Università di Pisa

Scuola Superiore Sant'Anna

Master Degree in Computer Science and Networking

Master Thesis

Parallel paradigms for Data Stream Processing

Candidates

Andrea Bozzi - Andrea Cicalese

Supervisor

Prof. Marco Vanneschi

Academic Year 2011/2012

You don't use science to show that you're right,
you use science to become right.
Randall Munroe

Ringraziamenti

Vogliamo ringraziare sentitamente il nostro relatore, Professor Marco Vanneschi. Fidandosi di noi, ci ha guidato durante il percorso di ricerca che abbiamo svolto in questo periodo di tesi.
Desideriamo esprimere la nostra gratitudine verso Gabriele Mencagli, Daniele Buono e Tiziano De Matteis, dottorandi e dottori di ricerca del Dipartimento di Informatica, che ci hanno aiutato con i loro preziosi consigli durante la fase di sperimentazione.
Sarebbe stato per noi impossibile raggiungere questo importante obiettivo senza il sostegno delle nostre famiglie, dei nostri colleghi laureandi Daniele e Francesco e dei nostri colleghi del Corso di Laurea in Informatica e Networking.

Desideriamo inoltre esprimere un ringraziamento a LIST Group, per aver supportato i nostri studi tramite una borsa di studio e di approfondimento.

Contents

Introduction **i**

1 Data Stream Processing state of the art **1**

1.1 Background 1

1.2 The Data Stream Model 3

1.2.1 Sliding windows 6

1.2.2 Approximation Techniques 7

1.3 Basic counting 8

1.4 DaSP parallel module structure 10

2 Distribution techniques **12**

2.1 Distribution techniques for one stream 13

2.1.1 Distribution with replication 13

2.1.2 Pane-based distribution 17

2.1.3 Batch-based distribution 19

2.2 Distribution techniques for multiple streams 20

2.2.1 Aggregation 21

2.2.2 Parallelization 22

3 Computation on tumbling windows **24**

3.1 Problem 25

3.2 Map 27

3.2.1 Map: streaming and total buffering 29

3.2.2 Map: streaming only 29

3.3 Map with replication 31

3.3.1 Map with replication: streaming and total buffering . . 33

3.3.2 Map with replication: streaming only 33
3.4 Stencil 37
3.4.1 Stencil: streaming and total buffering 40
3.4.2 Stencil: streaming only 40
3.5 Stencil with replication 41
3.5.1 Stencil with replication: streaming and total buffering . 44
3.5.2 Stencil with replication: streaming only 44
3.6 Comparisons and numerical examples 47
3.6.1 Memory occupation 47
3.6.2 Completion time 49

4 Parallel Stream Join **53**
4.1 Stream Join semantics 53
4.2 Sequential Algorithm 55
4.3 DaSP module formalization 56
4.4 Existing implementations 59
4.4.1 CellJoin 59
4.4.2 Handshake join: How soccer players would do stream joins 60
4.5 Proposed parallel solutions 62
4.5.1 Time-based distribution solution 65
4.5.2 Round-robin distribution solution 67
4.5.3 Possible mappings of input data onto workers 69
4.6 Implementation 72
4.6.1 Assumptions 72
4.6.2 Tuples attributes and join predicate 72
4.6.3 Tuples memory organization 73
4.6.4 Inter-thread communications: FastFlow queues 75
4.7 Tests 76
4.7.1 parJoin 78
4.7.2 Comparison with literature using symmetric input rates 83
4.7.3 Comparison with literature using asymmetric input rates 87

5 Conclusions **90**

Introduction

Data Stream Processing

In the last years, as the network infrastructures kept on growing faster, many data-intensive and network-based application (e.g. financial applications, network monitoring, security, telecommunications data management, web applications, sensor networks) gained more and more importance in the Information Technology area. For these applications the input data are not available from random access memory since the beginning of the computation, but rather arrive on one or more continuous *data streams*. This research field is called *Data Stream Processing* [BBD+02], in short DaSP.
In this *data stream model*, data items continuously arrive in a rapid, time-varying, possibly unpredictable fashion. Furthermore, the dimension of the stream is potentially unbounded, or anyway it does not permit the memorization of all the data as it is possible in a traditional memory-bounded application.
Another important consideration is that, in *data streaming applications*, more recent data are considered to be more important with respect to the older ones [Gul12]. This feature derives from the unbounded nature of the streams and confirms that data are not sent to a processing node in order to be stored; rather, data are sent in order to produce new results as soon as possible, i.e. on the fly. Considering for example Data Base Management Systems (DBMS), in the streaming context, a query is no longer something that is sporadically executed by a user (or periodically triggered). On the contrary, a specific query is performed continuously and its computation is updated any time new informations are available (*continuous query*). Obviously, the

algorithm that is computed must be able to keep pace with the arrivals from the stream.
This last consideration introduces another key factor in data stream processing: keeping the latency of the application as small as possible. For example, in online trading scenarios involving credit card transactions it is imperative to provide low latency as these kind of applications must complete their tasks with really strict time limitations.
Many studies about DaSP have been made in the Data Base Management Systems research field. As already said, in streaming applications, it is possible to maintain in memory only a portion of the information received. In order to make the computation possible with this assumption a new concept has been introduced in DBMS: the idea is to compute functions over portions of the incoming data (referred to as *windows*). Windows are used to maintain only the most recent part of a stream. It is important to distinguish two different semantics that the concept of window can assume:

- The window's size tells the application which are the more recent data that are important. It is given as an input of the application. This is the most common case.
- The window's size is chosen considering the trade-off between available memory and the precision of the results that will be obtained.

Windows can also be classified in two different types: *time-based windows* are defined over a period of time (e.g. elements received in the last 10 minutes) while *count-based windows* are defined over the number of stored inputs (e.g. last 50 received elements).
Important research studies and analysis of data streams were also done from an algorithmic perspective [DM07]. When limited to a bounded amount of memory, it is often possible and acceptable to have approximate results. There is a trade-off between the accuracy of the solution and the memory required. A series of approximation techniques have been proposed, e.g. sketches [RD09], wavelets [CGRS01], histograms [IP99].
Because of the unique features described, pre-existing applications cannot be simply adapted to the streaming context. This introduces several new

research problems.

High Performance Computing

In recent years, a substantial improvement in computer and network technology made available parallel and distributed architecture capable of reaching unprecedented performance. Over the last decade, an important technological revolution has taken place: Chip MultiProcessors (CMP), simply known as multi-cores, almost replaced uniprocessor-based CPUs for both the scientific and the commercial market. They can be considered multiprocessors integrated on a single chip, thus many theoretical results found for classical shared memory architectures are also valid for multi-cores. According to new interpretations of the Moores law, the number of cores on a single chip is expected to double every 1.5 years. It is clear that so much computational power can be used at best only resorting to parallel processing.
High Performance Computing (HPC) [Van09, Van12] deals with hardware-software architectures and applications that demand high processing bandwidth and low response times. Shared memory multiprocessors (SMP, NUMA) are a notable example of HPC systems. To be exploited efficiently, these systems require the development of parallel applications characterized by high levels of performance and scalability.
The change from single-core to multi-core architectures impacted also the programming model scenario introducing the need for efficient parallel programming paradigms and methodologies. The spread of a parallel programming methodology is acquiring a great importance in the scientific community. In structured parallel programming (or skeleton-based parallel programming) a limited set of parallel paradigms (*skeletons*) is used to express the structure of the parallel applications. Properties like simplicity and composability, as well as parametric cost models, make structured parallel programming a very attractive approach to dominate the complexity of the design and the evaluation of parallel and distributed applications. In structured parallel programming, an application can be seen as a graph of communicating modules that can have a parallel implementation exploiting one of many

existing skeletons (e.g. farm, pipeline, map).

Thesis contribution

Despite both the theory behind structured parallel programming and DaSP are quite solid, it does not exist a large literature in parallel Data Stream Processing. The DBMS community addressed the problem because classical data bases universally store and index data records before making them available for query activity. Such outbound processing cannot deliver real-time latency, as required by DaSP. To meet more stringent latency requirements, the DBMS community introduced the Stream Processing Engines (SPEs) that adopt an alternate model where query processing is performed directly on incoming messages before (or instead of) storing them.
We began our thesis work with the study of the main results obtained in the SPE context, but we did not stick just with DBMS on stream issues.
Our thesis work aimed to introduce a parallel streaming methodology that adapts to streaming problems, independently from the application context. In order to adapt the well known results in structured parallel programming, we propose a parallel module for streaming applications based on existing parallel paradigms (e.g. farm, data parallel, stencil).
One of the main issues in parallel data stream is the distribution of the elements to the parallel workers of the module. At this point it is worth noting that stream windows can also overlap: a parameter called *slide* determines how many elements or time instants to wait for the beginning of the successive window. If the slide is equal to the windows size, and thus windows do not overlap each other, we have *tumbling* windows, otherwise we fall in the *sliding* window model and different windows share some elements. Much effort has been put in exploiting distribution to transform a sliding window stream computation in a tumbling window one without changing its semantic. We noticed that, if windows do not overlap (tumbling window case) and we are able to make our parallel module not a bottleneck, than we can reuse classical parallel paradigms with small modifications. This happens because every tumbling window can be considered as an independent data structure received on stream.

In our thesis work we also addressed a significant stream problem that does not fall under the tumbling window hypothesis. The problem we dealt with is the *Stream Join*: it consists in continuously performing the classical data bases join on the tuples coming from two different unbounded streams. Stream join has sliding windows and it is a time-based streaming problem. *We choose to address stream join because there are parallel solutions in literature that we can compare with and also because it falls in a class of streaming problems (sliding windows) that we did not analyse previously. At first, we studied formally the problem and found the theoretical maximum output rate of the module implementing a stream join algorithm, then we proposed a parallel solution to the problem.*
Our solution is based on a data parallel paradigm (*stencil*) and takes advantage of the results obtained in the previous chapters of the thesis about the distribution of input elements.
We performed different kind of tests on our application parJoin:

- We studied how parJoin behaves by measuring its scalability and its output rate with respect to the theoretical one.
- We compared parJoin with *handshake join* [TM11] which is the fastest solution in literature to the best of our knowledge.

