mu\text{sec}$, and a function \mathcal{H} with $T_{\mathcal{H}} \approx 20\mu\text{sec}$ executed by the collector. Each test ran for 180 seconds.

For each parallelism degree we determine the highest input rate (throughput) sustainable by the system. To detect it, we repeat the experiments several times with growing input rates. The generator checks the TCP buffer of the socket. If it is full for enough time, it stops the computation and the last sustained rate is recorded. Figs. 6a, 6b and 6c show the results in three scenarios. In the first one the keys probabilities are uniformly distributed ($p = 10^{-3}$). In the second one we use a *skewed* distribution with the most frequent key with $p^{max} = 3\%$. The last case is a *very skewed* distribution with $p^{max} = 16\%$. We also report the results with two worker threads per core (24), which is the best hyperthreaded configuration found in our experimental setting. In the first two scenarios key partitioning (KP) provides slightly higher throughput than window farming (WF) and window partitioning (WP) due to better reuse of window data in cache. WF and WP are comparable. The best results are achieved with PF: throughput is 5 times higher than the other patterns. The reason is that PF uses a faster sequential algorithm that shares the pane results in common between consecutive windows of the same key. Interesting is the very skewed distribution (Fig. 6c) where the rate with KP stops to increase with more than 6 workers. In fact, with $p^{max} = 16\%$ the scalability is limited to 6.25.

Fig. 6d shows the average latency for each elapsed second of execution in a scenario with input rate of 200K tuples/sec. To not being bottleneck we use 10 workers for WF, KP and WP and two workers with PF. The latency is plotted using two logarithmic scales: the one on the left is used for WP and PF, the scale on the right for WF and KP. As expected WF and KP have similar latencies because each window is processed serially. PF has latency 5 times lower than WF and KP. As discussed in Sect. 3.3, the latency reduction factor is roughly equal to the number of panes per window if $T_{\mathcal{H}} \sim 0$ as in this benchmark. In contrast WP produces a latency reduction proportional to the parallelism degree (partition size). With 10 workers the latency is 27.53% lower than PF. Tab. 7 summarizes the best scalability results. In summary the achieved performance is good, owing to the efficient **FastFlow** runtime and the data reuse in cache enabled by the sliding-window nature of the computation.

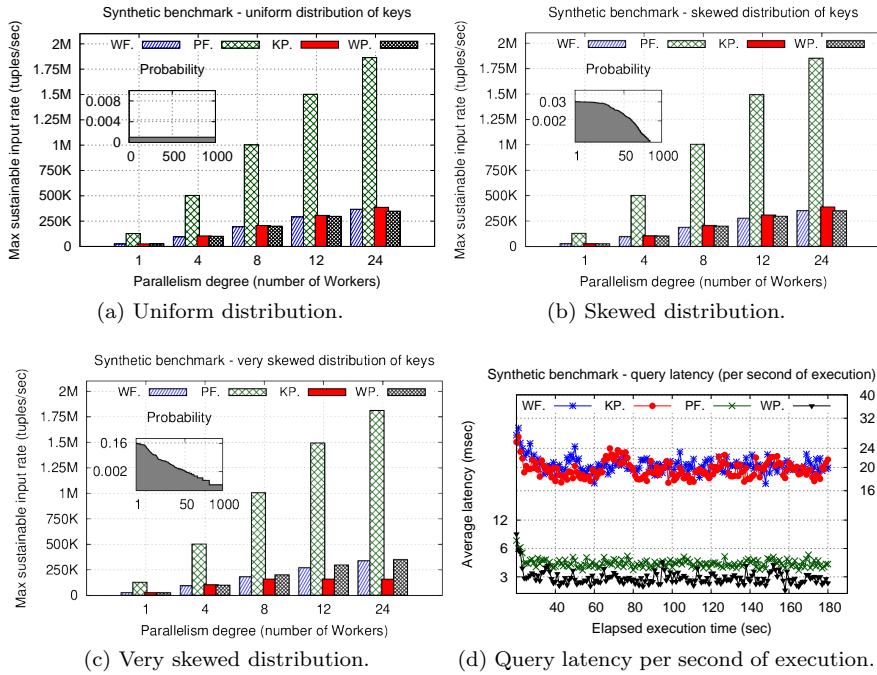


Fig. 6: Throughput and average latency with window farming (WF), key partitioning (KP), pane farming (PF) and window partitioning (WP). Distribution granularity of 1 tuple.

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 7: Scalability with 12 and 24 workers. Syntax ParDegree:Sca1.

