A High-Throughput and Low-Latency Parallelization of Window-based Stream Joins on Multicores

Daniele Buono, Tiziano De Matteis and Gabriele Mencagli
Department of Computer Science, University of Pisa
Largo B. Pontecorvo, 3, I-56127, Pisa, Italy
Email: {d.buono, dematteis, mencagli}@di.unipi.it

***Abstract*—Data Stream Processing (DaSP) is a paradigm characterized by on-line (often real-time) applications working on unlimited data streams whose elements must be processed efficiently "on the fly". DaSP computations are characterized by data-flow graphs of operators connected via streams and working on the received elements according to high throughput and low latency requirements. To achieve these constraints, high-performance DaSP operators requires advanced parallelism models, as well related design and implementation techniques targeting multi-core architectures. In this paper we focus on the parallelization of the *window-based stream join*, an important operator that raises challenging issues in terms of parallel windows management. We review the state-of-the-art solutions about the stream join parallelization and we propose our novel parallel strategy and its implementation on multicores. As demonstrated by experimental results, our parallel solution introduces two important advantages with respect to the existing solutions: *(i)* it features an high-degree of configurability in order to address the symmetricity/asymmetricity of input streams (in terms of their arrival rate and window length); *(ii)* our parallelization provides a high throughput and it is definitely better than the compared solutions in terms of latency, providing an efficient way to perform stream joins on latency-sensible applications.**

Keywords*—*Data Stream Processing, Window-based Stream Joins, Continuous Queries, Throughput, Latency, Multicores

I. Introduction

Data Stream Processing (*DaSP*) is a recent and highly active research field. In a DaSP computation data are not modeled as traditional permanent data structures or relations, but as continuous streams whose elements must be processed "on the fly", with critical performance requirements in terms of throughput and latency. Several important on-line and real-time applications can be modeled as DaSP [18], [6], including network traffic and sensor data analysis, financial trading, data mining and many others.

The topic of DaSP emerged from the execution of database queries on systems supplied by potentially infinite streams of data (*continuous queries*). The semantics of blocking operators (e.g. sorting, grouping, skylines, aggregate functions) and of stateful ones (e.g. joins, intersection) requires a special attention when input data sets are continuous streams. Blocking operators require to process the entire input before an output can be delivered, whereas stateful operators may require to buffer all the received elements leading to unbounded memory occupancy. In both the cases the common solution is to perform the computation on a finite subset of the input (named *window*), representing the most recent portion of the stream. Several software infrastructures for continuous queries have been proposed in the past [2], [11], [10], [8], [5], [15], extending the SQL syntax to support window-based variants of the common operators.

High-performance DaSP applications need *high throughput* and *low latency* implementations of continuous query operators. Latency is especially critical in many data intensive applications, e.g. notably in algorithmic trading and environmental monitoring [6], [11]. The problem of designing and implementing efficient DaSP computations is a quite complex one, given the presence of multiple streams, with unlimited or unknown length and different arrival rates. Furthermore, the presence of complex correlations between stream elements and the dynamic management of windows exacerbates the problem of writing efficient parallelizations.

The aim of this paper is to study the design issues of parallel implementations for *window-based stream joins* [14]. Stream joins are very important operators, needed whenever data from different streams have to be combined to calculate correlations. For this reason, parallel solutions have been already proposed in the past. In this work we describe the original sequential algorithm devised by Kang [16] and we investigate the parallelization issues posed by the window distribution and management. We review the existing approaches, in which windows are partitioned among cores. Data distribution is performed in a centralized fashion, in the case of CellJoin [12] (an implementation explicitly developed for the IBM Cell processor), and with a fully decentralized approach in Handshake Join [20] (a more recent parallelization targeting general-purpose multicores).

Our contribution consists in a different approach with respect to existing solutions. We design a scalable, low latency parallelization obtained by making use of both *partitioning* and *replication* (sharing on shared-memory architectures) of window data across cores. The degree of partitioning and replication of each window depends on the topology (also called *layout*), a configurable feature of our parallelization. To increase the scalability, data is distributed in a partially decentralized way, by exploiting core-to-core communications. To prove the effectiveness of our parallelization in terms of throughput and latency we present a wide range of experiments:

- we compare the throughput of our parallelization with the one achieved by the most recent existing solution [20]. The results show that our approach provides better throughput with the same number of cores;
- we propose a novel latency analysis which *has not*

been discussed in the previous works [12], [20]. To provide a comparison, we have adapted the source code of Handshake Join [20] to collect the latency measurements. In particular the results show that our parallel solution outperforms Handshake Join, providing a significantly lower latency;
- we analyze the behavior of our parallelization with different layouts, showing that the best one optimizing latency and throughput depends on the characteristics of the windows and the stream rates;
- we study the parallel behavior in *symmetric* and *asymmetric* scenarios (depending on the arrival rates and window lengths of the streams).

The outline of this paper is as follows. Sect. II presents related works for parallel DaSP and window-based stream joins. Sect. III reviews the existing solutions and presents our parallel proposal. Sect. IV presents throughput and latency results on an Intel multicore. Finally, Sect. V states the conclusion of the work and the future research directions.

II. Related Work

Several software infrastructures (Data Stream Management Systems - DSMSs) enabling continuous query execution have been proposed over the last years. Examples are Borealis [2], System S [11], GigaScope [10], NiagaraCQ [8] and, more recently, StreamCloud [15]. One of the main issues of DSMSs is the performance of the continuous queries. A first solution relies on exploiting parallelism among operators of the same query or among different queries (*inter-operator and inter-query parallelism*), by properly scheduling operators or entire queries on the available computing resources. This approach has been followed by System S [4], in which graphs of queries are partitioned into sub-queries assigned to a set of parallel processing elements. A similar approach has been adopted in [23], with special attention to load balancing mechanisms able to adapt the mapping between operators and resources according to the current stream rates. On Cloud infrastructures the same problem has been addressed in [15], with special attention on the use of nodes also by taking into account the virtualization overhead and the economic cost of the cloud.

A different approach consists in exploiting parallelism inside critical operators (*intra-operator parallelism*), i.e. for hot spot operators in the query graph. As an example, parallelizations of stateful operators have been described in [22] using locks to access shared data, and in [21] for computations amenable to be parallelized using MapReduce-like frameworks. However, the application of parallelism paradigms to DaSP can be much more complicated, especially when stateful operators involve a sliding (overlapping) window semantics [9]. A study of window distributions has been proposed in [7], by referring to tumbling and sliding windows and providing optimizations (i.e. pane-based distributions) when operators are based on associative functions.

