Università di Pisa
Dipartimento di Informatica

Technical Report: TR-13-18

Towards a Methodology for Parallel Data Stream Processing: application to Parallel Stream Join

Daniele Buono Tiziano De Matteis Gabriele Mencagli
Marco Vanneschi

November 27, 2013

address: Largo B. Pontecorvo 3, 56127 Pisa, Italy. tel: +39 050 2212700 fax: +39 050 2212726

Towards a Methodology for Parallel Data Stream Processing: application to Parallel Stream Join

Daniele Buono Tiziano De Matteis Gabriele Mencagli
Marco Vanneschi

November 27, 2013

Abstract

This paper deals with high-performance Parallel Data Stream Processing methodologies and implementation techniques. Data Stream Processing (DaSP) is referred to in the most general and interesting sense: on-line (often real-time) applications working on multiple, nondeterministic streams, with unlimited or unknown length and highly variable arrival rate, whose elements must processed efficiently "on the fly". Traditional high-performance solutions are not sufficient to meet the critical requirements of high throughput and low latency with acceptable memory size: typical DaSP applications require quite novel parallelism models, as well as related design and implementation techniques on the emerging highly parallel architectures. The aim of this paper is to give an original contribution to the design and implementation of parallel DaSP applications. The contribution is twofold: (1) the definition of an approach to a new general model for data-parallel DaSP computations according to a paradigm called Data Stream Parallelism, (2) the application of this approach to the parallel Stream Join problem, showing that the most interesting parallelizations in the literature are particular cases of our approach and that, compared to them, better throughput and latency are achieved by our implementation on multicore architectures.

1 Introduction

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

The topic of DaSP emerged by considering the execution of classic database queries on systems supplied by potentially infinite streams of data. When input data sets are continuous streams, some database operators become difficult to

be efficiently implemented compared to static relational tables as in traditional database systems. Notable examples are aggregate functions and join operators. While the former are mainly based on a single input stream of tuples, in the latter the problem is complicated by the presence of multiple streams and the symmetric nature of joins. Several software infrastructures enabling continuous query execution have been proposed over the last years [1, 11, 10, 8, 5, 15].

Some remarkable works [19, 6] have studied DaSP according to a computational and algorithmic modeling viewpoint, highlighting that this topic, besides being a fascinating and challenging research one, can be of strategic importance for data- and communication-intensive applications.

As known, the response time of a service is the sum of the waiting time for receiving the service and of the service latency. The former can be minimized by properly increasing the service throughput, thus high throughput is important anyway. However, this is not sufficient, because in some data-intensive computations the latency component might be much greater (even orders of magnitude greater) than the waiting time. As an example, in algorithmic trading applications [6, 11] such performance feature is fundamental to detect real trading opportunities. Nevertheless, some notable parallel DaSP proposals in the literature privilege throughput only [7, 12, 22]. The problem of designing and implementing high throughput and low latency DaSP applications - at the same time satisfying requirements of memory and power consumption constraints - is a quite complex one per se and because of the presence of multiple, nondeterministic streams, with unlimited or unknown length and highly variable arrival rate. *Traditional high-performance solutions, or their simple variants, are not always sufficient to solve this problem.* Stream processing has been intensively studied and applied in parallel computing, however by assuming that operations are independently applied to distinct elements of a stream, or to single corresponding elements of multiple streams in a data-flow fashion. Aggregate functions, join operators, complex correlations and other typical computational patterns of DaSP applications require *quite novel parallelism models* and related design and implementation techniques on the emerging highly parallel architectures (multi-/many-core processing elements combined in large systems and heterogeneous clusters).

Another important research issue is that even the traditional parallelization problems on permanent "in-memory" data structures can now be modeled as *equivalent parallel DaSP computations*: the multiple stream elements are obtained through the decomposition of finite or infinite data structures generated in some system nodes and to be processed nondeterministically by other nodes "on the fly". This way of representing computations is more general, and: *i)* it is useful per se, because it offers a further optimization chance as far as communication and processing bandwidth are concerned, *ii)* it is consistent with the characteristics of several important applications, in which data structures cannot be stored/operated as a whole, and/or their stream representation is more flexible and powerful for system resource exploitation, memory, real-time requirements, software modularity and portability.

The aim of this paper is to give an original contribution to the design of

parallel DaSP applications with stringent requirements of both throughput and latency. The contribution is twofold:

- the definition of an approach to a new general model for data-parallel DaSP computations according to a paradigm called *Data Stream Parallelism* (Section 3);
- the application of this approach to the *parallel Stream Join* problem, showing that the most interesting parallelizations, notably CellJoin [12] and Handshake Join [22], are particular cases of our approach and that, compared to them, better throughput and latency are achieved by our implementation on multicore architectures (Section 4).

Detailed evaluation experiments and state-of-the-art comparisons are described in Section 5.

2 Related Work

Several software infrastructures (Data Stream Management Systems - DSMSs) enabling continuous query execution have been proposed over the last years. Examples are are Borealis [1], Systems S [11], GigaScope [10], NiagraCQ [8] and STREAM [5] and more recently StreamCloud [15]. They are frameworks providing SQL-like languages and run-time supports to continuous query definition and execution.

Main issues in DSMSs are related to *memory usage* and *management*, and the performance of query processing. In the literature the first aspect has been studied through the engineering of data structures capable of providing succinct representations of received data feeds [19, 6] (e.g. histograms, sketches and synopsis) and by proposing optimized algorithms for stateful operators (e.g. stream joins and aggregate functions applied on temporal windows [9]). In both the cases the goal is to limit memory occupancy and I/O transfers, and to produce sufficiently accurate results. Other strategies like load shedding [23] have been used both to mitigate the pressure of high traffic rates and to save memory. However, they can be used only if the application semantics allows to loose data or to sample input streams.

The solution to the performance problems of DSMSs relies on parallelization. As a first case, we can exploit parallelism among operators of the same query and/or among different queries (i.e. *inter-operator and inter-query parallelism*), placing operators on the available computing resources. This approach has been followed by System S [4], in which data-flow graphs of queries are partitioned into sub-queries assigned to a set of parallel Processing Elements (PEs) mapped onto physical computing resources. A similar approach has been adopted in [27, 21], with special attention to load balancing mechanisms able to adapt the mapping between operators and resources according to the current stream rates and load variance. On emerging Cloud infrastructures the same problem has been addressed in [15].

Intra-operator parallelism is exploited when single operators are hot spots and deserve to be internally parallelized. The paradigm enabling this possibility is *data parallelism* [16, 24, 14]. It has been applied to streaming applications (and also some DaSP computations) often in an unsystematic fashion by relying on the simple idea that by using operator replicas we can parallelize the computation on different subsets of received data. As an example, parallelizations of stateful operators have been described in [26] using locks to access shared data, and in [25] for computations amenable to be parallelized using the MapReduce pattern. Nevertheless, the application of data parallelism to DaSP can be much more complicated especially when stateful operators involve a sliding (overlapping) window semantics [9]. An interesting studying 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.