Structure of the thesis

This will be the structure of the thesis.

Data Stream Processing state of the art We here formally introduce DaSP and highlight the concepts that we are going to exploit in our parallel model.

Distribution techniques Distribution techniques for parallel modules operating on streams. In some cases we are able to transform sliding windows streams into tumbling windows streams.

Computation on tumbling windows Here we show how we can exploit well known data parallel paradigms for data stream processing on tum-

bling windows. We describe a data stream problem and we solve it in four different manner.

Parallel Stream Join In this chapter we analyse the stream join problem and we show our proposed solution along with its implementation. Test results are discussed and a comparison with an existing solution is presented.

Conclusions Final remarks and future works.

Chapter 1

Data Stream Processing state of the art

In this chapter we will analyse the background on the Data Stream Processing (DaSP). This research topic has been studied from many different points of view (data base, financial data analysis, network traffic monitoring, telecommunication monitoring), here we try to highlight the concepts that we want to reuse for our model. Later we define the general parallel module structure.

1.1 Background

In order to describe the data stream unique challenges we present a possible application: the *Data Stream Management System* (DSMS) [BBD+02]. DSMSs are data base systems designed specifically to elaborate queries, expressed through a SQL-like syntax, on data streams. Three key features of these DSMSs are:

- the ability to execute continuous queries on a data stream.
- the focus on producing updated outputs as new tuples arrive on the input stream.

- the fact that the input data are not completely available at the begin of the computation

It is particularly meaningful to use an example coming from the data base context because much effort was put on data stream research by this community in the last ten years.
Some existing framework based on data stream management system concepts are the Stanford's STREAM [ABB+04], Wisconsin's NiagaraST [UoWM, NDM+01], Oldenburg's Odysseus [oO], MIT's Borealis [AAB+05, ACC+03] and Berkeley's TelegraphTQ [CCD+03].
A traditional database management system (DBMS) is not designed for rapid and continuous loading of individual data items, and do not directly support the continuous queries that are typical of data stream applications. DBMS focus largely on producing precise answers computed by stable query plans. Usually classic queries are executed on demand and are called one-shot query. DBMSs assume to have available the entire data set on which the queries are performed; on DSMSs this is not true, queries are evaluated on incoming data continuously.
A possible application domain for a DSMS is network traffic management, which involves monitoring network packet header information across a set of routers to obtain information on traffic flow patterns. Consider the network traffic management system of a large network [BBD+02], e.g. the backbone network of an Internet Service Provider (ISP). Such systems monitor a variety of continuous data streams that may be characterized as unpredictable and arriving at a high rate, including both packet traces and network performance measurements. A data stream system that could provide effective online processing of continuous queries over data streams would allow network operators to install, modify, or remove appropriate monitoring queries to support efficient management of the ISPs network resources. In these case the element of the input stream are packets, seen by the DSMS as tuples of elements. We can imagine different kinds of operations that the DSMS can perform over its data stream: it can be interesting for the user to know how many packets are directed toward the same destination, to find the number of packets produced by the same application level protocol, or to find the peak of traffic over a certain time. In this case, all the functions used (COUNT,

SUM, MAX and MIN) are aggregate functions. This does not mean that all the possible operations on stream are aggregate. Imagine that the user wants to compare a certain field (e.g. the TCP time to live) of the packets between all elements in two different flows. The computation will operate on two different streams and will consist of consecutive *join* operations that are not aggregate.
We now try to individuate the unique challenges related to Data Stream Processing.

Unbounded memory requirements Since data streams are potentially unbounded in size, the amount of storage required to compute an exact answer to a data stream query may also grow without bound. External memory solutions are not well suited to data stream applications since they do not support continuous elaboration and are typically characterized by high latency. For some problems a certain capacity of data storage will be unavoidable.

Low latency computation New data is constantly arriving even while the old data is being processed; the amount of computation time per data element must be small, otherwise the latency of the computation will be too high and the algorithm will not be able to keep pace with the data stream.

We can here foresee a relation between the input buffering and the latency of the successive elaboration. In the next chapters we will try to study this relation especially from the point of view of a parallelization methodology. Now we formalize the basic concepts of Data Stream Processing.

1.2 The Data Stream Model

In Data Stream Processing the input data are not available on disk or on local memory, but arrive on one or more *data streams*. The streams are potentially unbounded in size and the application has no control over the order in which data elements are received. The inter-arrival time of the streams may be unknown, highly variable and/or characterized by peaks of

burst traffic.
It is worth showing some example of applications generating an unbounded sequence of data [Gol06].

- Networks of sensors with wireless communication capabilities are becoming increasingly ubiquitous. Sensor networks are used for environmental and geophysical monitoring, road traffic monitoring, location tracking and surveillance, and inventory and supply-chain analysis. The measurements produced by sensors (e.g. temperature readings) may be modelled as continuous data streams.

- The World Wide Web offers a multitude of on-line data feeds, such as news, sports, financial tickers and social network databases. These data may be processed in various ways from the classical DB queries to the creation of linked graph.

- An overwhelming amount of transaction log data is generated from telephone call records, point-of-sale purchase (e.g. credit card) transactions, and Web server logs. On-line analysis of transaction logs can identify interesting customer spending patterns or possible credit card frauds.

- As said, in the networking and teletraffic community, there has been a great deal of recent interest in using a Data Stream Management System for on-line monitoring and analysis of network traffic. Specific goals include tracking bandwidth usage statistics for the purposes of traffic engineering, routing system analysis and customer billing, as well as detecting suspicious activity such as equipment malfunctions or denial-of-service attacks. In this context, a data stream is composed of IP packet headers.

The more common data stream applications work on large amounts of input data and are characterized by requirements of low latency and high bandwidth.
What the computation receives on the stream can be:

1. A finite or infinite series of independent and, from the computation viewpoint, unrelated values.

2. A finite or infinite succession of elements related by some data dependency.

3. A single data structure unpacked to be sent onto a finite stream.

4. A single data structure changing over time, unpacked to generate an infinite stream.

5. A finite or infinite set of data structures unpacked.

The application run-time can try to compute online or it can buffer some values in temporary data structures, which size is to be minimized according to the memory constraint of the computation. However, to store and access data on memory may result in high latency operations which a data stream computation may not be able to sustain. It is important to note that the size of the memory is negligible with respect to the size of the stream. A first issue arises: a data stream computation must be able to store as little input data as possible in order to have a small latency in data access.
In the simplest case, the computation is applied on the single values of the input stream independently. Having a sequential algorithm for this case, we can exploit all the theory about stream parallel and data parallel computation to study and implement efficient solutions for this kind of problems [Van09, Van12].
The main case that we want to address in this thesis coincides with computations applied to the whole input stream or to a part of it. In these applications one output value is obtained by applying a function to a subset of values received from one or more input streams. The focus of these computations is on recent data; for this reason a mechanism that allows to elaborate latest elements is needed. These computations usually exploit the concept of *sliding window*.

1.2.1 Sliding windows

In the sliding window model [BBD+02, DM07] only the newest elements arrived from the stream are considered relevant at any moment. There are two different ways to choose the elements:

- In the **count-based** windowed model, there is a fixed value N called window size. Only the last N elements are part of the current window.
- Using the **time-based** method, the windows are composed by all the elements received in a fixed time frame.

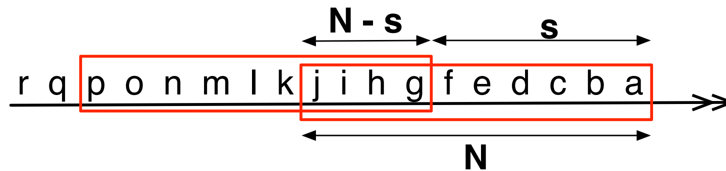


Figure 1.1: Concepts of window size and slide in sliding windows

In both cases we have a window size and a *slide*. The slide value represents how many elements to wait for the beginning of the next window. It also indicates whether or not two consecutive windows overlaps; given a window with window size N (in the count-based model) and a slide of s, then the last $N - s$ elements are in common with the successive window. If $N = s$ the windows of the stream do not overlap and we are talking of *tumbling windows* instead of sliding windows.
Every element is characterized by the notion of *unique id* (or *time-stamp* in the time-based model), which corresponds to the position of a data element inside the stream, in order to determine the position of an input in every windows it belongs.
The sliding window model emphasizes *recent* data, which in the majority of the data stream application are more important. This technique also represent an approximation method for infinite streams. In this case, the size of the windows must be decided accordingly to the results that we want to obtain from the computation, their approximation degree and the time that

we want to wait between two different results. However it is a deterministic approach, so there is no danger that unfortunate random choices will produce a bad approximation. In other cases, the window size is more algorithm-dependent: for example consider algebraic computations on matrices where the dependencies over the elements are not arbitrary.
As we have pointed out, memory occupation saving is a main issue in data stream processing. Having overlapping windows may lead to buffering: the common set of data between two or more input windows must be buffered by the application to be reused for different computations. We will see in the next chapters that, even after parallelizing the data stream computation, the overlapped parts of the stream may require some replication and/or buffering.

1.2.2 Approximation Techniques

As described in the previous section, when we are limited to a bounded amount of memory, it is not always possible to produce exact answers; however, high-quality approximate answers are often acceptable in many data stream applications. In other cases approximation is needed because the application cannot sustain some peaks of traffic in the stream. We will now review some of these techniques.

sketches Sketching [RD09] techniques summarize all the inputs as a small number of random variables. This is accomplished by projecting the domain of the input on a significantly smaller domain using random functions. In this way the data are summarized using a limited amount of memory.

random sampling and load shedding These techniques [BDM07] rely on dropping some part of the input. Usually sampling and load shedding rely on heuristics but for some kind of computation like query responses systematic approaches exist.

batches Rather than producing a continually up-to-date answer, the data elements are buffered as they arrive, and the answer to the query is computed periodically as time permits [BBD+02]. In this case the re-

sults are not approximate but arrive with a time delay due to buffering. It is also a good approach when data streams are bursty.

histograms These are data structures [IP99] that allows to summarize data sets and to quantify the error in the approximation of the computation.

wavelets Wavelets [CGRS01] are often used as a technique to provide a summary representation of the data. Wavelets coefficients are projections of the given set of data values onto an orthogonal set of basis vector. The choice of basis vectors determines the type of wavelets. The wavelet decomposition can be used to effectively generate compact representations that exploit the structure of data. Furthermore, wavelet transforms can generally be computed in linear time, thus allowing for very efficient algorithms.