Stream join operators are critical in DaSP applications. They can be used to detect trends and to find correlations between streams [16], [14]. Such computations are difficult to be parallelized efficiently: because of the symmetric semantics of joins, effective distribution strategies must be taken into account when designing an efficient parallelization. Furthermore, it is important to be aware of the effect of a parallelization not only on throughput, but on latency too. Some parallel solutions have been proposed in the literature. Two notable works are CellJoin [12] and Handshake Join [20]. The former has been evaluated only on a low-parallelism architecture (IBM Cell), while the latter provides good scalability but has limitations for latency-sensible applications. Other parallelizations exist (as a small example [3], [19]), but they operate on static relational tables and are not designed for performing joins over unbounded streams.

III. Parallel Stream Joins on Multi-core Architectures

In this paper we are interested in parallel solutions to apply join operators on streams. Join is one of the most common database operators, used to combine records from multiple tables. In its basic definition only two relations are used (*binary joins*); this, however, represents a meaningful case because it is possible to evaluate joins over more than two tables (multi-way joins) by using trees of binary join operators. For this reason we will refer to the problem of binary join throughout the paper, without losing generality. When we present the parallel solutions, we will just note whether or not multi-way joins can be natively supported by adapting the specific parallelizations.

The execution of joins poses semantics problems when input data come from unbounded streams [13], [7], [9]. Joins are *stateful* operators that require to store the entire input to produce outputs. In the case of unlimited streams, this would require unbounded memory and processing resources. An approximate solution is to apply these operators on a finite portion of each stream, named *window*, which dynamically changes over time. Conceptually a window represents a "slice" of the stream containing the most recent tuples, and thus a limited portion of it. According to the application requirements, various types of windows can be defined, with a different semantics in terms of boundaries and movement strategy of the window along the stream.

In analogy with CellJoin and Handshake Join, in this paper we focus on *sliding windows* that cover all the stream elements received from an earlier point up to the most recent tuple. Sliding windows can span over a fixed number of tuples, i.e. the last k tuples arrived (*count-based windows*) or over a fixed amount of time, i.e. tuples received within the last T_w time units (*time-based windows*). Our parallelization proposal can be applied both for time-based and count-based sliding windows. Time-based windows, however, are the most interesting case, since they retain a variable number of tuples during the execution based on the arrival rate of the stream and the window length T_w. In the sequel we refer to the case of sliding windows with a time-based semantics.

Given two streams Y and X, a pair of tuples must be compared if the following conditions are satisfied:

Definition 1 (Time-based Sliding Window Semantics)**.** *Given two tuples $x \in X$ and $y \in Y$ the join predicate $\bowtie_p$ will be evaluated on the pair if and only if the following time conditions are satisfied:*

- *if x is older than y, then the timestamps of the two tuples (denoted by t_x and t_y) must respect the following condition: $t_x \geq t_y - T_w^x$;*

- *otherwise, y must be in the current Y-window when x arrives, i.e. $t_y \geq t_x - T_w^y$.*

We denote by T_w^x and T_w^y the windows lengths (in time units) of the two streams.

To compute the join we adopt the general procedure devised by Kang [16]. It consists in the following sequence of actions performed at each new arrival of a tuple x from X (the actions are symmetric for the other stream Y):

1) scan the tuples in Y-window, evaluate the join predicate with x and propagate the results on the output stream R;
2) insert the new tuple x into the X-window;
3) remove all the expired tuples from the Y-window.

This version of the algorithm applies the invalidation in a *lazy* mode, i.e. expired tuples are removed when the corresponding window is probed to find matching tuples with the arrived element from the other stream. As described in [16], this solution optimizes the join processing cost (it scans only one window at each new arrival). In this paper we will assume this version of the algorithm in our description.

The general procedure used by Kang's algorithm is a *nested-loop evaluation*, in which the join predicate is evaluated for all tuples satisfying the window constraints. For certain predicates (e.g. equi-joins) the procedure can be optimized using efficient data-structures (e.g. hash tables and tree indices) in order to avoid to enumerate all the possible pairs. In the following we suppose a generic join problem on streams without specifying the predicate and which kind of data structures are effectively used. In fact, as we will see, all the parallelizations will be completely agnostic w.r.t the specific implementation of Kang's algorithm.

In many applications stream joins must be performed by respecting strict throughput and latency constraints. To evaluate parallel solutions we introduce two metrics:

Definition 2 (Throughput)**.** *The throughput is the average number of output results (joined pairs of tuples) per time unit produced onto the output stream R.*

Definition 3 (Latency)**.** *Let us suppose that two tuples $x \in X$ and $y \in Y$ joins and the output result $r = (x, y)$ is produced at time t_r. The latency l_r of this pair is given by:*

$$l_r = t_r - \max\{t_x, t_y\} \tag{1}$$

i.e. the time from the reception of the most recent tuple of the pair to the time at which the result is ready to be transmitted to the output stream.

In Sect. III-A and III-B we will review different parallelization strategies applied by existing research works. In Sect. III-C we will describe our parallel proposal that represents the contribution of this paper.

A. Centralized Solution

The idea of this parallelization is to have a centralized Emitter functionality in charge of distributing stream elements to a set of parallel Workers. Each time a new tuple is received from one of the two streams (e.g. $x_i \in X$) we need to apply the join predicate between x_i and all the elements in the Y-window, which is well defined: it contains all the elements of stream Y received in the last T_w^y time interval. The Emitter must: *(i)* insert the element x_i in the X-window; *(ii)* remove the expired tuples from the Y-window; *(iii)* determine a partitioning of the Y-window in N segments, where N is the parallelism degree (the number of Workers); *(iv)* scatter the Y-window to the set of Workers; *(v)* multicast the element x_i to all the Workers. The actions are performed symmetrically at each reception of a tuple from the other stream Y. Each Worker applies step 1 of Kang's algorithm to evaluate the joins between the received element and the window partition. The idea of this parallelization has been used in CellJoin [12].

This solution, conceptually applicable to more than two input streams, is able to optimize throughput and latency *provided that* the Emitter has a sufficiently low service time to sustain the input streams rate. The data distribution overhead (actions *iv*, *v*) can be mitigated on shared-memory architectures by allocating windows contiguously and sending memory pointers to the correct areas, instead of actual values. If determining a balanced partitioning at each arrival becomes a problem (action *iii*), the Emitter can use the same partitioning on subsequent elements (of the same stream) and adapt it periodically to re-balance the amount of elements per partition. Nevertheless, actions *i* and *ii* cannot be further optimized and can possibly make the Emitter a bottleneck with a high number of Workers and very fast input streams. In the implementation proposed in [12] the authors exploit the heterogeneous nature of the IBM Cell (from which the name derives) by placing the Emitter on the single PPE and the Workers on the 8 SPEs. In this case, due to the very limited number of Workers, the Emitter does not become a bottleneck and the application scales for low parallelism degrees.