Windows-join operators are critical in DaSP applications. They can be used to detect trends and find correlations between streams [17, 13]. Such computations can be difficult to be parallelized efficiently: because of the symmetric semantics of joins, effective partitioning/replication techniques 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 Cell Join [12] and Handshake Join [22], however the former scales well only for low-parallelism applications, while the latter has good scalability but has limitations for latency-sensible applications: they will be characterized in Sections 3 and 4. Also approaches like [25] on top of Hadoop MapReduce are based on data partitioning without latency evaluation. Parallelization proposals developed in the past (as a small example in [2, 20]) operate on static relational tables, and are not designed for performing windows-joins over unbounded input streams.

3 Parallel Data Stream Processing

In this paper we focus on the parallelization of processing modules executing a single DaSP operator with *multiple streams*. Rarely the execution of a DaSP module has a *data-flow* semantics. The general case is the *nondeterministic* semantics: the execution is enabled if at least one input stream contains a value, where the actual selection of the input stream is driven by a strategy applied to the module internal state.

Inside each module it is possible to apply specific efficient paradigms for describing the parallel computation. A very general and powerful structured programming paradigm is *data parallelism* [24, 14]. In the traditional "in-memory" view, it is based on the *replication* of functions among a set of functionally identical executors (*Workers* in the following), on the *partitioning* of the input-output data (a static/dynamic partition of each data structure is assigned to each Worker), and possibly on the replication of some data in all the

Workers. This paradigm can be defined according to a number of important variants [16, 24, 14], notably *map* (fully independent Workers) and *stencils* (a static/dynamic pattern of data dependence exists among the Workers computations to the Workers). For example, a sequential program for matrix-vector product $C[M] = A[M][M] \times B[M]$ can be transformed into a data-parallel program with parallelism degree N (N Workers), in which each Worker is assigned a distinct partition of $g = M/N$ elements of C and a distinct partition of g rows of A. If B is replicated in all the Workers, then a map computation is expressed. If B is partitioned too, then we have a stencil computation in which, at each step on j, a data dependence on $B[j]$ is established on the set of Workers (owing to the associative property of addition, a ring stencil pattern can be recognized).

At the expense of some implementation complexity - because of the variety of data parallel computations and because load balancing is not free, also depending on some abstract knowledge of the sequential computation form - data parallelism is able to meet important goals simultaneously: *processing bandwidth* (*throughput*)[1], *latency* and *memory occupancy* can be optimized w.r.t other paradigms (e.g. farm).

In the following sub-sections 3.1, and 3.2 we review some basic concepts and techniques for traditional in-memory stream-based parallelism. Then in sub-section 3.3 we propose our methodology for parallel DaSP applications.

3.1 Data Parallelism and in-memory stream-based computations

A complete treatment of data parallelism [14] is out of scope for this paper. We will extract the essential concepts and issues which are useful to introduce our proposal.

A high-level scheme of stream-based data-parallel computation (sketched in Fig. 1) consists in a collection of functionally identical Worker entities $W_1, \ldots, W_N$ delegated to the execution of the core computation. Input streams are interfaced by an *Emitter* entity (Data Distribution strategy) and the output streams are interfaced by a *Collector* entity. Emitter, set of Workers, and Collector interact in a pipelined fashion on distinct stream elements. The Emitter is responsible of the following actions:

- *data-flow* or *nondeterministic* acceptance of input data;
- *distribution of data* to the Workers according to a partitioning or to a replication strategy;
- *load balancing* of Worker computation.

The Collector accepts result partitions non-deterministically and produces output data structures according to their type. For example, result partitions are

[1] In the following we use the term (processing) "bandwidth" as a synonym of throughput.

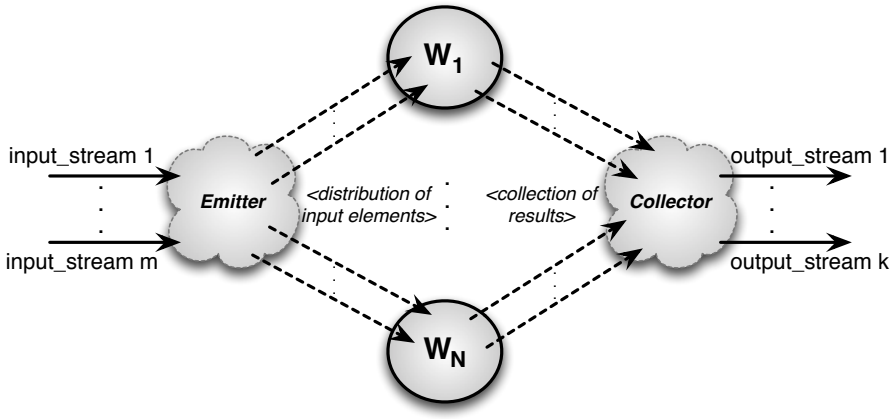


Figure 1: Abstract scheme of a generic Data Parallel computation.

gathered into the output data structure. Another well-known collection pattern is the *reduce* operator applied to all the result elements according to an associative function.

Either the Emitter or the Collector, or both, could be actually implemented in a *centralized* manner, or in a partially or fully *distributed* manner. For example, data partitioning/replication could be distributed among the Workers, with the Emitter acting as an interface for the data acceptance and, possibly, by providing some partial support to load balancing.

A notable distributed case is the *pipelined* data-parallel implementation [24], in which the pipeline stages correspond to distinct groups of loop iterations: data partitioning and replication are actually implemented through communications from one stage to the next. Although simple to be implemented and able to optimize both the throughput and the memory occupancy, the pipelined approach is characterized by the maximum *latency*. On the other hand, in a fully centralized solution the Emitter could easily become a throughput bottleneck when the degree of parallelism is high. In many cases, *proper map solutions with distributed data partitioning/replication exist which are able to optimize both throughput and latency*. Notable examples will be shown in Section 4.

3.2 Abstract data-parallel model and architectural dependent implementations

It is convenient to refer to an abstract model for data parallel computations, in which Emitter, Workers and Collector interact by means of *by-value communications*, e.g. message passing. As known, a message passing concurrent program is inherently independent of the underlying architecture (it is portable, in principle): provided that the proper run-time support is designed, a message-passing computation can be implemented/ported on a distributed memory cluster or on

a shared memory multiprocessor, or (perhaps the most interesting case for high parallelism) on a cluster of multi-processors.

In fact, the abstract message-passing description can be actually implemented in different ways which are more suitable for the underlying architecture. For example, let us consider *data partitioning* and *replication*:

1. in the abstract model we use *collective communication primitives*, notably *scatter* (for partitioning input data), *multicast* (for replicating input data), *gather* (for collecting results), as well as collective operations like *reduce*;

2. in the actual implementation for a *distributed memory architecture*, such functionalities are really implemented by message passing primitives (e.g. MPI), possibly with a run-time support able to overlap communication and calculation;

3. in the actual implementation for a *shared memory architecture* (e.g. a multi-/many-core component), communications can be implemented by reference. For example, passing the data structure pointer (capability) from the Emitter to the Workers could be sufficient for implementing the equivalent semantics of multicast/replication (all Workers access a single copy of the shared data structure) or of scatter/partitioning (each Worker knows the partition of the shared structure according to an index function). Analogously, stencil communications from Worker W_i to W_j can be implemented by a synchronized direct access of W_j to the variable modified by W_i (often, in order to synchronize all the Workers, a barrier is roughly used).