Approximation introduces an important trade-off in data stream applications between the accuracy of the results produced and memory utilization. As much as the accuracy of the outputs decreases also the buffering requirements of the application drops. Obviously the same trade-off can be seen in terms of latency and accuracy: more approximation gives a smaller latency and also implies less memorization and precision. In order to choose one of these technique, a deep comprehension of the computation is necessary. For our purposes it will not be important whether or not approximation is used because the parallelization of a data stream computation will not affect the sequential algorithm used.

1.3 Basic counting

In this section we present a very simple problem on single stream that can be useful to better understand the sliding window model, the dependencies over data and the trade-off between the accuracy of the solution and the memory required.

Basic Counting [DM07]
Given a stream of data elements, consisting of 0's and 1's, main-

tain at every time instant the count of the number of 1's in the last N elements.

This problem can be seen as a simplification of a more realistic problem in a network-traffic scenario. Imagine an high speed router that maintains statistics over packets. In particular we do a prefix match to check if a packet originates from the `unipi.it` domain. At every time instant we want to compute the number of packets, among the last N, which belonged to that specific domain. In our **Basic Counting** problem a 1 represents a packet from the `unipi.it` domain.
The nature of the problem leads to the utilization of the sliding window model. In particular, a window will have a size of N because we want to count the number of 1's over N successive elements. At every time instant we want to compute the result, therefore the slide between the windows will be $s = 1$.
A sequential trivial solution for this problem requires $O(N)$ bits of space. In particular, an exact algorithm utilizes a counter of $\log N$ bits for the result and a FIFO queue containing the last N elements received. In this way every time a new element arrives its value is added to the counter while the value of the element extracted from the queue is subtracted.
In [DM07] the authors present an approximation technique called Exponential Histogram that solves the problem requiring $O(\frac{1}{\epsilon} \log^2 N)$ bits of memory and provides a result accurate within a factor of $1 \pm \epsilon$. These results are extended also for the following problems:

- sum, variance, L_p norm sketches
- k-median, similarity, rarity
- approximated count, min/max, quantile

We can see that there is a trade-off between the accuracy of the solution and the memory required for the buffering because of the overlapping windows.

1.4 DaSP parallel module structure

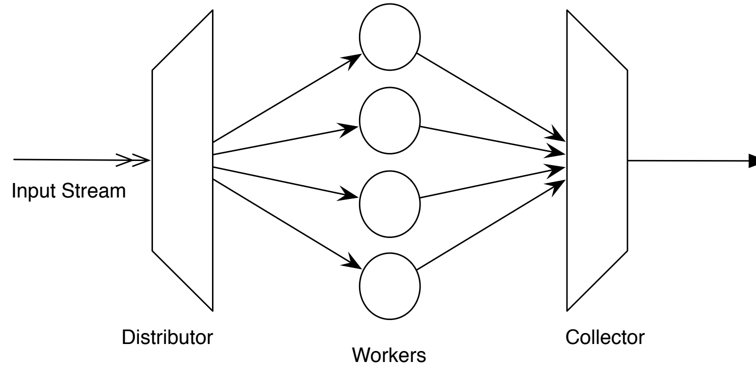


Figure 1.2: Generic Data Stream Processing parallel module

In order to parallelize a Data Stream Processing (DaSP) computation, we will utilize the well-known model with computation graphs made of parallel modules communicating through typed channels [Van09, Van12].
The parallel computation will be composed of a *distributor*, a certain number of *workers* (internally implemented in a sequential or a parallel manner) and a *collector*.
Figure 1.2 shows a DaSP module composed of:

- a *distributor* D which sends the elements of the stream to the workers. The distribution methods and the possible replication of the elements will be described in chapter 2.

- W *workers* which implement the computation of the desired function over the input elements. The workers can be internally implemented in a parallel manner. The execution phase and the organization of the workers will be analysed in chapter 3.

- a *collector* C which receives all the results computed by the workers.

The number of workers W, which represents the parallelism degree of the DaSP module, has to be chosen accordingly to the cost model in order to

obtain the desired completion time or the needed memory occupation.

If a worker receives only input elements belonging to non overlapping windows (cf. tumbling windows in section 1.2.1), we can observe some positive effects. In this case the worker schedule is:

1. compute over the N elements of a window
2. obtain a result
3. restart an *independent computation* over the successive N elements

There is no need for additional memory because there are no shared elements among the windows received. Therefore, from the worker viewpoint, it would be desirable to receive input elements that belong to non overlapping windows. Delivering the elements to the workers, we can obtain the case of independent computations under some conditions that will be described in chapter 2.

The execution phase in the workers is another important aspect of a DaSP module structure that will be analysed in chapter 3. In all the cases, workers can utilize any approximation technique presented in section 1.2.2, after parallelization is enabled.

Chapter 2

Distribution techniques

In this chapter we analyse distribution techniques for parallel computations working on streams of data. Data Stream Processing can be parallelized partitioning the input data over different workers. To do so we utilize the well-known model with computation graphs made of parallel modules communicating through typed channels [Van09, Van12]. The final graph will have in general a distributor (also called *emitter*), a certain number of workers (internally implemented in a sequential or a parallel manner) and a collector.
In DaSP applications, the most interesting case occurs when a result depends on many (all the) elements of the stream. In general, the output of a computation is obtained from a subset of successive elements of the input stream.
For most data streaming computations, recent data are more useful than older data; the processing is done on a subset of the last elements received from the input stream. These applications utilize the sliding window model already described in section 1.2.1.
In this overlapping windows scenario some elements belong to more than one window; computing on different windows involve the memorization of shared elements or the replication of the computation.
Elements belonging to different windows can be elaborated in parallel because there are no data dependencies among different windows. In order to parallelize the computation, each worker has to process different windows (which represents independent subsets of data). Therefore the distribution of the input elements is a crucial phase of the parallelization because it can

impact on the efficiency and the resources utilization.

2.1 Distribution techniques for one stream

In this section we present different forms of distribution of the elements to the workers. The first scheme is a window-based distribution exploiting some elements replication. Then we introduce the concepts of *panes* (window subsets) and *batches* (windows aggregates).

2.1.1 Distribution with replication

If the slide is equal to the size of the window ($s = N$) and we are in presence of *tumbling windows* the input stream can be simply partitioned among workers because each element belongs to exactly one window.

In general, when $s < N$, the distribution strategy employs replication of some elements because we fall in the *sliding windows* scenario where some elements belong to more than one window and therefore must be processed by more than one worker. In particular each element belongs to $\lceil \frac{N}{s} \rceil$ different windows and this is the number of time it should be replicated among different workers.
From this reasoning we can deduce two cases to be considered when in presence of window-based distribution with replication:

number of workers $\geq \lceil \frac{N}{s} \rceil$ Here we can distribute the input elements in such a way that the workers receive independent and non overlapping windows. This is the desired effect described in section 1.4: from the point of view of the worker it is like working with the *tumbling windows* model. This behaviour is feasible because we have enough workers to handle overlapping windows separately. Every element is replicated as many time as necessary among a certain number of workers (i.e. *multicast*).

number of workers $< \lceil \frac{N}{s} \rceil$ In this case the number of worker is not sufficient to replicate the elements as many time as necessary, and therefore a worker will receive windows that still overlap each other. From the point of view of the worker we are still in a *sliding windows* model and some buffering of the inputs elements is needed, or an approximation technique must be used if the memory occupation becomes a problem. Every element is replicated among all the workers (i.e. *broadcast*).

In both the cases described the emitter has to replicate a certain number of elements among the workers. A round-robin scheduling strategy can be used, however this strategy is not the only one possible. For example an on-demand strategy is also feasible, but it has to be done with respect to the windows instead than the single input elements. In general, the emitter has to be aware of the window size N and the size of the slide s in order to distribute the elements in the desired manner. It has also to determine the begin and the end of every window that has to be sent to a specific worker. To solve this problem each element of the stream has a unique *id* (count-based model, cf. section 1.2.1) or a *time-stamp* (time-based model, cf. section 1.2.1) that can be used along with the information of N and s to recognize the boundaries of every window.

Figure 2.1 shows an example where the number of workers ($W = 5$) is greater than $\lceil \frac{N}{s} \rceil$. In this particular case the window size is $N = 4$ and the slide is $s = 1$. The computation done by the workers could be the **Basic Counting** shown in section 1.3 or any other function that works on sliding windows with slide one.
In steady state, each element will be replicated $\lceil \frac{N}{s} \rceil = 4$ times and will be sent to four different workers. Each worker computes the function over N elements, sends the result to the collector and restarts the computation on the successive N elements. If the problem to be solved is Basic counting, the worker needs only the counter to store the sum of the N elements. There is no need for the additional FIFO queue and therefore the memory occupation for each worker is $O(\log N)$. Of course we should remind that the number of worker in this case is $W > \lceil \frac{N}{s} \rceil = N$.
The emitter does not need additional memory to store some elements, it

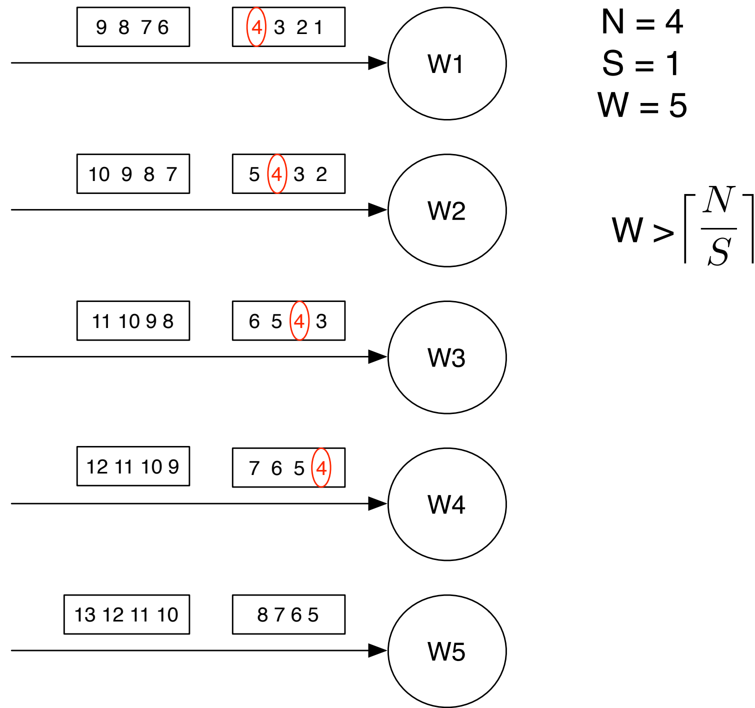


Figure 2.1: Distribution with replication where $W > \lceil \frac{N}{s} \rceil$

just sends each element in multicast to the proper workers in a round-robin fashion. The impact of the multicast on the total computation should be considered to check if the emitter becomes a bottleneck of the computation graph. When it becomes a bottleneck and its distribution time ($L_{scatter}$) is bigger the the inter-arrival time (T_A) some buffering may be necessary also in the emitter.