B. Pipeline-based Solution

To solve the scalability problems of the centralized solution, alternative parallelization models can be defined in order to perform the data distribution in a decentralized manner. The *pipeline-based* model is a potential candidate, with a fully decentralized window partitioning and management among the pipeline stages. In order to efficiently meet pipelining features and joining requirements, an interesting variant to the pure pipeline solution has been proposed in [20] with the so-called *Handshake Join* parallelization, valid for the case of two streams (binary joins) that flow in the opposite directions as depicted in Fig. 1. In this way each element of one of the two stream will encounter, sooner or later, any element of the other stream. In the figure we denote by X-i and Y-j the i-th and the j-th partition (also called segment) of the X-/Y-window.

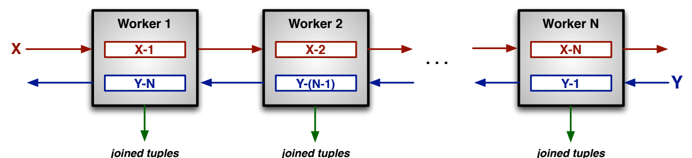


Fig. 1: Abstract scheme of the Handshake Join parallelization.

To scale efficiently, this parallelization scheme needs a distributed load balancing mechanism between Workers. In

order to keep the size of the partitions as similar as possible, neighboring Workers exchange tuples (from left to right for stream X and vice-versa for Y) when they detect a local unbalance of their partition sizes. Thank to this mechanism, tuples will be pushed in the pipeline and become evenly distributed over all the Workers throughout the execution (see [20] for further details).

As it is quite evident in this case, the throughput and scalability advantages are achieved at the expense of a too long latency. In fact, a new tuple x before being compared with all the tuples in the Y-window, needs to cross a subset of the stages of the pipeline whose number depends on the speeds of the two streams and the window (time) lengths. As it will be experimentally proved in Sect. IV-C, this solution is not acceptable for latency-sensitive applications.

C. A Low-Latency Stream Join Parallelization

In both the previous approaches the rationale is to partition the windows to execute each join in parallel among the set of Workers. Our proposal is based on a different approach in which the windows are both partitioned and replicated among a set of Workers to execute the joins on each received tuple *and* between multiple arrived tuples in parallel. The degree of partitioning and replication can be changed by exploiting different topologies (layouts). Furthermore, as it will be discussed in Sect. III-D2, on shared memory architectures replication can be efficiently implemented by using shared data among Workers.

In the case of two streams, bidimensional matrix-like layouts of Workers can be arranged (Fig. 2), with the window of stream Y partitioned among Workers on different columns and each partition replicated on the corresponding column. The situation is symmetric for the other stream X as depicted in Fig. 2. Tuples are assigned to a set of Workers at their arrival; in this way each Worker maintains its own window partition for each stream and: *(i)* the ownership of a tuple is held by a subset of the Workers (a row/column, based on the topology); *(ii)* each Worker executes all the steps of Kang's algorithm, by removing expired tuples in their partitions independently. This second point represents an important difference with respect to the previous approaches, since in CellJoin window updates are performed by the centralized Emitter, whereas in Handshake Join tuples are inserted/removed by flowing in the pipeline.

To implement the layout, proper data distribution activities must be performed. To avoid the issues of the centralized approach, we devise a partially decentralized data distribution performed in a collaborative fashion by the Emitter and the Workers. The distribution, outlined in Fig. 2 for a generic layout of $m \times n$ Workers, is performed as follows:

- a tuple $y \in Y$ is sent to a Worker $W_{1,i}$ in the first row selected by the Emitter. The tuple is forwarded to all the Workers on the i-th column (the Emitter transmits y to $W_{1,i}$, that forwards it to $W_{2,i}$, and so on up to Worker $W_{m,i}$);
- symmetrically, at each new arrival of a tuple $x \in X$ the Emitter selects a Worker $W_{j,1}$ in the first column. The tuple is multicasted to all the Workers in the j-th row using the same approach as before.

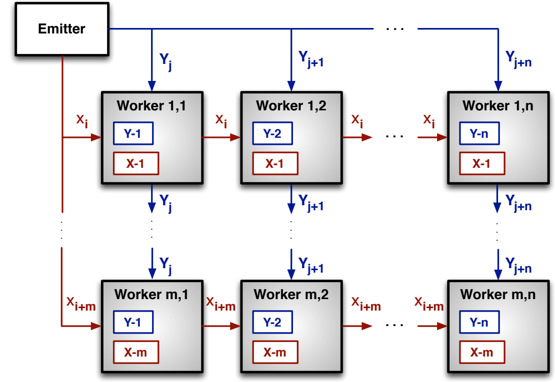


Fig. 2: General layout of $m \times n$ Workers. We denote by X-i and Y-j the i-th and j-th partition of X-window and Y-window respectively.

Given the fact that the ownership of each tuple is fixed throughout the execution, a critical point is to provide a load-balanced partitioning. If the join processing time is constant, or with a small variance, then *load balancing* can be achieved by maintaining, for each window, the same number of tuples on each Workers. In this case a simple yet effective policy is to distribute tuples in a round-robin fashion, e.g. upon the reception of a tuple $y_i \in Y$, the Emitter forwards it to a Worker $W_{1,k}$ in the first row such that $k = (i \bmod n) + 1$, and symmetrically for the other stream. In this way new tuples are *evenly distributed* to Workers and *evenly removed* from their partitions at their expiration time. It is worth noting that the windows are not partitioned in contiguous segments, but each Worker has its own set of tuples that span over the entire window length. Other strategies, such as on-demand distribution, can be adopted to provide better load balancing if the join processing time has a high variance.

Proposition 1 (*Correctness*). *The window-based stream join parallelization with a layout of $m \times n$ Workers outputs all the pairs of tuples satisfying the join predicate $\bowtie_p$ and respecting the timing constraints of Definition 1.*

Proof sketch: to have an idea of the proof, let us consider the situation at moment t_{y_i} when a tuple $y_i \in Y$ is received by the Emitter (the situation is symmetrical for the other stream). The tuple is multicasted to the k-th column of the layout such that $k = (i \mod n) + 1$. Tuples from each stream are received and processed by each Worker in-order (i.e. y_i is evaluated before the evaluation of y_j if and only if $t_{y_i} < t_{y_j}$). Each Worker $W_{*,k}$ scans its X-window partition and evaluates for each pair (x, y_i):

1) the *timing constraints*, i.e. x will be removed from the partition if $t_x + T_w^x < t_{y_i}$. It is worth noting that with this action we will not lose time-compatible pairs because tuples from the same stream are received in timestamp order;
2) for all the tuples x satisfying the timing constraints the Worker evaluates $x \bowtie_p y_i$. If the join condition holds, the pair (x, y_i) is transmitted onto the output stream R.

Since the union of the partitions of Workers $W_{*,j}$ is the full window of stream X, all the time-compatible pairs are

analyzed and the results produced onto the output stream. ■

Using matrix-like topologies we can decide a proper allocation of Workers to rows and columns. We can identify three notable cases: *(i) linear layouts*, in which $m = 1$ or $n = 1$, resulting in the partitioning of a window and the replication of the other; *(ii) square layouts*, in which $m = n = \sqrt{N}$, so that windows are equally partitioned and replicated, or *(iii)* a general *rectangular layout,* where $m \neq n$.