In conclusion, the by-value communication abstract model is general and is able to capture all the needed features of a data-parallel program. Once a "good" abstract solution has been designed, the architecture-dependent run-time support implementation, though being a complex issue, can be faced according to the state-of-the-art knowledge in parallel and distributed architectures. Subsection 3.3 will demonstrate the validity of this approach for DaSP applications.

3.3 Data Stream Parallelism

We propose the *Data Stream Parallel* paradigm which extends the traditional data-parallel one with powerful features and mechanisms for structured parallel DaSP. The main features concern *windowing management* and *data distribution* according to a limited set of solutions with a clear semantics and able to optimize some or all performance metrics of interest (throughput, latency, memory).

3.3.1 Windowing methods

in DaSP, the concept of windows is applied according to at least the following modes [7, 9]:

a) *tumbling*, or non-overlapped, vs *sliding*, or overlapped, windows. In the former case a discrete timing model is assumed in order to fully separate the computation on a data window $DW1$ from the computation of the next data window $DW2$, i.e. , i.e. $DW1$ expires when the computation on its elements terminates. With sliding windowing, a continuous timing model is assumed in which, according to some overlapping degree, expiring is applied to individual/subsets of stream elements, not to whole windows;

b) *count-based* vs *time-based* windows, in which the elements belonging to a data window are, respectively, a given number of consecutive stream elements or are the elements received within a given time interval T_w, called *window length*. Often the time-based mode implies the utilization of the most recent values, and stream elements are associated a unique identifier in the form of *timestamp*, i.e. each stream element is a pair (value, timestamp).

All the four combinations of a) and b) modes are possible. However, *the semantics of specific computations is captured (or better captured) by one of them.*

When streams are derived from the decomposition of in-memory data structures, the tumbling windows semantics is quite natural, with the count-based or the time-based mode depending on the data structure type. Many aggregate functions and join operators of DSMSs are notable examples of the sliding window mode and, most frequently associated to a time-based semantics. In Section 4 we will study the Data Stream Parallel implementation of stream join computations according to the time-based sliding window mode. In general, this seems a frequent case of computations in which streams are not derived from the decomposition of in-memory data structures.

3.3.2 Abstract module structures for DaSP

as said, the most interesting acceptance strategy of input data (Emitter functionality) is the nondeterministic one. The output stream values can be produced in a different order with respect to the corresponding input values, i.e. Collector is not in charge of reordering the results. If the time-based mode is applied, the timestamps associated to the output values are sufficient for the destination module being able to correctly utilizing the received results. However, also in count-based computations it is easy to utilize unique identifiers dependent on the computational problem semantics (e.g. array indexes).

Important issues in the abstract model are: (i) *data distribution and collection management*, (ii) *windowing management*. Two main models can be recognized:

a) **centralized model**, also called *Agnostic Workers*;

b) **distributed model**, also called *Active Workers*.

In the centralized model, for each stream the nondeterministic Emitter is in charge of:

- performing all the processing and storing actions needed to build and to update the data window for each stream, according to the tumbling/sliding and count-based/time-based semantics;
- distributing data windows to the Workers according to the *partitioning* or to the *replication* strategy of the paradigm application, in general taking into account load balancing and other optimizations too.

In this model, Workers are *agnostic* of the window management and data distribution, i.e., they are just in charge of applying the computation to the received window data.

In the distributed model, sliding window management and data distribution are partially or entirely delegated to the Workers. They receive elementary data values, or data window segments, from the Emitter and are in charge to performs actions to build and to update the data window. In other words, now Workers are *active* on sliding windows management too, thus their code is very similar to (coincides with) the sequential version. Emitter and Collector acts mainly as intelligent interfaces w.r.t the Worker strategies. It might be convenient to have a partially distributed version, by delegating a few tasks to Emitter (and Collector) in order to simplify, or to render more efficient, some actions. Notably, some load balancing strategies could be executed directly by the Emitter, e.g., according to round-robin or on-demand techniques.

Being a distributed version, Workers can take part to the distribution strategies themselves too, i.e., they can implement distributed forms of partitioning (scattering), replication (multicasting) and collection (gathering), as well as collective operation, like reduce, in a distributed mode.

The application of the centralized and distributed model depends on the computation algorithm and on the windowing modes. As it happens in the traditional data-parallel model, some optimizations can be introduced depending on the specific semantics of the sequential computation. For example, for count-based windows aggregate functions, the *pane* method [7] can be applied in order to optimize the memory occupancy by reducing the replication degree of stream elements belonging to the common part of consecutive windows.

The application of centralized and distributed models to the Stream Join problem will be studied in Section 4.

3.3.3 Optimal Parallelism Degree

let us consider a Data Stream Parallel computation acting on multiple streams $X_1, \ldots, X_m$ with inter-arrival rates $\lambda_1, \ldots, \lambda_m$ respectively. The *maximum bandwidth* (throughput) can be expressed as a function of the following type:

$$B^{\omega}_{max} = B^{\omega}_{max}(\lambda_1, \ldots, \lambda_m, \omega, \Gamma) \tag{1}$$

where ω is a set of parameters related to the windowing semantics (e.g. window length) and Γ is a set of parameters characterizing the sequential algorithm. For example, in Stream Join, ω is the temporal window length T_w and Γ is the hit rate p of the join operator.

One goal of a parallelization is to achieve the maximum bandwidth at *steady state*, i.e. after and before the initial "filling" and the final "emptying" transient phases of the steaming execution. To this goal, the *optimal parallelism degree* N_{opt} is the minimum number of Workers such that the output rate of results is equal to the maximum bandwidth.

4 Parallel Stream Join on Multicore Architectures

As seen, Data Stream Parallelism is potentially able to optimize *both throughput and/or latency* by selecting and instantiating the best abstract model among a very limited number of structured versions, i.e., centralized, pipeline-distributed, partially distributed with linear or multidimensional topology. In this section we apply the methodology of Section 3 by discussing a novel parallelization of join operators on data streams and comparing it with the current state-of-the-art parallel implementations. We remark that out goal are *both* high throughput *and* low latency parallel join implementations.

We adopt a *time-based sliding window* semantics according to the basic procedure referred as Kang's algorithm, originally described in [17]. The description and our parallelization proposal are valid for *an arbitrary number of input streams* and for *time-based and count-based windows*. Just for clarity reasons we refer to the basic case of two streams with a time-based window semantics.

Given two streams X and Y, for each new arrival from stream X the algorithm consists in the following actions:

1. scan the Y-window, evaluate the join predicate and propagate the results;
2. insert the new tuple into the X-window;
3. invalidate all the expired tuples from the X-window.

This sequence is performed symmetrically for the other stream.

The generic procedure for the Kang's algorithm is a *nested-loop evaluation* in which all the possible joining tuples satisfying the window constraints are firstly enumerated and then filtered according to the join predicate. For certain join predicates (e.g. equi-join) 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 of tuples. In the following we suppose a generic window-join problem on streams without specifying the predicate and which kind of data structures is effectively used. In fact, as we will see, our abstract parallelization design will be completely agnostic w.r.t the specific implementation of the Kang's algorithm.