The second example where the number of workers ($W = 2$) is smaller than $\lceil \frac{N}{s} \rceil$ is shown in figure 2.2. In this case the window size is $N = 6$ and the slide is $s = 2$. The computation performed by the workers could be any function that can work on sliding windows with slide two; it is possible to imagine also a generalization of the Basic Counting for this case.
In steady state, the emitter just replicates the input stream to all the workers with a broadcast. Every element will be replicated $W = 2$ times (which is

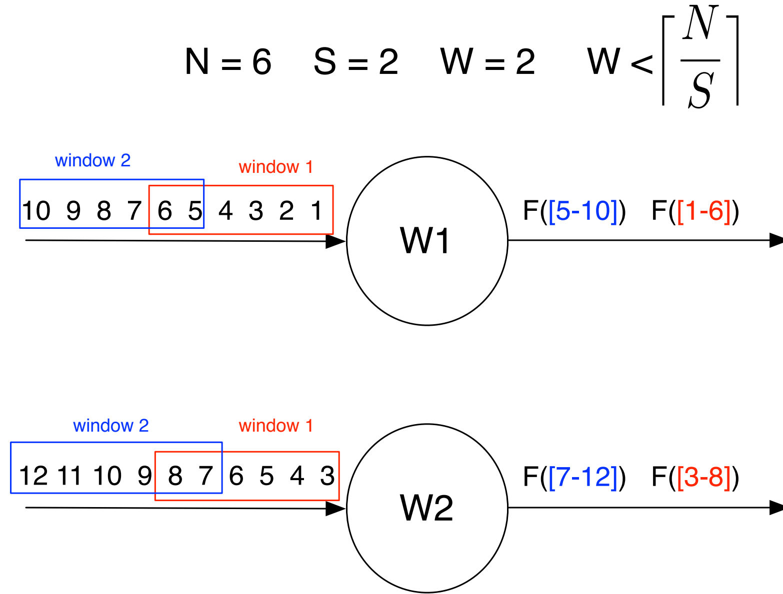


Figure 2.2: Distribution with replication where $W < \lceil \frac{N}{s} \rceil$

smaller than $\lceil \frac{N}{s} \rceil$) and sent to all the workers as shown in figure 2.2.
From the point of view of the worker this case is similar to the sequential one because the elements received belong to overlapping windows. However the number of elements shared among windows are less than the sequential case and therefore the memory required for the buffering is smaller in every worker. If there is not enough memory for the temporary storage, the approximation techniques can be used in each worker without changing the emitter logic.
The impact of the broadcast on the total computation should be considered to check if the emitter becomes a bottleneck of the computation graph. In this case we don't have the maximum number of replication ($\lceil \frac{N}{s} \rceil$) of the elements and the impact of the broadcast should be smaller than the one of the multicast in the first case described.
The negative aspect of this case, with respect to the first one, is that in each worker we have to pay additional memory occupation. This is due to the buffering of the elements shared among windows received by the same worker.

2.1.2 Pane-based distribution

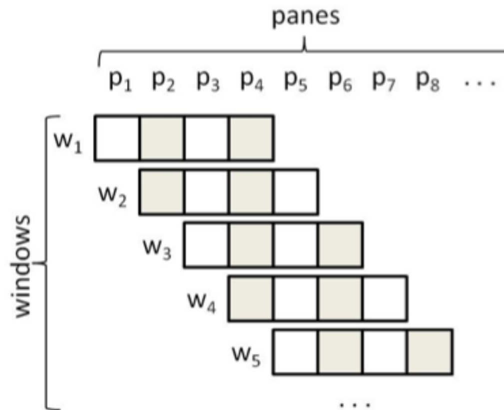


Figure 2.3: Division of windows in panes

The idea behind this form of distribution is to divide the windows in smaller subsets of elements called *panes* which are then assigned to the workers [BT11, LMT^{+}05]. This method gives a finer grain with respect to the precedent one which is window-based.
The panes are disjoint subsets of elements and do not overlap each other. Given a problem with window size N and slide s, the stream is separated into panes of size $GCD(N, s)$. Each window of size N is therefore composed of $\frac{N}{GCD(N,s)}$ consecutive panes as shown in figure 2.3 (adopted from [BT11]). Every pane is distributed to a different worker in a round-robin fashion avoiding replication for the emitter module. The computation is performed on panes and not on windows any more; we are decreasing the computational grain. The workers communicate each other on a ring topology sending the panes necessary to compute the complete result on the whole window. From the point of view of the worker it is like working with the *tumbling windows* model because the panes received are not overlapping each other.
If the function to be computed is aggregate the messages sent between the

workers are not the original panes but the partial results obtained elaborating the panes. In this case there is also no replication of the computation, while in the other case there is replication among the workers.

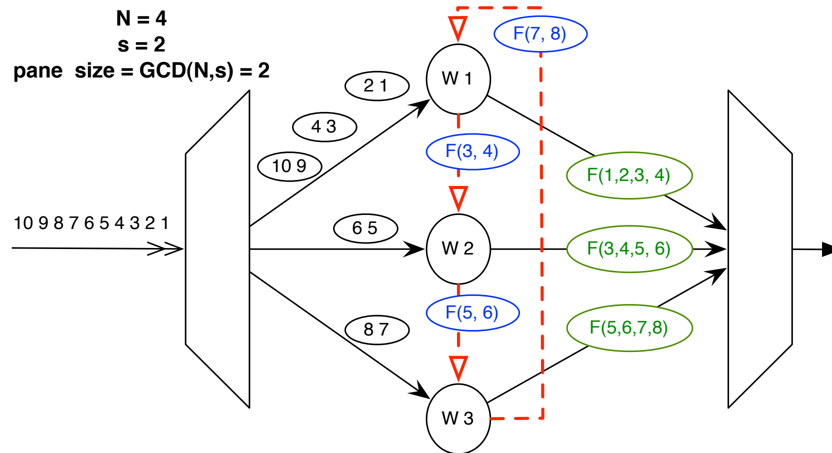


Figure 2.4: Pane-based distribution

Figure 2.4 shows an example of distribution with panes in presence of an aggregate function. Each worker computes a partial result on the pane received from the emitter (in black) and sends it to next worker on the ring. Then it receives from the previous worker on the ring a result (in blue) that will be utilized to complete the computation on the assigned window.

The pane-based distribution is very useful in presence of aggregate functions because it avoids replication of input elements done by the emitter and avoids replication of computations in the workers. Of course there is a inter-worker communication to be considered. This distribution scheme can be seen as a data-parallel computation with *stencil*. Indeed we have an emitter, a collector and some workers communicating with a ring topology.

2.1.3 Batch-based distribution

In this section we present a distribution method of the windows which has an approach opposite to the one with panes.

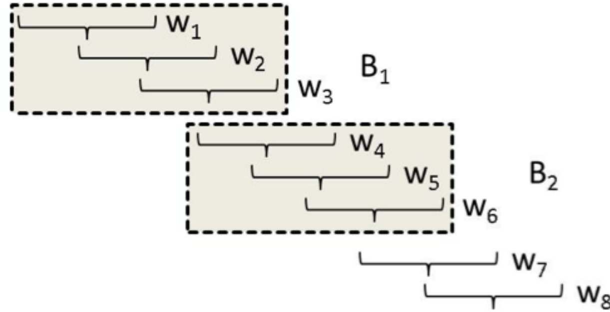


Figure 2.5: Aggregation of windows in batches

A number of consecutive windows is grouped into a *batch* as shown in figure 2.5 (adopted from [BT11]). The key idea behind this coarser-grained partitioning scheme is to reduce the element replications over workers by reducing the overlapped part across the new data partitions (batches).
Given a problem with window size N, slide s and a batch size of B (number of windows in the same batch), then the number of elements that can fit in a batch-window is $w_b = N + (B - 1) * s$ and slide $s_b = B * s$. Unlike the window-based partitioning case, if multiple windows are in the same batch, then common data elements do not need to be replicated to multiple workers. This is the advantage of this distribution scheme: to further reduce the replication done by the emitter and decrease the latency of the distribution phase.
From the point of view of the worker this case is very similar to the sequential one because the received elements belong to overlapping windows with the same slide size and window size. Therefore we can still apply all the considerations on the trade-off between the accuracy of the solution and the memory required for the buffering.
While in the window-based distribution scheme each element belongs to $\lceil \frac{N}{s} \rceil$

windows, in the distribution with batches each element belongs to $\lceil \frac{w_b}{s_b} \rceil = \lceil \frac{N+(B-1)*s}{B*s} \rceil < \lceil \frac{N}{s} \rceil$. Obviously the benefit of this approach would increase with a larger batch size but this also implies a larger execution time for the workers. The decision of the correct batch size is crucial and impacts on the total computation cost because tends to reduce the latency of the distribution phase and increase the computation time of the workers with respect to the window-based case.