Although the degree of parallelism in term of total number of Workers (i.e. $N = n\,m$) is an important aspect to optimize throughput, the layout plays a decisive role to minimize latency. At each reception of a tuple x (y) from stream X (Y), the average latency depends on the number of Workers in a row (column) of the layout. In other words, the smaller the window partitions, the smaller the latency of the joins on x (y). The intuitive result is that in a *symmetric scenario*, with streams having the same arrival rate and window length ($\lambda_x = \lambda_y$ and $T_w^x = T_w^y$), the size of the two windows in terms of tuples is the same, and the layout optimizing the latency is a square one, i.e. with the same number of Workers per column and per row. In the case of *asymmetric streams*, the size of the two windows can be in general different, and a proper rectangular (or even linear) layout can be the best solution to provide low latency. Both the situations will be studied in more detail in Sect. IV.

We point out that our parallelization and the distribution strategy have a *regular* structure with Workers having two input and two output neighbors. This structure is amenable to be efficiently executed on NUMA-like shared-memory parallel architectures such as multi-processors of multi-core CPUs (e.g. Intel multi-processors with the `QuickPath` interconnect). Neighbor Workers can be easily mapped onto cores in order to exploit short-distance communications (resulting in a higher probability to find data in shared levels of cache). The Emitter is a little price to pay: it performs only point-to-point communications with a sub-set of the Workers (the ones in the first row and in the first column of the layout).

Finally, our approach can be easily extended to more than two input streams (multi-way joins). As an example, in the case of three streams we can arrange a cube of Workers each one maintaining a window partition for each stream. Now the Emitter is responsible to distribute tuples over Workers lying on three adjacent faces of the cube. Hypercubes can be imagined with more than three streams. In this paper we do not discuss this extension, that will be covered in our future work.

D. Implementation and Optimizations

Starting from the the abstract parallelization model described in Sect. III-C, several optimizations can be introduced to target general-purpose shared-memory multicores. The most important points discussed here are: (1) how data structures are organized in memory; (2) how replication has been really implemented.

1) Attribute-oriented organization: each received tuple consists of a set of attributes $\{a_i\}_{i=1}^k$. Two different ways to store tuples in memory can be identified [12]: (1) *tuple-oriented* (row-oriented) organization: different tuples are stored contiguously in memory; (2) *attribute-oriented* (column oriented) organization: we use k separated data structures to contiguously store the same attribute of different tuples.

As stated in [12], [20], the second approach should be preferred to reach better performance. It can reduce the amount of *cache lines* transferred to evaluate the join predicate over the entire window (in general the join predicate applies on a limited set of attributes). Furthermore, according to the specific definition of the join operator, such organization can be useful to exploit the *SIMD capability* of modern CPUs without overhead in arranging temporal structures for the operands of SIMD instructions. In analogy to the works [12], [20], we use this organization in our parallel stream join implementation.

Our parallel solution exploits both replication and partitioning of the two stream windows onto a generic matrix topology of Workers. Workers belonging to different rows/columns operate on different segments of the same stream window (e.g. conventionally X-window is partitioned among Workers on different rows and Y-window among Workers on different columns). The opposite is for replication: Workers on the same row/column have a replica of the corresponding segment of X-/Y-window. An example of such organization is shown in Fig. 3 with a square layout of nine Workers: by supposing only two attributes a_1 and a_2 per tuple, each window segment consists of two attribute-oriented data structures denoted by $X.A_1$, $X.A_2$ and $Y.A_1$, $Y.A_2$.

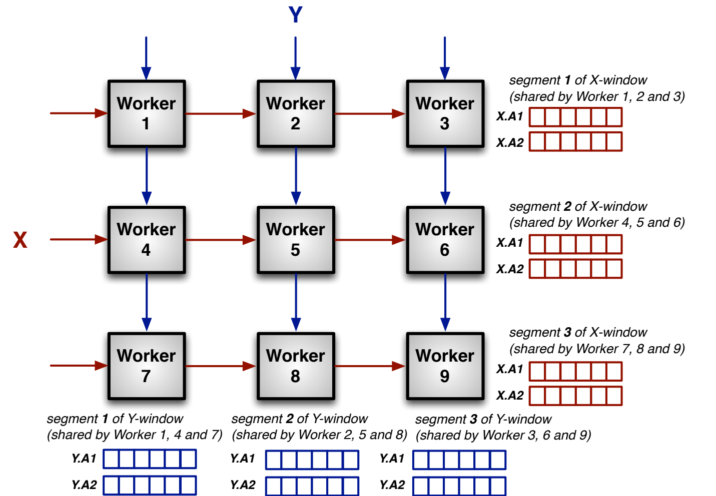


Fig. 3: Attribute-oriented data organization. X-/Y-windows are replicated (actually shared) and partitioned among Workers.

2) Support to replication: replication yields to a higher memory occupation. On today's multicores, replication can be implemented efficiently from the memory occupancy viewpoint by exploiting *sharing* instead of pure replication of data-structures, provided that the functionalities of the parallel program share part of their logical address space (e.g. they are implemented as threads). Let us focus on how replication of X-window segments is realized (the same principle applies to Y-window). Workers belonging to the same row access the same segment of the X-window (see Fig. 3). By relying on shared memory, the segment is physically shared by Workers

on the corresponding row: they have the address to the initial portion of the same X-window segment.

Using shared window segments, communications between Workers have a different nature. Once the Emitter receives a tuple x, it firstly selects the destination row in a round-robin fashion, then the tuple is transmitted to the first Worker of that row. The Worker is responsible to physically modify its X-window segment by adding the new tuple. All the other Workers on the same row only require to be notified of the presence of the new tuple to start the join evaluation with their segment of Y-window. The notification is performed exchanging special messages of the minimum size as possible according to the underlying architecture specifications (usually messages of one word). Such optimization has two important consequences on our parallelization:

- by avoiding real replicas of window segments, the memory occupancy of the parallel program will be roughly equal to the sequential version;
- sharing makes it possible to exchange smaller messages between Workers instead of real tuples. This leads to further improvement in latency, since Workers on the same row/column can start the join evaluation earlier compared with a version with pure replication and communication of entire tuples by value.

Such way to perform replication has important impacts on the way in which expiring tuples are identified and removed from the two windows. In the abstract parallelization model shown in Sect. III-C, each Worker is responsible to remove expired tuples in its window segments. When sharing of window segments is applied, this activity must be performed in a careful way. Each expired tuple must be physically removed by only one Worker if and only if it is no longer necessary to any Worker sharing the same window segment. We manage this with a low complexity procedure:

- each Worker maintains a counter to the first valid tuple for each of its two window segments;
- Workers mark their expired tuples locally, by increasing the corresponding counter (tuples are maintained ordered w.r.t the timestamp attribute);
- expired tuples are physically removed periodically by one of the Worker sharing that window segment, by identifying expired tuples for all the Workers sharing the same window segment (i.e. by taking the minimum between the Worker counters).