Let us denote by R the stream of joined results, T_w the window length, and t_x and t_y the timestamps of two tuples x and y. Formally, the join condition can be defined as follows:

Definition 4.1 (Time-based Window-Join Semantics) *given two tuples x and y satisfying the join predicate, the pair (x, y) is in the output stream R iff:*

- *if x is older than y, then the timestamps must respect the following condition: $t_x \geq t_y - T_w$;*
- *otherwise x must be in the current X-window when y arrives, i.e. $t_y \geq t_x - T_w$.*

The latency is evaluated as follows:

Definition 4.2 (Latency) *let two tuples x and y with timestamps t_x and t_y. Let us suppose that the two tuples join and the output result $r = (x, y)$ is produced at time t_r. The latency l_r is given by:*

$$l_r = t_r - \max\{t_x, t_y\} \tag{2}$$

If we can assume that the calculation time of the join operator is constant (or featuring small variance), thus load balancing can be achieved by assigning to each Worker the same number of comparisons between tuples (i.e. the same number of tuples in the current windows).

4.1 Centralized solution

Let us consider what happen when a new element is received from one of the streams (i.e. x_i from stream X). We need to apply the join predicate between x_i and any element inside the current window of Y, which is well defined: it contains all the elements of stream Y received in the last T_w interval, and it has been partitioned by the Emitter among the set of Workers. Value x_i is multicasted to the Workers. Each Worker applies step 1 of the Kang's algorithm. Of course, when we receive an element from the stream Y (i.e. y_i) the situation is reversed. The Emitter is also responsible of inserting and removing elements from windows, i.e. executing steps 2 and 3 of the algorithm, so that Workers are completely agnostic of the sliding nature of windows. An abstract representation of this parallelization is sketched in Fig. 1.

This parallelization has been successfully used in CellJoin [12]. At the arrival of each element (x_i), the Emitter must: *(i)* determine the partitioning of a window (Y); *(ii)* scatter the window (Y) to the set of Workers; *(iii)* multicast the element (x_i) to all the Workers; *(iv)* insert the element (x_i) in its window (X); *(v)* remove the expired tuples from the corresponding window (X).

As known from the methodology of Section 3, this solution is able to optimize throughput *provided that* the Emitter has a sufficiently low service time to sustain the input streams rate. The data distribution overhead (actions *ii*, *iii*) can be mitigated on shared memory architectures by allocating windows contiguously and sending pointers to the correct areas, instead of actual values

(see Section 3.2). If determining a balanced partitioning at each arrival becomes a problem (action i), the Emitter can use the same partitioning on subsequent elements (of the same stream) and adapt it every once in a while to re-balance the amount of elements per partition. Nevertheless, actions iv and v cannot be further optimized and can possibly make the Emitter a throughput 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 architecture (from which the name derives) by placing the Emitter on the single PPE and the Workers on the 8 SPEs. In this case, given the very limited number of Workers, the Emitter does not become a bottleneck and the application scales for low parallelism degrees with low latency.

4.2 Distributed pipeline-based solution

To solve the throughput and scalability problems for highly parallel join applications, we study solutions based on the distributed model able to minimize the Emitter overhead.

The *pipeline data-parallel solution* is a potential candidate, with a fully distributed window partitioning and replication and window 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 [22] with the *Handshake Join* parallelization: the two streams flow in the opposite directions of the pipeline, as depicted in Fig. 2. In this way each element of a stream will, sooner or later, encounter any element of the other stream, effectively allowing a fully distributed parallelization of the Stream Join problem. We denote by X-i and Y-j the i-th and the j-th partition/segment of X-/Y-window.

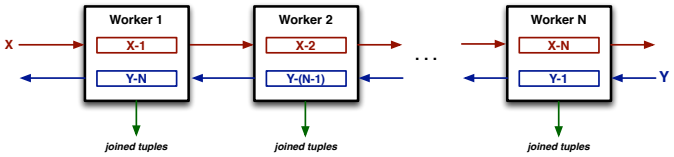


Figure 2: Abstract scheme of the Handshake Join parallelization.

As known in general, and as it is quite evident in this case, the throughout and scalability advantages are achieved at the expense of a too long latency that becomes a complex function of the expiring strategy. Thus this solution is not acceptable for latency-sensitive applications.

4.3 Distributed solution: proposed approach

We propose a novel parallelization that encompass the advantages of CellJoin and, at the same time, allows the definition of a lightweight, scalable (w.r.t the number of Workers) Emitter that removes the bottleneck problems of CellJoin

and allows extremely low latency compared to Handshake Join. The methodology of Section 3 is applied to find the best combination of partitioning and replication in a Data Steam Parallel module, at the same time being able to apply efficient architecture-dependent implementation techniques.

We use a distributed model with a very low number of functionalities left to the Emitter. The main idea is to introduce an Emitter that distribute tuples to the set of *Active Workers* (unlike CellJoin). On the other hand, we do not want to excessively increase the computation on the Workers, that are now in charge of collectively handling insertion and removal from the windows. This is possible by exploiting a tuple distribution that: (i) fixes the owner of each tuple to a (a set of) Worker(s), (ii) lets each Worker remove expired tuple independently, so that the Emitter is only responsible of selecting the destination Worker for each element of the streams, while each Worker handles its local window, by using the very same computation of the sequential join.

We partition a window (Y) and replicate the other (X), so that each element received from stream Y is sent to only one Worker, while elements received from stream X are multicasted to all N Workers. The correctness can be easily verified by ensuring that each tuple of one stream will be compared with all the elements of the other window:

- a tuple y_i received from stream Y is forwarded to a specific Worker. Given the replication of the other stream, such Worker contains the full X-window, and can perform the entire join on y_i;
- a tuple x_i received from stream X is forwarded to all the Workers. Each Worker contains a distinct partition of Y and performs a join on the local window (similarly to CellJoin). The cumulative results of all the Workers represent the entire join on x_i.

In order to achieve load balancing despite variability of tuple expiring timing and stream rate, a simple yet effective policy is to distribute elements in a round-robin fashion, i.e. y_i is forwarded to W_j with $j{=}(i \bmod N) + 1$. In this way new elements are evenly distributed (independently of the current rate) and, given the expiring condition, evenly removed. Using this distribution the two windows are not partitioned in contiguous segments, but each Worker has it own set of tuples assigned in an interleaved fashion.

To further limit the Emitter service time, we can distribute the multicast of X tuples (i.e. Emitter forwards x_i to W_1, that forwards it to W_2, and so on), so the Emitter only needs to forward each received tuple to one of the Workers, independently of the number of Workers in the parallel program. Fig. 3 illustrates a solution with a *linear set* of Workers.

This solution is still partially affected by latency problems. When an element is received on X, each Worker compute the join on a partition of Y, therefore splitting the amount of comparison among the whole set of Workers, and reducing latency w.r.t the sequential version. However, when an element is received on Y, the whole join will be computed by a single Worker, therefore achieving a latency comparable to the sequential version.

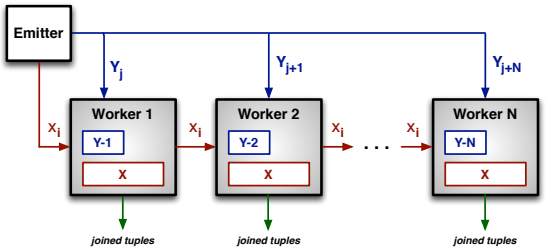


Figure 3: Linear set of Workers exchanging X tuples.

We know from the general theory of Section 3 that proper solutions exists, able to minimize latency too, while maximizing throughput. In our case, this is achieved by generalizing the Worker organization as a mix of partitioning and replication for *both* windows, considering a *square* or, more generically, *rectangular layout* as in Fig. 4.