In terms of memory occupancy the three patterns behave similarly. According to the FastFlow model, pointers are passed from the emitter to workers, thus tuples are actually *shared* rather than physically replicated. Consequently, no *batching* is necessary in WF. In conclusion, this benchmark shows that parallel patterns have different features. When applicable PF is preferable for throughput optimization while WP is the one giving the best latency outcome.

4.2 Time-based Skyline Queries

In this concluding section we study a real-world continuous query computing the *skyline* set [22] on the tuples received in the last $|\mathcal{W}|$ time units (*slide-*

by-tuple time-based window). Skyline is a class of preference queries used in multi-criteria decision making to retrieve the most interesting points from a given set. Formally, tuples are interpreted as d -dimensional points. A point $x = \{x_1, \dots, x_d\}$ belongs to the skyline set if there exists no *dominator* in the current window, i.e. a point y such that $\forall i \in [1, d] y_i \leq x_i$. The output is a set of skyline updates (entering or exiting of a tuple from the skyline).

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. Thus, the natural parallel pattern for this computation is Window Partitioning (WP). The skyline algorithm performs an intensive *pruning* phase [22]: 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 [22]. However, it produces a severe load unbalance because the partition sizes can change very quickly at run-time, even if the distribution evenly assigns new tuples to the workers.

Fig. 7a shows the maximum sustainable input rate with the three point distributions studied in [22]: **correlated**, **anticorrelated** and **independent**. Each distribution (represented in Fig. 7b) is characterized by a different pruning probability (higher in the correlated case, lower in the anticorrelated one).

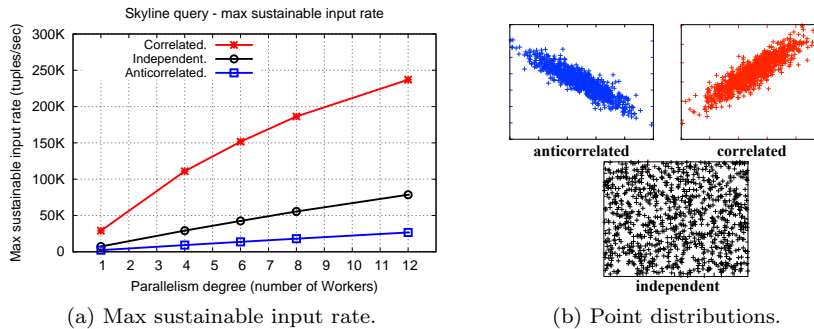


Fig. 7: Skyline query: distributions of points and maximum sustainable input rate per parallelism degree with the window partitioning pattern (WP). Windows of 10 seconds.

Each new tuple is assigned to the worker with the smallest partition to balance the workload. The correlated case is the one with the highest max rate, since partitions are smaller (due to pruning) and the computation has a finer grain. Load balancing is the most critical issue: the scalability with 12 workers is 8.16, 10.7 and 11.65 in the correlated, independent and anticorrelated cases. With a higher pruning probability it is harder to keep the partitions evenly sized. Hyperthreaded configurations are not effective. We omit them for brevity.

5 Related Work

This paper presented four parallel patterns targeting window-based DaSP computations. They represent an extension and specialization of the classic algorithmic skeletons [16] (notably, *pipeline*, *farm*, *map* and *reduce*). Existing SPEs allow the programmer to express only a subset of these patterns, usually without some of the possible optimizations and variants.

Storm [2] provides two primitive concepts: *spouts* (data sources) and *bolts* (operators). Storm does not have built-in stream operators and windowing concepts: users have to define the operator logic and the windowing mechanisms from scratch. Parallel patterns can be expressed by specifying how the input stream tuples are partitioned among multiple replicas of an operator (*grouping*). For multi-keyed stateful operators *field grouping* can be used for assuring that tuples with the same key are sent to same replica. Compared with our work this is similar to key partitioning, in which bolts are essentially active workers. The other parallel patterns introduced in this paper could be probably implemented on top of Storm by using active workers and custom grouping policies implementing user-defined distributions.

IBM InfoSphere [3] (IIS) is a commercial product by IBM. It provides a rich set of built-in operators and the possibility to implement user-defined operators. Like Storm, data can be routed at different operator replicas by means of *hashing*. Windowing is primitive in IIS, thus key partitioning can be easily implemented. In contrast, there is no support for customizable tuple routing, thus there is no direct possibility to mimic the other patterns.