An interesting *hybrid solution* could exploit both the techniques of panes (cf. section 2.1.2) and batches. In particular the windows are grouped in batches and the shared parts of the batches are divided in panes and distributed with the ring among the workers.
Choosing the correct size of batches and panes, the replication of the elements done by the emitter will be avoided totally at the expense of a greater execution time in the workers and an additional communication phase between the workers.
If the function is aggregate the data exchanged on the ring will not be the original elements of the stream but the partial results of the computation of each worker with a benefit for both the execution time in the workers and reduced communications between the workers.

2.2 Distribution techniques for multiple streams

In this section we discuss how having multiple input streams impact on our distribution techniques. Our aim is to reuse as much as possible the methods we have already introduced for the single stream scenario.
We proceed under the following assumptions:

- m input streams with the same windows size N.
- workers compute $F(e_1, e_2, ...e_m)$ where $e_i = \{k \mid k \in stream_i\}$ and $|e_i| = |e_j| \; \forall i \neq j$.

We describe two different solutions: the first one aggregates elements coming from different streams to create a single stream with a new type; the second

one modifies the distributor in order to have m parallel and independent sub-distributors.

2.2.1 Aggregation

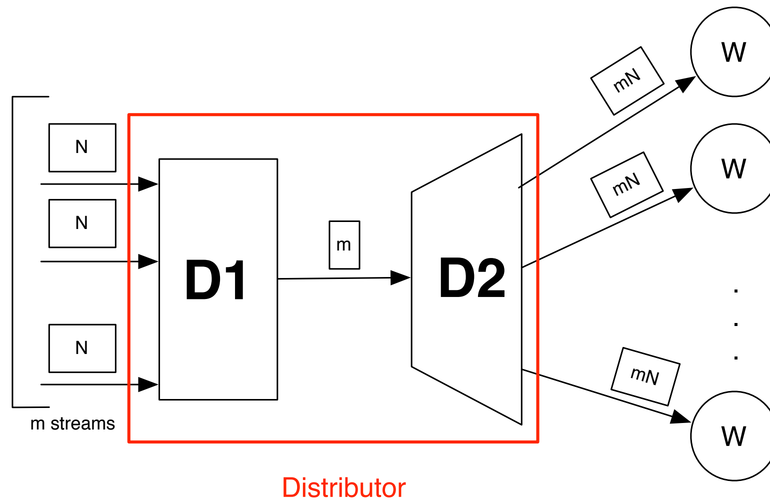


Figure 2.6: First solution for m input streams

We logically divide the distributor in two sub-modules called *D1* and *D2* as shown in figure 2.6. D1 takes care of receiving data from all the streams and generates new messages containing one element from each stream. This new messages of size m are sent to the second distributor. In this way D2 works on a single stream and can exploit all the techniques showed in previous sections to distribute the input elements to the workers. The only difference is that the input elements of D2 are tuples containing m elements.
In figure 2.6 D2 distributes windows of size mN (N tuples of size m) among its workers. In general D2 can utilize all the described techniques such as replication, pane-based distribution and batching.

We note that, if the streams do not have the same inter-arrival time, both in presence of count based or time based window, this solution introduces a

buffering time in D1 that will impact on the service time of the computation. The buffering delay is due to the waiting of m different elements necessary to create the new tuple to be sent to D2. Moreover is it difficult to evaluate a bound for the memory used by D1; we must take in consideration the inter-arrival times of m different streams and consider the bursty traffic situations. Note that this technique is not suitable when windows have different number of elements.

From the point of view of the workers the input messages and the input channels are modified. Instead of m input streams, a worker has only one input stream carrying messages containing one element from each original stream. The worker logic, in this case, may need to be changed in order to support the new communication channels and the new type of messages.

2.2.2 Parallelization

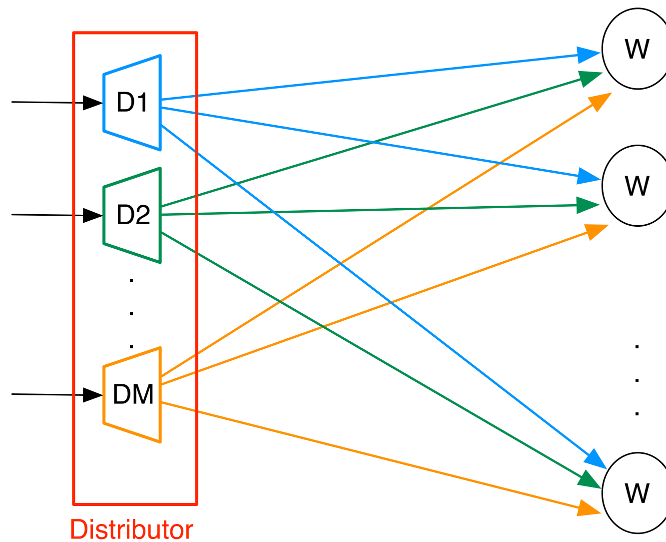


Figure 2.7: Second solution for m input streams

In this case the distributor is logically composed by m independent modules D_i, one for every input stream (figure 2.7). Here we can apply the

techniques proposed in the previous sections among every D_i and the workers. The main drawback of this solution is the high number of communication channels between the distributor and the workers.
Depending on the function to be computed by the workers, the distribution strategy adopted by each D_i may be different.
For example, in the case of two input streams, one distributor could utilize replication of the input data while the other could partition the windows among each worker.

From the point of view of the workers the input messages and the input channels are not modified. A worker still has m input streams as in the sequential case.

Summary

In this chapter we studied possible distribution techniques for parallel DaSP modules.
At first we analysed modules with only one input stream, we investigated how the number of workers of the module and the size of the windows impact on distribution of sliding windows. Then we presented two optimization techniques existing in literature that can be exploited also by our distribution schemes: pane and batch-based distributions. These two techniques allow us to change the grain of the windows and can be also mixed together. Finally, we showed how to reuse obtained results in the case of multiple input streams.

Chapter 3

Computation on tumbling windows

In this chapter we show how we can exploit and characterize well known parallelism schemes [Van09, Van12] for Data Stream Processing on tumbling windows. We already described in chapter 2 how, from the point of view of a single worker, some computations on sliding windows can be transformed in computations on tumbling windows by using an appropriate form of distribution. Here we want to study how a parallelism scheme adapts to a streaming computation and how this context impacts on its performance parameters.
To do this we will utilize an example; the idea is to study how our parallel computation behaves for the execution phase instead of the distribution one. We decided to study the behaviour of a stream computation considering a single window because the completion time obtained can be seen as the service time of the computation operating on a stream of elements in the tumbling windows model. This assumption has no impact on the semantics of our computation because we assumed to work on tumbling windows. Our example will highlight how we can parallelize the computation of a single window belonging to a finite or infinite stream.
It is worth noting that the solutions shown in this chapter may be seen as the structure of a macro-worker inside a DaSP module (cf. section 1.4) or as an entire DaSP module composed of a single macro-worker.

3.1 Problem

In order to analyse different implementations of a macro-worker we evaluate the following vector computation:

```
int A[N], B[N], R[N][N];
  ∀ i = 0 ... N−1:
    ∀ j = 0 ... N−1:
      R[i][j] = F(A[i], B[j]);
```

This simple program is composed of two nested cycles and computes N^2 elements given $2N$ inputs; every element in A must be paired with every element in B.

We will proceed in this way: we propose four different forms of parallelism to solve the problem. For each form of parallelism, we first present the *data parallel* solution that does not work on stream, then we analyse how the same solution behaves working on stream. To do so we compare two different approaches. The first consists of reusing the data parallel solution after buffering all the inputs from the stream, while the second tries to compute as soon as possible with the inputs it receives on the fly. Intuitively the first case will give us a result in the form:

$$T_{Buffer} + T_C$$

where T_{Buffer} is the time necessary to buffer all A and B from the stream and T_C is the completion time of the standard data parallel solution.

In the second case we will obtain just a T_{C_2} and we need to study how it relates to the other case. Our study will proceed under some important assumptions:

- the streams share the same inter-arrival time
- the workers are capable to overlap the execution of the function F with one send and one receive.
- A and B will be received on two different streams.

In figure 3.1 are displayed all the cases we are going to address in the next sections. The data parallel paradigms are two: *map* and *stencil*. For both we first partition array A and then replicate it on all the nodes obtaining four different parallel solutions. At this point we show the difference utilizing buffering or working purely on-stream.

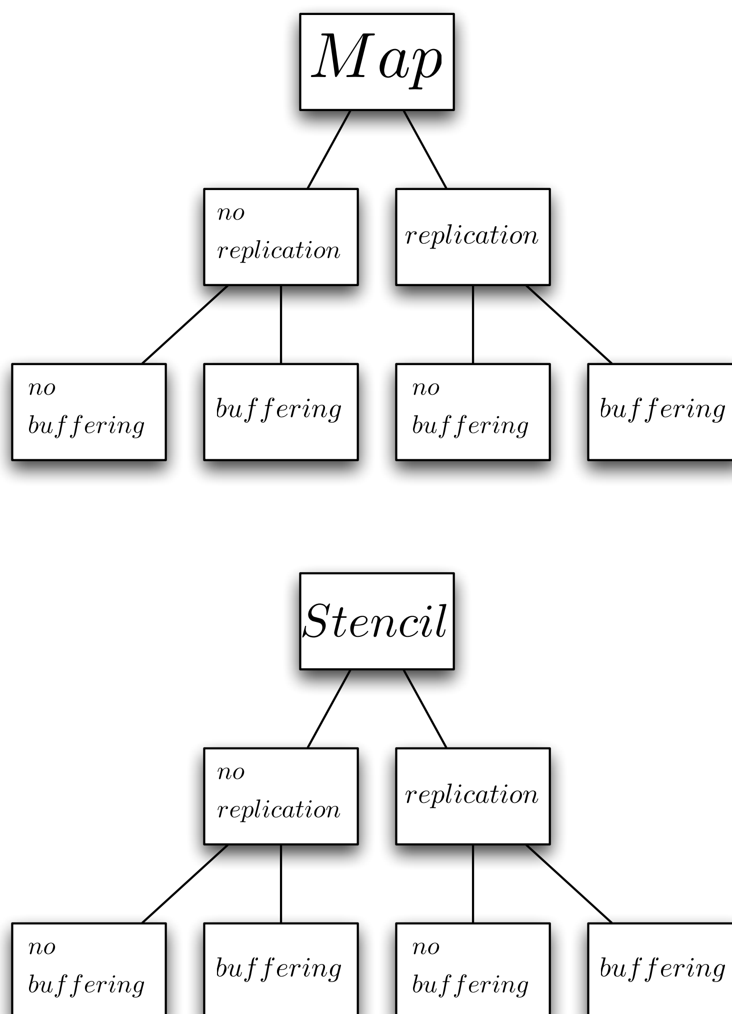


Figure 3.1: Different cases we are going to address

3.2 Map

In this section we present a solution to our problem based on the *map* data parallel structure. We apply the Virtual Processor (VP) model [Van09, Van12] to obtain a *map* computation with N^2 VPs.