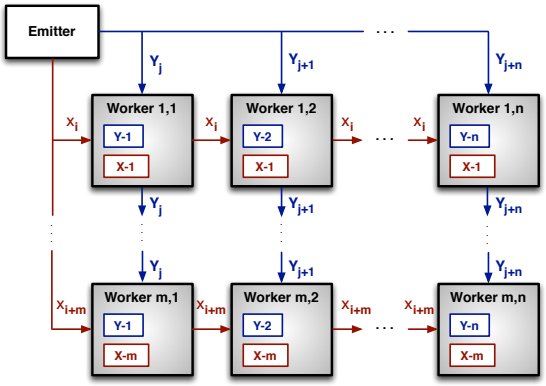


Figure 4: General layout of Workers using m rows and n columns.

We maintain the same partitioning of Y among columns, and further partition X among rows (for correctness the same considerations apply). For a square layout this comes at the (little) cost of not being able to use any number of Workers. The rectangular layout has not such restriction and still allows a good balanced (but not optimal) latency.

From the latency point of view a rectangular layout is also interesting in the case of *asymmetric streams* (i.e. X have an higher rate than Y or vice-versa), because it is able to balance the partition size of the two windows (in terms of elements) per Worker (this fact will be clarified in Section 5.3).

4.4 Implementation and Optimizations

The abstract parallelization described in Section 4.3 has been implemented on general-purpose shared memory multi-/many-core CPUs. The implementation relies on several optimizations related to:

- how data structures are organized in memory;
- how replication has been really implemented;
- efficient re-definition of the Kang's algorithm performed by Workers.

4.4.1 Attribute-oriented organization

each received tuple consists of a set of attributes $a_1, a_2, \ldots, a_k$. Two different ways to store tuples in memory can be identified: (1) *tuple-oriented organization*: different tuples are stored in contiguous regions of memory; (2) *attribute-oriented organization*: we use k separated data structures to contiguously store the same attribute of different tuples.

As stated in [12], the second approach should be preferred to reach better performance. It can reduce the amount of *cache* lines transferred to apply the join predicate over the entire window. 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 vector instructions. Due to these reasons, we use this organization in the implementation of our parallel join version.

Our version exploits replication and partitioning of the two stream windows onto a generic matrix structure of Workers. Workers belonging to different rows/columns are assigned 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. 5 supposing two attributes per tuple and a square structure of nine Workers.

4.4.2 Support to replication

replication yields to a higher memory occupation and Workers need to propagate each received tuple by value. On today's multi-/many-core machines, replication can be implemented efficiently from the memory occupancy viewpoint by exploiting *sharing* instead of pure replication of data-structures. Let us focus on how replication of X-window segments is realized (symmetrically the same principle applies to Y-window). Workers belonging to the same row have a replica of the same segment of X-window. By resorting on shared memory, the segment is physically shared by Workers on the corresponding row: such Workers 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

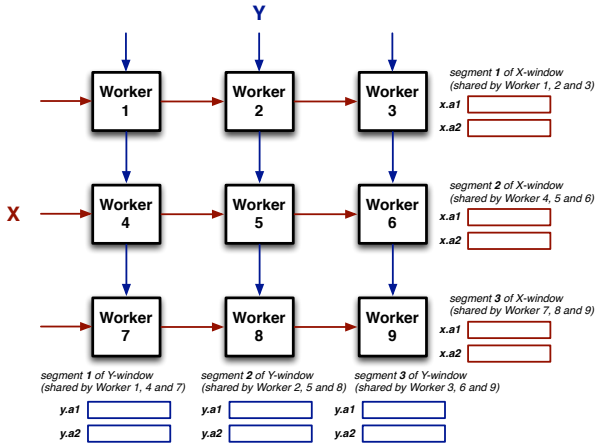


Figure 5: Data organization between Workers. X/Y-windows are replicated (shared) and partitioned among Workers.

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 replica of window segments, the memory occupancy of the parallel program will be similar to the sequential version (only two windows should be maintained and updated);
- sharing makes it possible to exchange smaller messages between Workers instead of real tuples. This result leads to further improvement in latency, since Workers on the same row/column start their join evaluation earlier compared with a version with pure replication and communication of entire tuples.

Such way to perform replication has important impacts on the way in which expiring tuples are identified and really removed from windows. In the abstract communication-based model, each Worker is responsible to remove expired tuples in its window segments. When segment sharing is applied, this activity must be performed in a careful way. In fact only one Worker effectively remove expired tuples form a shared segment and this action must be performed *iff* expired tuples are no longer necessary to all the other Workers sharing the same segment of the window. We manage this fact with a low complexity procedure:

- each Worker maintains a counter to the first valid tuple for each of its 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 from the window segment. This is performed periodically by identifying expired tuples for all the Workers sharing the same window segment (i.e. by taking the minimum between Worker counters).

4.4.3 Sequential algorithm improvement

the final implementation detail is related to the improvement of sequential algorithm performed by Workers. In the basic Kang's algorithm, at each reception of a tuple from stream X (and symmetrically from Y) we scan the Y-window to find joining tuples, next the new tuple is added to X-window and then the X-window is scanned to find expired tuples. In our version Workers optimize this procedure by avoiding to scan two window segments at each reception of a new tuple. After receiving a new tuple x, a Worker: (1) insert x in its segment of X-window; (2) scan its segment of Y-window to find tuples joining with x; (3) determine the number of expired tuples from Y-window by updating its counter correspondently.

Note that we look for expired tuples in the opposite window of the received element (differently from the Kang's algorithm). The result is that only one scan of a window segment is needed for each received element. Expired tuples in X-window will be determined only when a tuple from step Y is received. In this way we can reduce the Worker processing time with great advantages in terms of throughput and latency and respecting the correct join conditions provided in Definition 4.1 (tuples are always compared according to their timestamps before computing the join result).

5 Experiments

In this section we provide an experimental evaluation of achieved throughput and latency by our parallel solution, referred as *DP-Join* in the sequel[2]. Furthermore, we compare our approach with the most recent existing parallelization [22] - Handshake Join[3]. In Section 4.1 we saw that CellJoin [12] is a notable example of centralized implementation with low parallelism, while in this section we are interested in studying solutions without parallelism degree constraints.