IV. Experiments

In this section we provide an experimental evaluation of throughput and latency achieved by our parallel solution, referred to as *DP-Join* in the sequel[1]. Furthermore, we compare our approach with the most recent existing parallelization [20] - Handshake Join[2]. In Sect. III-A we saw that CellJoin [12] is an example of centralized implementation with low parallelism designed for a special-purpose architecture (IBM Cell), while in this section we are interested in studying solutions for general-purpose CPUs without parallelism degree constraints.

A. Experimental Setup

In this section we study the join problem with a *band-join* predicate. We use the band-join for two reasons: *(i)* for a fair comparison with Handshake Join, since it is the same predicate discussed in their work; *(ii)* it is a type of join predicate applied in realistic applications that require low latency joins over continuous domains such as time and distance [16] (e.g. to find events occurring nearly in time and space). The band-join predicate introduced in [20] applies the following join condition over two tuples $x \in X$ and $y \in Y$:

```
WHERE x.a BETWEEN y.a - 10 AND y.a + 10
    AND x.b BETWEEN y.b - 10 AND y.b + 10
```

where a and b are the two join attributes. In our experiments we use the same tuple generators used in [20] that generates the attribute values with a join probability (*hit rate*) $p = 3.6 \cdot 10^{-6}$. Because of the predicate, we use the general nested-loop join algorithm, where each window is implemented by a dynamic array for each attribute (see Sect. III-D1). Tuples and their timestamps are pre-generated according to the input stream rates and read from file before each test execution.

DP-Join is evaluated on an Intel multiprocessor composed of two Xeon E5-2650 Sandy-Bridge CPUs for a total of 16 cores and 32 SMT thread contexts (Simultaneous Multi-Threading). Each core has a private L1 and L2 cache of size 32 KB and 256 KB. Each group of 8 cores share a L3 cache of 20 MB. Communications are implemented by passing pointers to shared data (as described in Sect. III-D2) through efficient lock-free queues made available by the `FastFlow` library [1].

B. Throughput Analysis

In the case of the stream join operator the maximum throughput (also referred to as bandwidth) B^{ω}_{max} is expressed as a function of the input stream rates λ_x and λ_y, the window lengths T^x_w, T^y_w and the hit rate p. For simplicity we refer to the case of two streams with the same window length T_w, but the reasoning can be easily generalized to the case of different lengths. For each received tuple from X in a time interval $[t_0, t_1]$ we compute the join with all the tuples in the Y-window and vice-versa. By denoting $\mathcal{W}_x = \lambda_x \cdot T_w$ and $\mathcal{W}_y = \lambda_y \cdot T_w$ the average window size (in terms of tuples) of stream X and Y at steady state, the number of output pairs in that time interval is given by:

$$\begin{aligned} \Omega_{t_0,t_1} &= \Big[\lambda_x \ (t_1 - t_0) \ \mathcal{W}_y + \lambda_y \ (t_1 - t_0) \ \mathcal{W}_x\Big] \times p \\ &= 2\lambda_x \, \lambda_y (t_1 - t_0) T_w \, p \end{aligned} \quad (2)$$

Therefore, the steady-state maximum bandwidth (number of outputs per time unit) can be evaluated as follows:

$$B^{\omega}_{max} = \frac{\Omega_{t_0,t_1}}{(t_1 - t_0)} = 2\lambda_x \, \lambda_y \, T_w \, p \quad (3)$$

This concept is important to understand that the maximum throughput achieved by any parallelization cannot exceed

[1]The source code of DP-Join is available at the current address http://www.di.unipi.it/∼mencagli/DP-Join.zip

[2]We thank the authors of [20] for having made available their source code at the current url: http://dbis.cs.tu-dortmund.de/cms/de/publications/2011/soccer-players/

B^{ω}_{max} because of the limitation imposed by the stream configuration. For example, we expect that by increasing the number of Workers (i.e. *parallelism degree*) we are able to increase the bandwidth; however, once B^{ω}_{max} is reached, adding more Workers will not produce further improvements. On the other hand, if an implementation is producing outputs at a rate lower than B^{ω}_{max}, this means that it is not able to sustain the input rates, representing a *bottleneck*. Therefore, it is important to find the *minimum parallelism degree* n^{bw} such that the achieved throughput is equal to B^{ω}_{max}.

1) Parallelism degree and scalability: we perform experiments in a symmetric scenario (Fig. 4): each stream has an input rate of 2100 t/s (tuples per second) and a window length T_w of 300 seconds. By applying Expr. 3, we are able to estimate the maximum bandwidth of the module as $B^{\omega}_{max} = 9526$ t/s. We compare the achieved bandwidth of Handshake Join and of several DP-Join layouts; each one is characterized by a different minimum parallelism degree n^{bw} able to achieve B^{ω}_{max}. Fig. 4 shows that the square and the rectangular layouts have slightly greater minimum parallelism degrees (8 and 9 Workers) compared with the linear one (7 Workers), *because they can not be used with any parallelism degree* (e.g. a square layout exists only for perfect square parallelism degrees). Furthermore, the results confirm the accuracy of Expr. 3: the measured B^{ω}_{max} and the predicted one differ less than 2%.

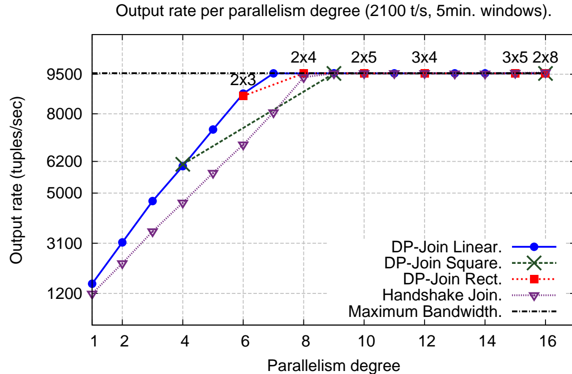


Fig. 4: Output rates of DP-Join and Handshake Join with a symmetric stream configuration. For the rectangular layout we show the number of Workers per row and column.