Every virtual processor receives from two input channels just one element

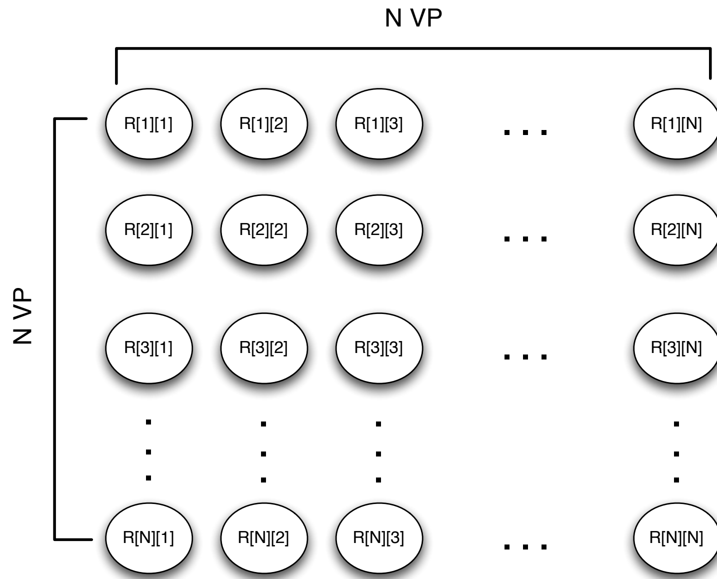


Figure 3.2: VP organization for the *map* with N^2 VPs

from A and one element from B. When the VP has received both A[i] and B[j], it computes one R[i][j] = F(A[i], B[j]). After the computation the VP will send R[i][j] on an output channel to the collector.
In order to have an efficient distribution, we assume to have two emitters that work in parallel: these modules send the same element of A or B to N VPs sequentially implementing replication of the inputs. With this solution we obtain $O(N^2)$ parallel computations and $O(N^2)$ communication channels:

- The emitter that distributes A has N^2 output channels, one for every VP.
- The emitter that distributes B has N^2 output channels, one for every VP.

- The collector has N^2 input channels, one for every VP.

Now, supposing to have n workers, we fix the computational grain as $g = \frac{N}{\sqrt{n}}$ because N elements must be partitioned among $\sqrt{n}$ workers. Then, every partition is replicated $\sqrt{n}$ times, as shown in figure 3.2.
In order to calculate the completion time of the computation, we search for the latencies of the scatter and the gather. In the next formulas $T_{send}(g) = T_{setup} + gT_{transm}$ is the latency necessary to send an element of size g on a channel.

$$L_{scatter} = nT_{send}(g)$$

One emitter sends an element of size g to $\sqrt{n}$ workers. To distribute all elements of A (or B) it must execute its routine for $\sqrt{n}$ times, thus we obtain a total of n send of size g.

$$L_{gather} = T_{send}(g^2)$$

We can observe a pipeline effect in our computation: part of the communications are overlapped with the computation of the workers. For the gather, we can consider just the latency of the last send to arrive to the collector because all the others are overlapped. The number of elements that a worker produces is g^2.
For the completion time, we just need to consider the execution time of a worker once. This happens because of the pipeline effect that we already mentioned: all the other service times are overlapped with the latency of the scatter and the execution time of the workers that we are considering.

$$\boxed{T_C = L_{scatter} + g^2 T_F + L_{gather} = O(nT_{send}(ng + g^2))}$$

Now we examine the memory requirements of the *map* computation. In every worker we just need two buffers for the inputs and one for the output. Memory per node is $O(g^2) = O(N^2/n)$. To obtain the total additional memory that the computation needs we just multiply the number of nodes times their buffers size and add the total size of the inputs. The total memory is $O(N + ng^2) = O(N^2)$.

3.2.1 Map: streaming and total buffering

At this point we introduce the streaming context in our problem: A and B are unpacked and received on two streams. We suppose that the size of the elements that the emitters receive on their stream is g, the grain of the computation.
Initially we analyse the solution that utilize the *map* described above (without changes) after buffering all the inputs received.
We can write the completion time of this solution as

$$\boxed{T_C = \sqrt{n}T_A + L_{scatter} + g^2 T_F + L_{gather}}$$

where T_A is the inter-arrival time from the stream.
The T_{Buffer} in this case is equal to $\sqrt{n}T_A$, because the input will be decomposed as $\sqrt{n}$ different messages ($g = \frac{N}{\sqrt{n}}$).
The memory in the workers is the same as in the data parallel case while the emitters have to buffer all the input array (A or B).

3.2.2 Map: streaming only

In this subsection we take advantage of the streaming context letting the emitters begin to distribute the inputs as soon as they arrive from the streams. We still have two emitters working in parallel and messages of size g. Every emitter distributes the input coming from the stream in a round-robin fashion. Every worker still behaves like in the data parallel case: after receiving its two inputs it computes the results and sends it to the collector. In this case the ratio between the inter-arrival time of the streams and the time to distribute the elements to the worker impacts on the computation behaviour.

If $T_A < \sqrt{n}T_{send}(g)$ a buffer will be needed in the emitters; this happens because the distributors become the bottleneck of the computation. The buffer size must be decided carefully using queueing theory concepts (expected number of users in the waiting line) and, in the worst case, can be up

to all the input size (as the solution with total buffering). For the completion time we have to consider, after the first inter-arrival time, the latency needed to distribute all the inputs to all the workers.

If $T_A \geq \sqrt{n}T_{send}(g)$ then the emitter can keep up with the arrivals so, for the completion time, we have to consider the total input arrival time to the emitters. Then we can consider the latency needed to distribute the last element of the stream and the latency that we must wait to compute the last result.

$$T_C = \begin{cases} T_A + nT_{send}(g) + g^2T_F + T_{send}(g^2) & \text{, if } T_A < \sqrt{n}T_{send}(g) \\ \\ \sqrt{n}T_A + \sqrt{n}T_{send}(g) + g^2T_F + T_{send}(g^2) & \text{, if } T_A \geq \sqrt{n}T_{send}(g) \end{cases}$$

Now let us compare the streaming with buffering solution and the pure streaming one; this comparison makes sense assuming the same parallelism degree n.
When $T_A < \sqrt{n}T_{send}(g)$ the completion times of the two solutions differ for the following addends:

$$T_A + nT_{send}(g) \text{ (pure streaming) and } \sqrt{n}T_A + nT_{send}(g) \text{ (buffering).}$$

Therefore we can see that in the streaming case we save $\sqrt{n}T_A$ in the completion time with respect to the solution with total buffering. It is worth noting that $\sqrt{n}T_A < nT_{send}(g)$ is the common part that we pay in both solutions; we are saving on the smallest factor of the formula.
On the other hand when $T_A \geq \sqrt{n}T_{send}(g)$ the completion times of the two solutions differ for the following addends:

$$\sqrt{n}T_A + \sqrt{n}T_{send}(g) \text{ (pure streaming) and } \sqrt{n}T_A + nT_{send}(g) \text{ (buffering).}$$

Therefore we can see that in the streaming case we gain $(n - \sqrt{n})T_{send}(g)$. Also in this case we are saving on the smallest factor because
$(n - \sqrt{n})T_{send}(g) < \sqrt{n}T_A$.

Instead of making a comparison using the same parallelism degree n it is also possible to compute the optimal parallelism degree for the pure streaming solution and obtain the minimum completion time. Although this case may have a different optimal n, if the dominant factor in the completion time formula is $g^2 T_F + T_{send}(g^2)$, the order of magnitude will remain the same. By modifying n the grain g is modified and, therefore, the total arrival time of the input data ($\sqrt{n} T_A$) changes.

While for the workers we need the same buffer size as in the other *map* solutions, the memory requirements can change strongly in the emitter.

3.3 Map with replication

In this section we still focus on a *map* computation, but this time we replicate the array A on all the nodes. After a VP receives the entire array A it stores it in its local memory. The VP will then perform the computation of F among all the elements of A and the received partition of B. Now the VPs are N and the computational grain is defined as $g = \frac{N}{n}$. In this case we have one emitter: it distributes B according to the grain g and broadcasts A to all the workers in one message.
The workers receive from one input channel the values from the emitter; only when a worker has received its partition of B and all the elements of A it begins its computation. After the computation, the worker sends all its gN results in one message on its output channel to the collector. Using this form of parallelism we obtain $O(N)$ elements computing in parallel and $O(N)$ communication channels: the distributor has N output channels, one for every worker, and the collector has N input channels as shown in figure 3.3.
In order to calculate the completion time of the computation, we search for the latencies of the communications for the partitioning of B, the replication of A and the collection of the results.

$$L_{scatter+multicast} = nT_{send}(g + N)$$

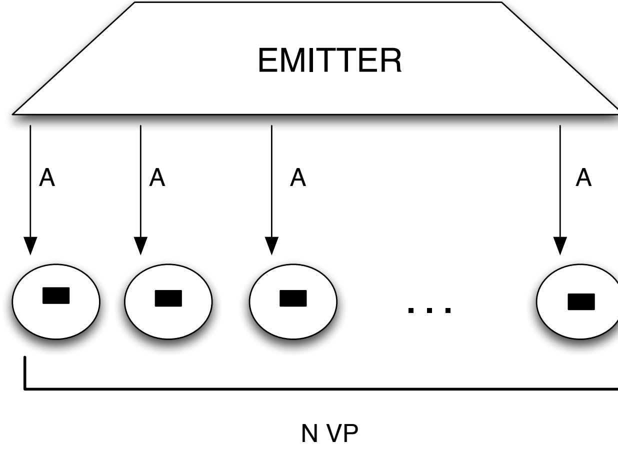


Figure 3.3: VP organization for the *map* with N VPs and replication of A

The emitter sends a partition of B of size g and the whole array A in one message to all the n workers. Therefore we obtain a total of n send of size g+N.