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

[3]We thank the authors of [22] for having made available their source code at the current url: http://people.inf.ethz.ch/jteubner/publications/soccer-players/

5.1 Experimental setup

We study the same join problem described in [22] and [12] consisting in a *band-join* predicate over two streams X and Y. Two tuples x and y (in their respective windows) join *iff* the following condition holds:

```
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 join attributes. Their values are generated in order to reproduce 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 Section 4.4). Tuples and their timestamps are pre-generated according to the input stream rates and read from file before each test execution.

DP-Join is implemented on a commodity Intel multi-processor architecture composed of two Xeon E5-2650 CPUs for a total of 16 cores and 32 hardware SMT contexts (HyperThreading). 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 using a "by reference" model (i.e. passing pointers to shared data as described in Section 3.2) using efficient lock-free queues made available by the `FastFlow` library [3]. We set the CPU-affinity of each thread on a corresponding SMT (Simultaneous Multi-Threading) context of the underlying architecture.

5.2 Bandwidth analysis

As introduced in Section 3.3.3, the optimal parallelism degree N_{opt} is the one able to achieve the maximum bandwidth B^{ω}_{max} expressed as a function of the input stream rates λ_x and λ_y, the window length T_w and the join hit rate p. 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 of stream X and Y respectively (we use the same window length T_w for both the streams), the number of joined tuples in that interval is given by:

$$\begin{aligned} \Omega_{t_0,t_1} &= \Big[\lambda_x \ (t_1 - t_0) \ \mathcal{W}_y + \lambda_y \ (t_1 - t_0) \ \mathcal{W}_x\Big] \, p \\ &= 2\lambda_x \, \lambda_y (t_1 - t_0) T_w \, p \end{aligned} \tag{3}$$

The steady state maximum bandwidth is evaluated as follows:

$$B^{\omega}_{max} = \frac{\Omega_{t_0,t_1}}{(t_1 - t_0)} = 2\lambda_x \, \lambda_y \, T_w \, p \tag{4}$$

5.2.1 Optimal parallelism degree and Scalability

Fig. 6 shows the optimal parallelism degree of different DP-Join layouts (square, rectangular and linear) for a problem consisting in two symmetric streams with

input rate 2100 tuples/sec and a window length of 300 seconds. For the sake of clarity we show the number of Workers per row and per column used in the rectangular layout. Square and linear layouts can be straightforwardly deduced from the parallelism degree. The square and the rectangular layouts have slightly greater optimal parallelism degrees (8 and 9 Workers) compared to the linear one (7 Workers), since they can not be used with any parallelism degree (a square layout exists only for perfect square parallelism degrees). Furthermore, the results confirm the accuracy of Expression 4: the measured maximum bandwidth and the predicted one differ less than 2%.

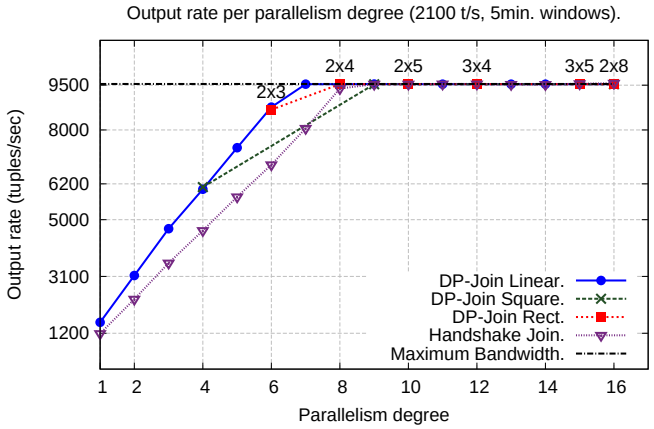


Figure 6: Optimal parallelism degrees of different layouts (DP-Join) and of the Handshake Join parallelization.

Fig. 6 gives us also an insight into the *scalability* of the parallel versions. Passing from 2 to 4 Workers in the linear layout the offered bandwidth approximately doubles (from 3100 to 6200 tuples/sec). A more complete scalability analysis is depicted in Fig. 7a using symmetric streams with a high input rate (6000 tuples/sec). By definition, the scalability with $n \times m$ Workers is:

$$\mathcal{S}^{(m \times n)} = \frac{B^{\omega}_{m \times n}}{B^{\omega}_{1 \times 1}} \tag{5}$$

For brevity we show the results for a linear layout of Workers (similar results can be achieved by the square and rectangular layouts). With the used configuration in terms of input rates and window length we have not sufficient cores to reach the maximum bandwidth of the problem, i.e. the offered bandwidth is lower than the maximum one $B^{\omega}_{m \times n} < B^{\omega}_{max}$. Nevertheless, *DP-Join is characterized by a near-optimal scalability.* With 15 and 16 Workers we exploit the SMT capabilities of the CPUs by allocating the Emitter and the Collector threads together with 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` user-level synchronization mechanisms (based on aggressive busy-waiting); to avoid this problem, we adopt more SMT-aware

synchronization techniques (i.e. based on busy-waiting with a slight delay) able to mitigate the overhead of the two support threads reaching a near-optimal scalability up to 16 Workers.

To provide a correct comparison with Handshake Join, we execute the two versions on the Intel multi-processor using the same data-set of input tuples with identical timestamps. Compared with Handshake Join, *DP-Join uses a faster sequential algorithm*: the difference is of 24% using 1 Worker as we can observe from Fig. 6. This is mainly due to our optimized version of the Kang's procedure (see Section 4.4), in which we perform a single scan of one of the two window segments for each received element (instead of both as in the classic procedure). Nevertheless, from the scalability viewpoint (Fig. 7a) the two implementations behave very similarly. The performance loss of Handshake Join using 15 and 16 Workers is due to the presence of the Collector and the *Driver* threads [22] (the latter in charge of inserting and expiring tuples) and the use of synchronization methods not optimized for a SMT architecture.

In sub-section 5.3 we will see that layouts have very important effects on the latency. In principle, *we expect that, with the same parallelism degree, any $m \times n$ layout of DP-Join is characterized by the same offered bandwidth*, i.e. for $m\,n$ less than the optimal parallelism degree:

$$B^{\omega}_{m \times n} = \frac{m\,n}{T_{\bowtie}}\,p \tag{6}$$

where $T_{\bowtie}$ is the average time to evaluate the join predicate on a pair of tuples from X and Y, i.e. each Worker evaluates the predicate every $T_{\bowtie}$ and produces a result (joined pair) every $T_{\bowtie}/p$. This expectation is confirmed by our experimental results. Table 1 shows the offered bandwidth using the same stream configuration of Fig. 7a (stream rates and window length) and three notable parallelism degrees equal to 6, 9 and 16 Workers, such that the offered bandwidth is smaller than the maximum one. With 6 and 16 Workers we use a 2×3 and a 2×8 rectangular layout.

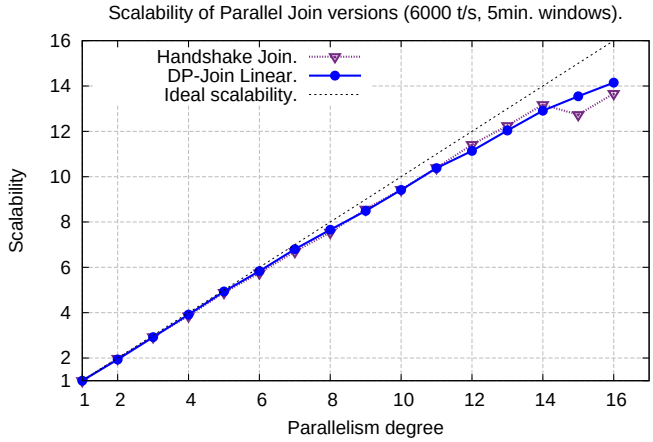
Par. Degree	Square Layout	Rect. Layout	Linear Layout
6	-	9,301	8,529
9	14,047	-	13,599
16	21,740	21,700	21,659

Table 1: Offered bandwidth (tuples/sec) by different layouts (DP-Join).

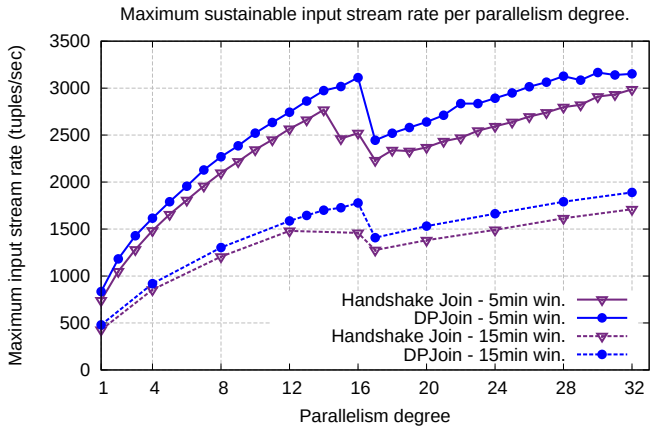
Actually, there are small differences between layouts (in particular, the linear layout provides a slightly lower performance), because of propagation delays between Workers (neglected in Expression 6).

5.2.2 Sustainable input rate and Multithreading