Spark Streaming [1] runs applications as a series of deterministic batch computations (micro-batching) on small time intervals. It supports various built-in operators and a limited set of window-based operators working with associative functions (currently only time-based windows are implemented). While in Storm and IIS the operators are continuously running and tuples are processed as long as they appear, the Spark execution engine is essentially a task scheduler. Each operator is translated into a set of tasks with proper precedences/dependencies. The *count/reduceByWindow* operators are similar to the window partitioning pattern, while *count/reduceByKeyAndWindow* recalls the nesting of key and window partitioning.

6 Conclusions

This paper presented four parallel patterns targeting DaSP computations, defined as extensions of the classic algorithmic skeletons [16]. For each pattern we outlined the pros and cons and optimizations. We implemented the patterns in the **FastFlow** [4] framework for multicores. Extensions of our work consist in the integration of the patterns as high-level patterns in **FastFlow** with the support for distributed-memory architectures. In the future we plan to enhance the patterns with autonomic capabilities [19].

Acknowledgements This work has been partially supported by the EU H2020 project RePhrase (EC-RIA, H2020, ICT-2014-1).

References

1. Apache spark streaming. <https://spark.apache.org/streaming>
2. Apache storm. <https://storm.apache.org>
3. Ibm infosphere streams. <http://www-03.ibm.com/software/products/en/infosphere-streams>
4. Fastflow (ff) (2015). URL <http://http://calvados.di.unipi.it/fastflow/>
5. Aggarwal, C., Yu, P.: A survey of synopsis construction in data streams. In: C. Aggarwal (ed.) *Data Streams, Advances in Database Systems*, vol. 31. Springer US (2007)
6. Aldinucci, M., Calcagno, C., Coppo, M., Damiani, F., Drocco, M., Sciacca, E., Spinella, S., Torquati, M., Troina, A.: On designing multicore-aware simulators for systems biology endowed with on-line statistics. BioMed Research International (2014)
7. Aldinucci, M., Danelutto, M., Kilpatrick, P., Meneghin, M., Torquati, M.: An efficient unbounded lock-free queue for multi-core systems. In: *Proceedings of the 18th International Conference on Parallel Processing, Euro-Par'12*, pp. 662–673. Springer-Verlag, Berlin, Heidelberg (2012)
8. Arasu, A., Babu, S., Widom, J.: The cql continuous query language: Semantic foundations and query execution. *The VLDB Journal* **15**(2), 121–142 (2006)
9. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS '02*, pp. 1–16. ACM, New York, NY, USA (2002)
10. Balkesen, C., Tatbul, N.: 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 (2011)
11. Bertolli, C., Mencagli, G., Vanneschi, M.: Analyzing memory requirements for pervasive grid applications. In: *Parallel, Distributed and Network-Based Processing (PDP)*, 2010 18th Euromicro International Conference on, pp. 297–301 (2010). DOI 10.1109/PDP.2010.71
12. Cole, M.: Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Computing* **30**(3), 389 – 406 (2004)
13. Cugola, G., Margara, A.: Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.* **44**(3), 15:1–15:62 (2012)
14. Dobra, A., Garofalakis, M., Gehrke, J., Rastogi, R.: Processing complex aggregate queries over data streams. In: *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, SIGMOD '02*. ACM, New York, NY, USA (2002)
15. Gedik, B.: Partitioning functions for stateful data parallelism in stream processing. *The VLDB Journal* **23**(4), 517–539 (2014)
16. González-Vélez, H., Leyton, M.: A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Software: Practice and Experience* **40**(12), 1135–1160 (2010)
17. Hirzel, M., Soulé, R., Schneider, S., Gedik, B., Grimm, R.: A catalog of stream processing optimizations. *ACM Comput. Surv.* **46**(4), 46:1–46:34 (2014)
18. Li, J., Maier, D., Tufte, K., Papadimos, V., Tucker, P.A.: No pane, no gain: Efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Rec.* **34**(1), 39–44 (2005)
19. Mencagli, G., Vanneschi, M.: Towards a systematic approach to the dynamic adaptation of structured parallel computations using model predictive control. *Cluster Computing* **17**(4), 1443–1463 (2014)
20. Patroumpas, K., Sellis, T.: Maintaining consistent results of continuous queries under diverse window specifications. *Inf. Syst.* **36**(1), 42–61 (2011)
21. Tanbeer, S.K., Ahmed, C.F., Jeong, B.S., Lee, Y.K.: Sliding window-based frequent pattern mining over data streams. *Information Sciences* **179**(22), 3843 – 3865 (2009)
22. Tao, Y., Papadias, D.: Maintaining sliding window skylines on data streams. *Knowledge and Data Engineering, IEEE Transactions on* **18**(3), 377–391 (2006)