$$L_{gather} = T_{send}(gN)$$

Also in this case we can observe a pipeline effect in our computation: part of the communications are overlapped with the computation of the workers. For the gather, we can consider just the latency of the last send to arrive to the collector because all the others are overlapped. The number of elements that a worker produces is gN.
For the completion time, we just need to consider the execution time of the last worker. This happens because of the pipeline effect that we already mentioned: all the other service times are overlapped with the latency of the scatter and the execution time of the workers that we are considering.

$$\boxed{T_C = L_{scatter+multicast} + gNT_F + L_{gather}}$$

Now we examine the memory requirements of the *map* with replication computation. Every VP needs the space necessary to buffer the entire array A plus a partition of the array B and the result of its computation. This last factor is the largest and thus we obtain a memory per node which is

$O(gN) = O(\frac{N^2}{n})$. For the total memory consumption we have to consider n times the memory required by a node plus the space necessary for the inputs. Therefore, the total memory required is $O(ngN) = O(N^2)$
We can note that also if the number of the worker is N the memory required by the workers will be at least $2N$ because A is entirely replicated and there are N results.

3.3.1 Map with replication: streaming and total buffering

From this point our module receives the two input data structures (A, B) on two different streams. We suppose that A and B have the same inter-arrival time and that the size of the elements that the emitter receive on the streams is g. First we analyse the solution that exploits the *map* described above after buffering all the inputs.

$$\boxed{T_C = nT_A + L_{scatter+multicast} + gNT_F + L_{gather}}$$

The T_{Buffer} in this case is equal to nT_A, where T_A is the inter-arrival time from the stream. The input will be decomposed in n different messages because $g = \frac{N}{n}$.
The memory required in the workers is the same as in the data parallel case while the emitter has to buffer all the arrays A and B.

3.3.2 Map with replication: streaming only

In this case we take advantage of the streaming context by beginning the computation of a single result as soon as its inputs are available. We still have one emitter that receives the input elements of A and B with size g from two streams with the same inter-arrival time T_A. When it receives a partition of B (of size g), it forwards the message only to one worker. Then, when a partition of A (of size g) is received, the emitter broadcasts it to all the nodes to obtain the replication of A.

Now the workers behave differently from the data parallel case: each worker receives non deterministically from its two input streams and when it has received at least one partition of A and its partition of B it begins to compute F. When a result (g^2 elements obtained by applying F to all possible combinations of g elements of B and g elements of A) has been computed, the worker immediately sends it to the collector on its output channel.

Also in this case the ratio between the inter-arrival time of the streams and the time spent by the emitter to distribute the elements impacts on the computation's behaviour. This is due to the fact that the emitter becomes a bottleneck of the computation.

If $T_A < nT_{send}(g)$ a buffer will be needed in the emitter; its size must be decided using queueing theory notions (expected number of users in the waiting line) and, in the worst case, can be up to all the input size (as the solution with total buffering).
Now we discuss the completion time. We have to consider, after the first inter-arrival time, the latency needed to distribute the inputs necessary to have all the workers computing. We suppose the same inter-arrival time for the two streams and that the emitter sends the partition of B before broadcasting the partition of A. After the distribution of the first $n-1$ partitions of A and B, all but the last worker are computing. We consider the time needed to sent in broadcast $n-1$ partitions of A to n workers plus the latency to partition B (n messages) among n workers in order to enable the computation also of the last worker. According to our reasoning this first part of the completion time becomes: $L_{scatter} = (n-1)nT_{send}(g) + nT_{send}(g) = n^2T_{send}(g)$.

We now consider that the last worker to receive its partition of B will take advantage of the time necessary to replicate A in order to begin its computation. The last worker to receive its g elements of B will be also the last to receive the final g elements of A, because of the sequential implementation of the scatter. Therefore this worker can compute $g(N-g)$ results with the inputs it has already received.
We will consider the maximum between three different times that overlap.

- The first is the time to broadcast the last g elements of A $(nT_{send}(g))$ that is also the time that the last worker has to wait to receive the last partition of A.
- The second time we consider is the time that takes for the n-th worker to compute F over all the elements of A and its partition of B, obtaining gN results.
- The last time we consider is the time necessary to the worker to send all its result to the collector $(nT_{send}(g^2))$.

As said before, we considered the workers capable of computing and executing one send in parallel. In any case, we must consider the latency necessary to compute the last g^2 results and the corresponding send of size g^2 which cannot be overlapped.
$L_{execution+gather} = \max(nT_{send}(g), g(N - g)T_F, (n - 1)T_{send}(g^2)) + g^2T_F + T_{send}(g^2)$

In the case where $T_A \geq nT_{send}(g)$ the $L_{scatter}$ is completely overlapped by the time necessary to receive the inputs (nT_A). For the execution time we obtain the same result as in the other case using the same reasoning.

$$T_C = \begin{cases} T_A + n^2T_{send}(g) + L_{execution+gather}, & \text{if } T_A < nT_{send}(g) \\ nT_A + L_{execution+gather}, & \text{if } T_A \geq nT_{send}(g) \end{cases}$$

If n is the same, we can try to compare the formulas of the completion time with total buffering and with pure streaming.
We study the mathematical difference between the completion time of the solution with total buffering and the solution with pure streaming: this difference represents the time gained (if positive) or lost (if negative).

When $T_A < nT_{send}(g)$, whichever is the maximum in the formula with pure streaming, the difference has both a positive factor (different in the three cases) and a negative factor (always n^2T_{setup}). Therefore this difference could also be negative and the solution on stream could be worse than the solution with the buffering of all the input data. This happens because the

replication of the array A is done with repeated broadcast of grain g to all the processes instead of one broadcast of grain N. In this way we pay $n^2 T_{setup}$ instead of nT_{setup} and this factor could be greater than the time saved avoiding the buffering.

For the case where $T_A \geq nT_{send}(g)$, table 3.1 shows the mathematical differences between the completion time of the solution with total buffering and the solution with pure streaming.

$L_{execution+gather}$	Completion time difference
$nT_{send}(g)$	$\frac{N^2(n-1)}{n^2}(T_F + T_{transm}) + nNT_{transm}$
$g(N-g)T_F$	$nT_{setup} + (N + nN + \frac{N^2(n-1)}{n^2})T_{transm}$
$(n-1)T_{send}(g^2)$	$\frac{N^2(n-1)}{n^2}T_F + T_{setup} + (N + nN)T_{transm}$

Table 3.1: Completion time values in function of $L_{execution+gather}$

Considering the same n, in the streaming case we always have a positive difference and we are gaining time with respect to the solution with total buffering.

Now we analyse the memory usage in the streaming case. For sure, like in the data parallel case, any worker needs $N + g$ bytes to store A and its partition of B. Differently from the data parallel case, a worker needs just g^2 bytes to contain the results of the computation of F on g elements of A and g elements of B. This happens because in the streaming case we send results on the fly while in the data parallel case we wait for all the results. Memory per node in the streaming case becomes $O(N + g^2) = O(N + \frac{N^2}{n^2})$. As every pure streaming solution, the memory in the emitter must be evaluated knowing the values of T_A and $L_{scatter}$.
The total memory is $O(nN + \frac{N^2}{n}) = O(N^2)$.

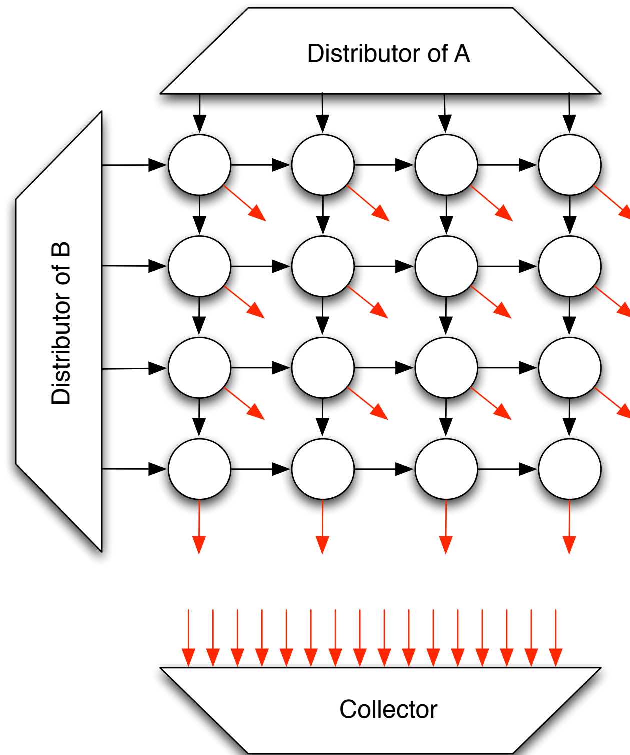


Figure 3.4: VP organization for the *stencil* with N^2 VPs

3.4 Stencil

In this section we utilize a *stencil* solution (shown in figure 3.4) to solve the problem. This solution is similar to the first solution with the *map* (cf. section 3.2) because there are N^2 virtual processors. Every virtual processor still receives only one element of A and one element of B to compute one element of R. The difference with respect to the *map* solution is in the way the two arrays are distributed.
The two emitters send messages only to a limited number of virtual processors; particularly each emitter will communicate with N VPs (one side of the square of virtual processors). When a virtual processor receives an element, forwards it to the next virtual processor on the same row or column depending on which input channel the element was received. A VP begins

its computation only after it has forwarded both the values it receives. The communications are static, fixed and happen only among neighbours.
With this solution we obtain $O(N^2)$ parallel computations and $O(N^2)$ communication channels:

- The emitter that distributes A has N output channels, one for every VP of the side of the square.
- The emitter that distributes B has N output channels.
- The VPs are interconnected through $O(N^2)$ channels.
- The collector has N^2 input channels, one for every VP.