we propose a different bandwidth analysis, by reproducing the experiment described in [22]. For each parallelism degree we measure the maximum input rate



(a) Scalability.



(b) Maximum sustainable input rate (symmetric streams).

Figure 7: Bandwidth analysis: (a) scalability of DP-Join (linear layout) and Handshake Join; (b) maximum sustainable input rate (per stream) using different parallelism degrees.

(per stream, in the symmetric case) sustainable by the parallelizations. Results are depicted in Fig. 7b using the SMT facility, where we study the linear layout using up to 32 workers. We consider this layout because all the possible parallelism degrees are allowed by it, providing a complete trend of the parallelization behavior. We consider two different window lengths of 300 seconds and 900 seconds (the maximum length discussed in [22] and in [12]) .

As expected, using a windows length of 900 seconds we observe the same behavior: the results are just numerically lower, since longer windows imply a coarser grain problem. A significant slowdown is present when we start to allocate multiple Workers on the same cores using more SMT contexts: as we use

larger 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.

As seen in Fig. 7a, our parallelization outperforms Handshake Join. In Fig. 7a and 7b we can note a significant drop in performance using 15 or 16 Workers with Handshake Join, due to the presence of the Emitter and the Driver process. Differently to our parallelization, Handshake Join exhibits a slight performance improvement by using more Workers per core. The main reason for this is probably due to the performance drop using 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.

5.3 Latency analysis

A proper selection of the best layout of DP-Join, although it does not influence the offered bandwidth, *is of great importance to minimize the average latency.* The concept of latency for a stream join computation has been stated in Definition 4.2.

We consider a simple yet useful approximation of the average latency of a generic $m \times n$ layout of Workers, that allows us to qualitatively understand the effects on the latency. At each reception of a tuple x from X, the Emitter schedules the new tuple to a row of Workers. Each Worker applies in parallel the join evaluation on its Y-window segment. By assuming that joining pairs of tuples are uniformly distributed, the average latency is given by:

$$L^x_{m\times n} \simeq \frac{\sum_{i=1}^{\mathcal{W}_y/n} i\, T_\bowtie\, p}{p\, \mathcal{W}_y/n} \simeq \frac{\mathcal{W}_y\, T_\bowtie}{2\, n} \tag{7}$$

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:

$$L_{m\times n} = p_x\, L^x_{m\times n} + p_y\, L^y_{m\times n} \tag{8}$$

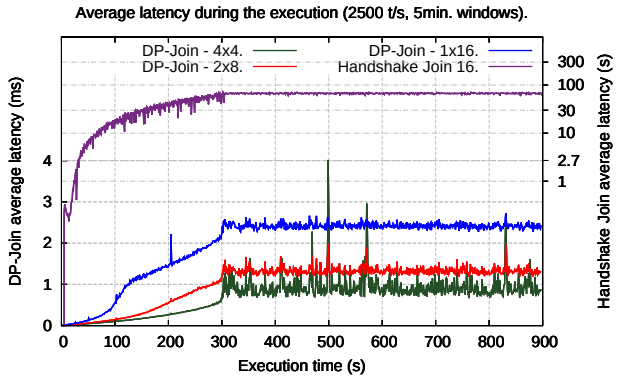
In the case of symmetric streams we can assume the same probability $p_x = p_y = 0.5$. The previous expression can be rewritten in the following way:

$$L_{m\times n} = \frac{\lambda\, T_\bowtie\, T_w (m+n)\, p}{4\, m\, n} \tag{9}$$

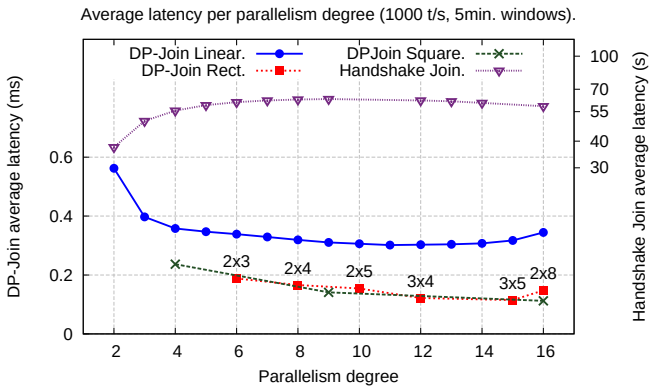
By comparing layouts using the same total number of Workers ($m\, n$), we observe that the layout with the lowest latency is the one that minimize $m+n$, i.e. the square one. This observation states an intuitive fact of our parallelization: in the case of symmetric streams, to minimize the average latency it is important to equally lower both the terms of Expression 8.

5.3.1 Symmetric streams

To provide a real validation of this behavior, we perform an experiment using symmetric streams of rate 2500 tuple/sec and a window length of 300 seconds. The latency evaluation is meaningful when the system is not a bottleneck, i.e. when $B^{\omega}_{m\times n} = B^{\omega}_{max}$. Fig. 8a shows the average latency of different layouts (with the same parallelism degree) during the execution. We can easily observe that the steady-state phase starts after 300 seconds, i.e. when the two windows reach their maximum size in terms of elements. *The square layout provides the best latency (*0.95 *ms on average).* The average latency is 60% better with the linear layout (2.42 ms), while the rectangular one provides a latency between the other two layouts.



(a) Measured latency (total execution length of 900 seconds).



(b) Latency with symmetric streams.

Figure 8: Latency analysis: (a) average latency over the execution; (b) average latency with different layouts and parallelism degrees in the case of symmetric streams.

Fig. 8b presents a more complete evaluation using different parallelism degrees. We consider streams of 1000 tuple/sec and we take into account only the layouts 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 one minimizing latency*. 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 16 Workers. In that case two Workers are executed on the same core of the Emitter and the Collector threads resulting in worse results in terms of latency.

Our DP-Join approach represents a clear improvement w.r.t the actual state-of-the-art. In [22] Handshake Join has not been analyzed in terms of latency. To do that, we have slightly modified their source code in order to collect such measurements. The latency comparison is summarized in Fig. 8a and 8b by showing the results of Handshake Join and different layouts of DP-Join, which *provides an average latency at least four orders of magnitude smaller*. To the sake of clarity, we show the Handshake Join latency using a different (logarithmic) scale represented in the right part of the plots. The reason for this great difference is related to the rationale behind this parallelization. A tuple x assigned to a Worker leaves its partition only when it becomes the oldest one in that segment and Workers exchange tuples (between neighbors) only to balance their workload (the size of their window segments). Each tuple stays in a segment for T_w/N on average, with N the number of Workers. This means that the speed at which tuples are exchanged depends entirely on the stream rate instead of the Worker performance. On the contrary, in our approach, at each reception of a new tuple *all* the comparisons are performed simultaneously (neglecting the propagation times for Worker communications).

5.3.2 Asymmetric streams

To conclude, we show a last experiment using *asymmetric streams* characterized by significantly different input rates (Fig. 9). The first one, X, with 1000 tuple/sec and Y with 3000 tuple/sec. In this case our analytical studying 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*. This intuition is confirmed by the results shown in Fig. 9: the rectangular layout is now the one giving 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 worth noting the behavior of the linear layout, where the latency increases as we use higher parallelism degrees. This fact can be justified by the propagation delay of received tuples along the pipeline. This aspect, not captured by the previous qualitative analysis, deserves to be investigated more deeply in our future work.

Finally, Fig. 9 highlights that the average latency of Handshake Join is several orders of magnitude higher than our DP-Join parallelization also in the case of asymmetric streams.