In the previous experiment our DP-Join does not scale beyond 9 Workers *only* because, with the used input rates and window length, 9 Workers are enough to reach the maximum bandwidth with any layout. A complete scalability analysis is depicted in Fig. 5a in a symmetric scenario with an input rate per stream of 6000 t/s and $T_w = 300$ seconds. By definition, the scalability with $n \times m$ Workers is $\mathcal{S}^{(m \times n)} = B^{\omega}_{m \times n} / B^{\omega}_{1 \times 1}$, where $B^{\omega}_{m \times n}$ is the achieved bandwidth with $n \times m$ Workers and $B^{\omega}_{1 \times 1}$ with *one* Worker. For brevity we show the results for a linear layout of Workers (similar results are achieved by the other layouts). With the used configuration in terms of input rates and window length we have not sufficient cores to reach B^{ω}_{max}. Therefore we are able to study how DP-Join scales up to the maximum number of cores of the architecture. With 15 and 16 Workers we exploit the SMT capabilities of the two CPUs by allocating the Emitter and the Collector threads on the same cores of two Workers. In general this could limit the scalability because of a reduced performance of these two Workers; in our case the problem is exacerbated by the `FastFlow` synchronization mechanisms (based on an aggressive busy-waiting [1]); to avoid this problem, we adopt a SMT-aware synchronization (based on busy-waiting with a slight delay) able to mitigate the overhead of the Emitter and Collector threads reaching a good scalability up to 16 Workers.

To provide a fair comparison, we have executed our parallelization and the Handshake Join on the Intel multiprocessor using the same data-set of input tuples with identical timestamps. Compared with Handshake Join, DP-Join is characterized by a more optimized sequential algorithm: in the case of one Worker the difference is of 24%. Given that from the scalability viewpoint (Fig. 5a) the two implementations behave very similarly with a near-optimal scalability, a similar gain is maintained with the increasing of the parallelism degree. The performance loss of Handshake Join using 15 and 16 Workers is due to the presence of two support threads, the Collector and the Driver ones [20] (the first collecting results from Workers and the latter in charge of pushing newly arrived tuples and pulling expired ones) and the use of synchronization not optimized for using multiple SMT contexts per core.

In Sect. IV-C we will see that different layouts have very important effects on the latency. In principle, *we expect that, layouts with the same parallelism degree are characterized by the same offered bandwidth*:

$$B^{\omega}_{m \times n} = \frac{m\,n}{T_{\bowtie}}\,p \tag{4}$$

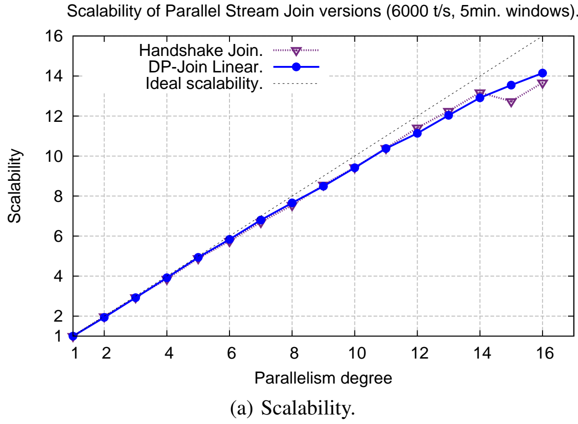
where $T_{\bowtie}$ is the average time to evaluate the join predicate on a pair of tuples, i.e. each Worker evaluates the join predicate every $T_{\bowtie}$ and produces a result every $T_{\bowtie}/p$ on average. It is important to note that this estimation is valid when the parallelization does not reach the maximum throughput (i.e. $B^{\omega}_{m \times n} < B^{\omega}_{max}$) and it is independent from the stream parameters, since it represents the maximum number of predicates that the parallelization can evaluate in a time unit. Tab. I shows the offered bandwidth using the same stream configuration of Fig. 5a (stream rates and window length) and three parallelism degrees equal to 6, 9 and 16 Workers for which several linear, rectangular and square layouts can be used. With 6 and 16 Workers we use a 2×3 and a 2×8 rectangular layout.

Parallelism Degree	Square Layout	Rectangular Layout	Linear Layout
6	-	9,301	8,529
9	14,047	-	13,599
16	21,740	21,700	21,659

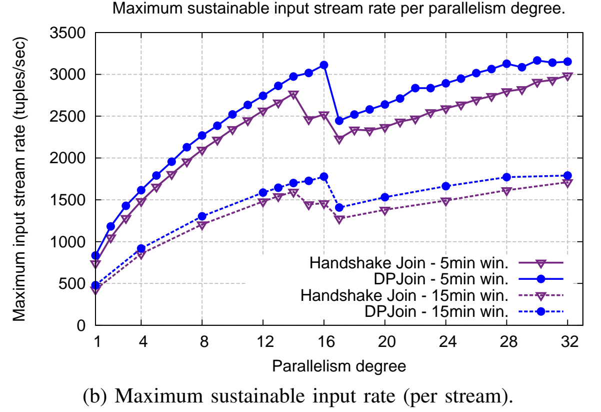
TABLE I: Offered bandwidth (t/s) by different layouts.

Due to the efficiency of the communication mechanisms there are small differences between layouts (in particular, the linear one provides a slightly lower performance), because of propagation delays along a longer sequence of Workers.

2) Sustainable input rate and multithreading: we propose a different bandwidth analysis by reproducing the experiment described in [20]. For each parallelism degree we measure



(a) Scalability.



(b) Maximum sustainable input rate (per stream).

Fig. 5: Throughput analysis of DP-Join (linear layout) and Handshake Join: (a) scalability; (b) maximum sustainable input rate (per stream) using different parallelism degrees.

the maximum input rate (per stream) that the parallel implementation can sustain without being a bottleneck. Results are depicted in Fig. 5b, also exploiting SMT multithreading with the linear layout using up to 32 Workers. We consider this layout because all the possible parallelism degrees can be used. We analyze two experiments: in the first we use symmetric streams with a window length of 300 seconds, in the second we use a longer window length of 900 seconds (the maximum length discussed in [20] and [12]).

The general result is that DP-Join outperforms Handshake Join, by providing higher maximum sustainable rates. We can notice a slowdown when we start to allocate multiple Workers on the same cores using SMT contexts, i.e. with more than 16 Workers in our parallelization (two contexts are used for the Emitter and the Collector threads). As we use higher parallelism degrees, the performance slowly increases, roughly reaching the same input rate sustainable by 16 Workers. In conclusion, while useful to efficiently support the Emitter and the Collector threads, SMT is ineffective to further increase the performance of our DP-Join, which is a positive signal on the efficiency of our code.

In Fig. 5a and 5b we can note a drop in performance using 15 or 16 Workers with Handshake Join, due to the presence of the Emitter and the Driver processes. Differently to our parallelization, Handshake Join exhibits a slight performance improvement by using more contexts per core. The reason is probably due to the performance inefficiency with 15 and 16 Workers. If the parallelization had been efficient with those parallelism degrees, probably the slight advantage of using more Workers per core would have been completely vanished.

C. Latency Analysis

Although the layout of DP-Join does not influence the offered throughput, *it is of great importance to minimize the average latency* (see Definition 3).