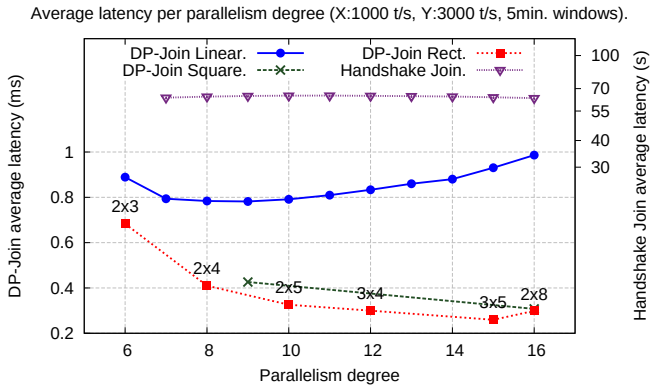


Figure 9: Average latency with different layouts and parallelism degrees in the case of asymmetric streams.

6 Conclusion

The recent interest in DaSP enables a new class of problems that can not be straightforwardly parallelized using the traditional high-performance solutions. In this paper we introduce the basis of a novel methodology to assist the programmer during the design and implementation phase of a parallel DaSP computation. We propose a general approach that, by extending the common concepts of data parallelism and stream parallelism, proposes a unified view of continuous query computations, like in DSMSs, and computations defined on in-memory structures but executed on streams. We identify data distribution/collection and windows management techniques as the most important issues to achieve scalable solutions providing satisfactory results in terms of throughput and latency. For this reason, we propose scalable distributions and window management strategies based on distributed/decentralized models and implementations.

To demonstrate its effectiveness, we have applied our approach to a critical benchmark, i.e. the window-based join problem. The first notable result is that current state-of-the-art implementations, i.e. CellJoin [12] and Handshake Join [22], are captured by our methodology; nevertheless other interesting parallelizations are possible. In fact, we introduce a novel parallel implementation that effectively overcomes the performance problems of previous works, offering higher throughput and lower latency at comparable parallelism degrees. We also introduce a further degree of freedom, i.e. different layouts depending on the windows partitioning/replication methods.

Another contribution is the experimental study of two important, yet previously uncovered, aspects: the average *latency* of the stream join and its behavior w.r.t *asymmetric stream rates*. We show that the two aspects are interrelated: the best result in term of latency is obtained by using different layouts, depending on the symmetricity or not of stream rates.

Several improvements can be made on our implementation, both in the algorithmic sense (e.g. using more than two streams, sequential code optimizations, use of SIMD instructions) and from the experimental standpoint (parallelization on shared and distributed memory architectures). Finally, starting from the results of Section 5, the definition of a *performance model* will constitute an important advancement in determining the optimal parallelism degree and layout able to achieve the desired levels of throughput and latency.

While sufficiently mature to be applied on real applications, the proposed approach to parallel DaSP applications needs further development. In particular, we aim at studying other stream problems to further prove the generality of the approach, and a detailed and formal study on the concept of windowing and its general applicability to different algorithms.

Finally, considering the high-variability of input rates and workload intensity in DaSP applications, the approach is worth to be studied in terms of *dynamic adaptiveness*, i.e. providing run-time supports and adaptation strategies [18] able to dynamically change the parallel structure in terms of distributions, windowing methods and layouts.

References

[1] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Mitch Cherniack, Jeong hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, Er Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The design of the borealis stream processing engine. In *In CIDR*, pages 277–289, 2005.

[2] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *Proc. VLDB Endow.*, 5(10):1064–1075, June 2012.

[3] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Massimiliano Meneghin, and Massimo Torquati. An efficient unbounded lock-free queue for multi-core systems. In *Proc. of 18th Intl. Euro-Par 2012 Parallel Processing*, volume 7484, pages 662–673, Rhodes Island, Greece, aug 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, February 2011.

[5] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Rajeev Motwani, Itaru Nishizawa, Utkarsh Srivastava, Dilys Thomas, Rohit Varma, and Jennifer Widom. Stream: The stanford stream data manager. *IEEE Data Eng. Bull.*, 26(1):19–26, 2003.

[6] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer 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] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. Niagaracq: a scalable continuous query system for internet databases. *SIGMOD Rec.*, 29(2):379–390, May 2000.

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

[10] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav 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] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol 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] Buğra Gedik, Rajesh R. Bordawekar, and Philip S. Yu. Celljoin: a parallel stream join operator for the cell processor. *The VLDB Journal*, 18(2):501–519, April 2009.

[13] Lukasz Golab and M Tamer 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.

[14] Horacio González-Vélez and Mario Leyton. A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers. *Software–Practice & Experience*, 40(12):1135–1160, November 2010. 2010 JCR Impact Factor 1-year: 0.573 5-year: 0.786.

[15] Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martinez, Claudio Soriente, and Patrick Valduriez. Streamcloud: An elastic and scalable data streaming system. *IEEE Trans. Parallel Distrib. Syst.*, 23(12):2351–2365, December 2012.

[16] Arnaud Legrand Henri Casanova and Yves Robert. *Parallel Algorithms*. Chapman & Hall/CRC, Numerical Analysis & Scientific Computing Series, New York, USA, 2008.

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

[18] Gabriele Mencagli, Marco Vanneschi, and Emanuele 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 8–17. IEEE Computer Society, 2013.

[19] 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.

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

[21] M.A. Shah, J.M. Hellerstein, S. Chandrasekaran, and M.J. Franklin. Flux: an adaptive partitioning operator for continuous query systems. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 25–36, 2003.

[22] Jens Teubner and Rene 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.

[23] Yi-Cheng Tu, Song Liu, Sunil Prabhakar, and Bin Yao. Load shedding in stream databases: a control-based approach. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, pages 787–798. VLDB Endowment, 2006.

[24] Marco Vanneschi. The programming model of assist, an environment for parallel and distributed portable applications. *Parallel Comput.*, 28(12):1709–1732, December 2002.

[25] Rares Vernica, Michael J. Carey, and Chen 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.

[26] Sai Wu, Vibhore Kumar, Kun-Lung Wu, and Beng Chin 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.

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