We consider an approximation of the average latency of a generic $m \times n$ layout, that allows us to qualitatively understand the effects on the latency. At each reception of a tuple $x \in X$, the Emitter distributes it to a row of Workers. Each Worker evaluates in parallel the joins on its Y-window segment. By assuming that joining tuples are uniformly distributed, the average latency can be approximated as follows:

$$L^x_{m\times n} = \frac{\sum_{i=1}^{\mathcal{W}_y/n} i\, T_{\bowtie}\, p}{p\,\mathcal{W}_y/n} \simeq \frac{\mathcal{W}_y\, T_{\bowtie}}{2\,n} \tag{5}$$

where $L^x_{m\times n}$ denotes the average latency of tuples received from X. Similarly we can compute $L^y_{m\times n}$ where the X-window segment is of size $\mathcal{W}_x/m$. The average latency is:

$$L_{m\times n} = p_x\, L^x_{m\times n} + p_y\, L^y_{m\times n} \tag{6}$$

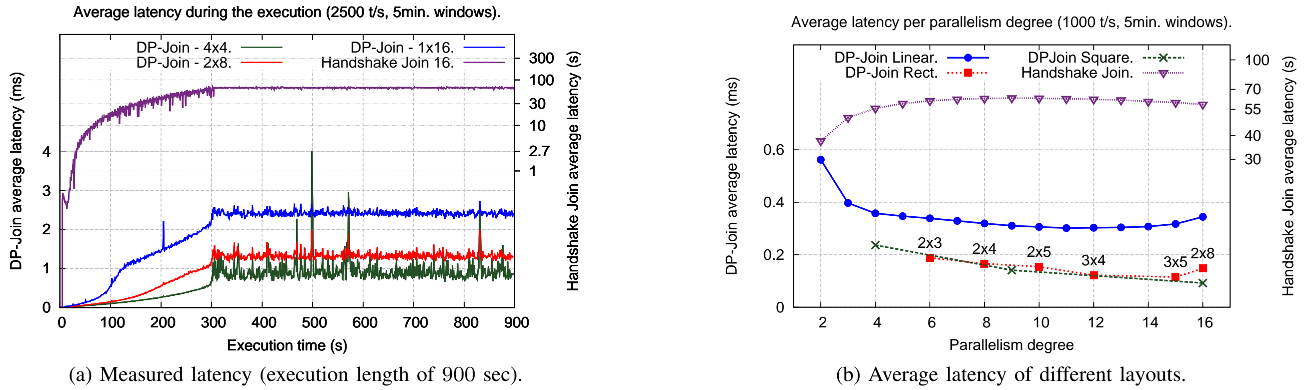
In a symmetric scenario, in which the two streams have the same arrival rate $\lambda_x = \lambda_y = \lambda$ and window length T_w, we can assume the same probability $p_x = p_y = 0.5$. In this case the previous expression can be rewritten in the following way:

$$L_{m\times n} \simeq \frac{\lambda\, T_{\bowtie}\, T_w (m+n)\, p}{4\, m\, n} \tag{7}$$

By comparing layouts using the same number of Workers ($m\,n$), we observe that the layout with the lowest latency is the one that minimizes the numerator of Expr. 7: $m+n$, i.e. the square one. This observation states an intuitive fact: in the case of symmetric scenarios, to minimize the average latency it is important to equally lower both the terms of Expr. 6.

1) Symmetric scenarios: we provide an evaluation using streams with the same rate of 2500 t/s and $T_w = 300$ seconds. In this scenario the amount of work needed for the elements received by the two streams is the same. The latency evaluation is meaningful when the system is not a bottleneck, i.e. the offered bandwidth equals the maximum one ($B^{\omega}_{m\times n} = B^{\omega}_{max}$). Fig. 6a shows the average latency of different layouts (with the same parallelism degree of 16 Workers) during an execution of three times the window length. We reach the steady state after 300 seconds, i.e. when the two windows reach their maximum size in terms of tuples. *The square layout provides the best latency* (0.95 ms on average). The latency is 60% better than the linear layout (2.42 ms), while the rectangular one provides a latency between the other two layouts confirming the intuition presented in Expr. 7.

Fig. 6b presents a more complete evaluation using different parallelism degrees. We consider streams of 1000 t/s and we



(a) Measured latency (execution length of 900 sec).

(b) Average latency of different layouts.

Fig. 6: Latency analysis in a symmetric scenario: (a) average latency over the execution; (b) average latency with different layouts and parallelism degrees.

take into account only the layouts with a number of Workers able to reach the maximum bandwidth. For each parallelism degree we show the average latency with the possible layouts. As confirmed by our analysis, *the square layout is the best one*. The best result is achieved using a 4×4 layout (the latency is slightly better than the 2×8 rectangular one). We can observe that the average latency is monotonically decreasing with the parallelism degree except with 15-16 Workers. In that case two Workers are executed on the same core of the Emitter and the Collector threads resulting in worse latency results.

In [20] the authors of Handshake Join have not analyzed their parallelization in terms of latency. To provide a comparison we have slightly modified their source code in order to collect such measurements. The latency comparison is summarized in Fig. 6a and 6b by showing the results of Handshake Join (67 sec on average) and different layouts of DP-Join, which *provides an average latency at least four orders of magnitude smaller*, and a minor standard deviation (0.33 msec wrt. 1.46 sec). For the sake of clarity, we show the Handshake Join latency using a different logarithmic scale (in seconds) represented in the right part of the plots. The reason for this great difference is related to the rationale behind Handshake Join. A tuple x (y) assigned to a Worker leaves its partition only when: *(i)* it becomes the oldest one in its window partition, and *(ii)* that Worker detects a local unbalance with its right (left) neighbor and the tuple is exchanged to balance the partition sizes as described in Sect. III-B. If you assume that the number of arrived tuples is uniformly distributed in time, after an initial filling phase of the window (equal to T_w) each tuple stays in a window partition for a time interval equal to T_w/N where N is the number of Workers, independently from the arrival rate. Therefore the joins with the last arrived element will be performed in a time proportional to the window length. In contrast, in our approach, at each reception of a new tuple *all* the comparisons are performed simultaneously.

2) Asymmetric scenarios: we show two experiments in which the window size of the two streams is different. The first experiment (Fig. 7a) shows two streams with different rates $\lambda_x = 1000$ t/s and $\lambda_y = 3000$ t/s, and the same window length $T_w = 300$ seconds. In this case Expr. 7 is no longer valid since the two windows have different sizes at steady-state; intuitively, *a proper rectangular layout is the best solution to minimize latency on both windows* because it is important to partition the larger window among a higher number of Workers. The results of Fig. 7a confirm this fact: the rectangular layout gives the best latency (0.26 ms with 3×5 Workers compared with 0.30 ms which is the best result obtained by the 4×4 square layout). Finally, it is interesting the behavior of the linear layout, where the latency increases as we use higher parallelism degrees (a similar behavior can be observed also in the symmetric scenario, Fig. 6b). This fact can be justified by the propagation delay of received tuples along the pipeline. This aspect deserves to be investigated more deeply in our future work. As discussed before, the average latency of Handshake Join is again several orders of magnitude higher than our DP-Join parallelization.

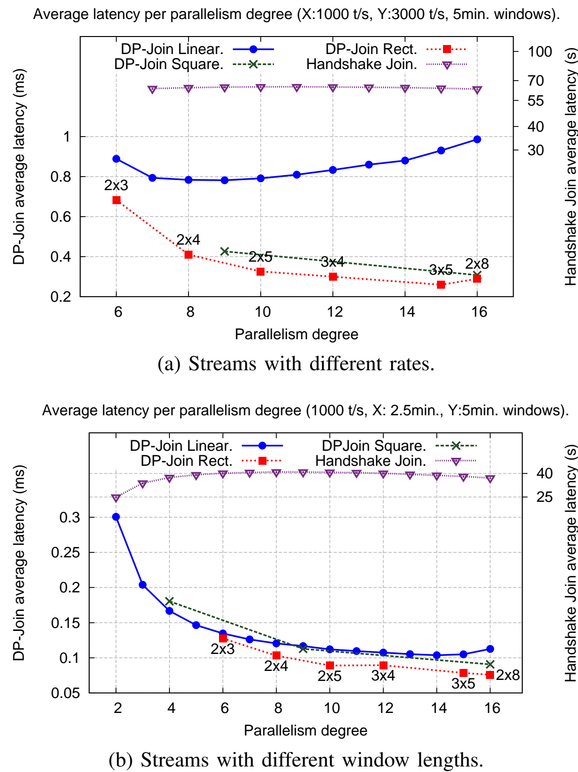
To conclude our discussion, we show a different asymmetric scenario by using the same input rate $\lambda = 1000$ t/s for the two streams and different window lengths equal to $T_w^x = 150$ seconds and $T_w^y = 300$ seconds. Conceptually, this is only a different source of asymmetricity, and can be handled again by selecting a rectangular layout that balances the size of each Worker partition. This experiment is reported in Fig. 7b. The qualitative results are very similar to the previous example, with the rectangular layout outperforming the others.

V. Conclusions

In this paper we investigate the problem of designing and implementing efficient parallel solutions for window-based stream joins, an important class of operators for continuous query applications. We present the sequential algorithm devised by Kang [16]. We introduce the main concepts of two existing parallelizations referred to as CellJoin [12] and Handshake Join [20]. Both the approaches are based on a pure partitioning of the stream windows among parallel Workers.

We introduce a parallelization based on a more complex partitioning and replication scheme, with Workers organized in bidimensional layouts. In order to alleviate the Emitter workload, a partially decentralized distribution is designed. We describe the optimizations for an implementation on multicores, by addressing the problems about how tuples are stored in memory, and the effective implementation of the replication scheme through sharing of window partitions among Workers.

The experiments show a deep analysis in terms of throughput and latency of our parallelization compared with Hand-



(a) Streams with different rates.

(b) Streams with different window lengths.

Fig. 7: Latency analysis in asymmetric scenarios: (a) streams with different arrival rates and the same window length; (b) streams with the same arrival rate and different window lengths.

shake Join, the other existing solution targeting general-purpose multicores. In terms of bandwidth our parallel version features a higher throughput (24% greater) and a near optimal scalability up to 16 cores. In terms of latency the comparison is novel and interesting. The results show a great advantage of our solution that maintains the average latency within few milliseconds, i.e. several orders of magnitude smaller than the latency provided by Handshake Join.

Improvements can be developed in the future, to study the behavior of our parallelization with more than two input streams (a case not addressed in Handshake Join). Finally, considering highly variable input rates, the approach is worth to be studied in terms of *dynamic adaptiveness*, i.e. providing adaptation strategies [17] to change the parallel structure in terms of distributions methods and layouts.

REFERENCES

[1] The fastflow (ff) parallel programming framework, 2014. http://mc-fastflow.sourceforge.net/.

[2] D. J. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J. hyon Hwang, W. Lindner, A. S. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the borealis stream processing engine. In *In CIDR*, pages 277–289, 2005.

[3] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *Proc. VLDB Endow.*, 5(10):1064–1075, June 2012.

[4] H. Andrade, B. Gedik, K. L. Wu, and P. S. Yu. Processing high data rate streams in system s. *J. Parallel Distrib. Comput.*, 71(2):145–156, Feb. 2011.

[5] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas, R. Varma, and J. Widom. Stream: The stanford stream data manager. *IEEE Data Eng. Bull.*, 26(1):19–26, 2003.

[6] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. 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, pages 1–16, New York, NY, USA, 2002. ACM.

[7] C. Balkesen and N. Tatbul. 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, August 2011.

[8] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: a scalable continuous query system for internet databases. *SIGMOD Rec.*, 29(2):379–390, May 2000.

[9] L. Chen and G. Lin. Extending sliding-window semantics over data streams. In *Computer Science and Computational Technology. ISCSCT '08. International Symposium on*, volume 2, pages 110–113, 2008.

[10] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. 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, 2003. ACM.

[11] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. Spade: the system s declarative stream processing engine. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1123–1134, New York, NY, USA, 2008. ACM.

[12] B. Gedik, R. R. Bordawekar, and P. S. Yu. Celljoin: a parallel stream join operator for the cell processor. *The VLDB Journal*, 18(2):501–519, Apr. 2009.

[13] L. Golab and M. T. Özsu. Issues in data stream management. *SIGMOD Rec.*, 32(2):5–14, June 2003.

[14] L. Golab and M. T. Özsu. Processing sliding window multi-joins in continuous queries over data streams. In *Proceedings of the 29th international conference on Very large data bases - Volume 29*, VLDB '03, pages 500–511. VLDB Endowment, 2003.

[15] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez. Streamcloud: An elastic and scalable data streaming system. *IEEE Trans. Parallel Distrib. Syst.*, 23(12):2351–2365, Dec. 2012.

[16] J. Kang, J. Naughton, and S. Viglas. Evaluating window joins over unbounded streams. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 341–352, 2003.

[17] G. Mencagli, M. Vanneschi, and E. Vespa. 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, 2013. Won the outstanding paper award.

[18] S. Muthukrishnan. Theory of data stream computing: where to go. In *Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '11, pages 317–319, New York, NY, USA, 2011. ACM.

[19] D. A. Schneider and D. J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. *SIGMOD Rec.*, 18(2):110–121, June 1989.

[20] J. Teubner and R. Mueller. 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, 2011. ACM.

[21] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 495–506, New York, NY, USA, 2010. ACM.

[22] S. Wu, V. Kumar, K.-L. Wu, and B. C. Ooi. 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, 2012. ACM.

[23] Y. Xing, S. Zdonik, and J.-H. Hwang. Dynamic load distribution in the borealis stream processor. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 791–802, 2005.