In order to have n workers the computational grain can be defined as $g = \frac{N}{\sqrt{n}}$. Each worker will compute g^2 results using $2g$ input elements.
We spend now some time showing the behaviour of the *stencil* computation. Figure 3.5 depicts the order in which the input elements are received by the workers.
Each worker needs to wait that its corresponding input elements are sent by the emitters and then forwarded by all the previous workers on the same row and column. We can see that the last worker to start the computation is the one on the bottom right corner of the workers matrix. The workers on the same diagonal in the figure start the computation at the same time.

We need to compute the latencies of the communications in order to obtain the completion time of the computation.

$$L_{scatter} = \sqrt{n} T_{send}(g)$$

Both the emitters have to partition the array A (or B) sending elements of size g to $\sqrt{n}$ workers. Therefore we obtain a total of $\sqrt{n}$ send of size g.

$$L_{communications} = \sqrt{n} T_{send}(g)$$

Using a *stencil* we have to consider the impact of the communications among workers. Each worker forwards the messages received to the successive worker on the same row or column. For the completion time we just need to consider the entire path of the last element from the emitter: it has size g and will be forwarded among $\sqrt{n}$ workers. All the other communications are overlapped.

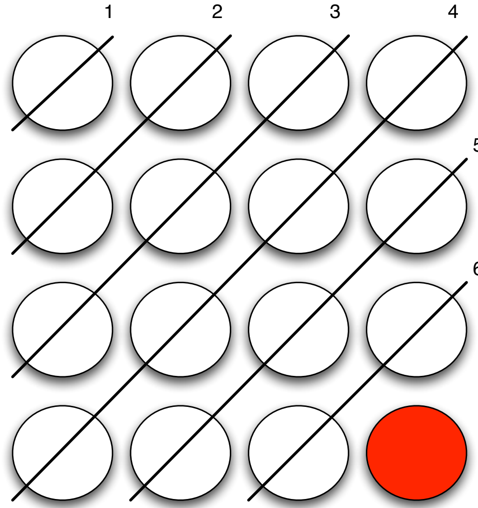


Figure 3.5: *Stencil* execution scheme

$$L_{gather} = T_{send}(g^2)$$

Also in this case we can consider just the latency of the last send to arrive to the collector because all the others are overlapped with the computation of the workers.
For the completion time we just need to consider the execution time of a worker once. All the other execution times are overlapped with the latency of the communications and the execution time of the n-th worker.

$$\boxed{T_C = L_{scatter} + g^2 T_F + \sqrt{n} T_{send}(g) + L_{gather}}$$

We have the following results for the memory requirement: the memory per VP consists just of the send and receive buffers while in the total memory we must consider the size of the inputs. Therefore the memory per worker is $O(g^2) = O(\frac{N^2}{n})$ and the total memory is $O(N + ng^2) = O(N^2)$.

3.4.1 Stencil: streaming and total buffering

At this point we suppose to receive and buffer all the elements of the two arrays before starting the computation. The size of the messages that the emitters receive on their stream is g.

$$\boxed{T_C = \sqrt{n}T_A + L_{scatter} + g^2T_F + \sqrt{n}T_{send}(g) + L_{gather}}$$

As in the *map* solution, the T_{Buffer} is equal to $\sqrt{n}T_A$, where T_A is the inter-arrival time from the stream. The input will be decomposed as $\sqrt{n}$ different messages because $g = \frac{N}{\sqrt{n}}$. The memory occupation is the same of the data parallel case.

3.4.2 Stencil: streaming only

In this subsection we try to exploit the stencil to compute results as soon as elements arrive on the input streams.
The emitters keep working in parallel and messages are of size g. Also the workers keep the same behaviour as in the data parallel case.
First, we study how the ratio between the inter-arrival time of the streams and the time to distribute an element to the worker impacts on the computation behaviour.

If $T_A < T_{send}(g)$, the time necessary to receive A and B is completely overlapped with the latency of the scatter because the emitter becomes a bottleneck for the computation. So, for the completion time, we consider the execution time of the last worker to receive the inputs, the latency of the gather, of the scatter and of the inter-workers communications. The emitters will need a buffer which size must be determined using queueing theory (expected number of users in the waiting line) because they cannot keep up with the inter-arrival time of the stream.
The completion time becomes

$$\boxed{T_C = T_A + \sqrt{n}T_{send}(g) + g^2T_F + \sqrt{n}T_{send}(g) + T_{send}(g^2)}$$

Considering the same n, using pure streaming we gain a factor $\sqrt{n}T_A$ with

respect to the solution with total buffering. We are gaining on the smallest factor of the completion time formula because $\sqrt{n}T_{send}(g) > \sqrt{n}T_A$.

If $T_A \geq T_{send}(g)$ all the distribution times, except the last, are overlapped with the time necessary to receive all the inputs from the streams. In order to obtain the completion time formula, we sum $\sqrt{n}T_A$ to the latency needed to distribute and forward the last elements of the streams and the latency that we must wait to compute the last result.
We obtain a completion time equal to

$$\boxed{T_C = \sqrt{n}T_A + T_{send}(g) + g^2T_F + \sqrt{n}T_{send}(g) + T_{send}(g^2)}$$

Therefore we can see that, considering the same n, in the streaming case we save $\sqrt{n}T_{send}(g)$. Also in this case we are gaining on the smallest factor because $\sqrt{n}T_{send}(g) < \sqrt{n}T_A$.

The memory used by the workers is the same as in the data parallel solution without streams.

3.5 Stencil with replication

In this section a *stencil* solution is still utilized, but we replicate the array A on all the nodes. Now there are N VPs and the computational grain is defined as $g = \frac{N}{n}$. In this case we have two emitters: one distributes g different elements of B to every worker and the other sends A to the first worker. The array A is forwarded by each worker to the next one in order to obtain the replication. Each worker behaves like this: first it waits to receive non deterministically A or its partition of B; as soon as it receives A it forwards it to the next worker; when the worker has forwarded A and received the partition of B, it begins its computation. In this way the communications are static, fixed and happen only among neighbours creating a pipeline distribution of the array A.
Using this stencil solution there are $O(N)$ computations in parallel. The

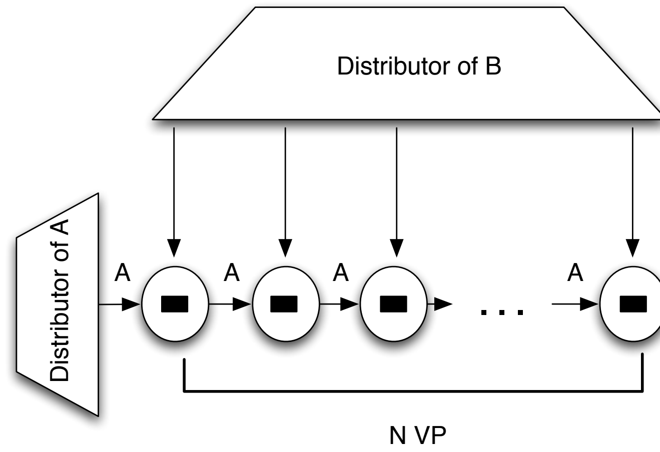


Figure 3.6: VP organization for the *stencil* with N VPs and replication of A

communication channels are $O(N)$:

- The distributor of B has N output channels, one per VP.
- The distributor of A has one output channel.
- The VPs are interconnected through $O(N)$ communication channels.
- The collector has N input channels.

In order to obtain the completion time of the total computation, we need to compute the latencies of the communications and observe if there are some overlapping.

$$L_{scatter} = T_{send}(N)$$

In this case one emitter sends all the array A to the first worker while the other emitter distribute the array B among all the workers. We should pay $T_{send}(N) + nT_{send}(g)$ for the latencies of the emitters, but as we will see, the second term can be omitted because it overlaps with the communications among workers.

$$L_{communications} = nT_{send}(N)$$

Each worker forwards the array A to the successive worker. Therefore the time spent to transmit the array A from the first to the last worker is $nT_{send}(N)$. This time is overlapped to the communication of the emitter which implements the scatter of B ($nT_{send}(g) < nT_{send}(N)$).

$$L_{gather} = T_{send}(gN)$$

We can consider just the latency of the last send to arrive to the collector because all the other are overlapped with the computation of the workers.
Finally, we just need to consider the execution time of the last worker. All the other execution time are overlapped with the latency of the communications and the execution times of the workers that we are considering.

$$\boxed{T_C = L_{scatter} + gNT_F + nT_{send}(N) + L_{gather}}$$

It is worth noting that the final formula for the completion time is equal to the formula for the map with replication (with a difference of one T_{setup}). Therefore, fixing the same parameters, we will have the same results of the map with replication in the case without stream.

Let us examine the memory requirements of this solution. Every VP buffers the entire array A plus a partition of the array B and the result of its computation. This last factor is the largest and thus we obtain a memory per worker which is $O(gN) = O(\frac{N^2}{n})$. For the total memory consumption we have to consider n times the memory required by a node plus the space necessary for the inputs. The total memory becomes $O(ngN) = O(N^2)$
We can note that also if the number of the worker is N the memory required by the workers will be at least $2N$ because A is entirely replicated and there are N results.

3.5.1 Stencil with replication: streaming and total buffering

Also in the case with stream and total buffering the formula for the completion time is equal to the formula for the map with replication.

$$\boxed{T_C = nT_A + L_{scatter} + gNT_F + nT_{send}(N) + L_{gather}}$$

Again, fixing the same parameters, we will have the same results of the map with replication.

3.5.2 Stencil with replication: streaming only

In this subsection we try to exploit the stencil with replication to compute results in the worker as soon as possible, that is as soon as the partition of B is received.
We still have two emitters working in parallel and messages of size g. One emitter distributes B among all the workers and the other sends the elements of A to the first worker. The elements of A are then forwarded by each worker to the next one in messages of grain g.
Now the workers have a different behaviour from the data parallel case. Each worker receives non deterministically from its two input streams and begins to compute F as soon as it has received its partition of B and at least one partition of A. When a result (g^2 elements) has been computed, the worker immediately sends it to the collector on its output channel.
Also in this case the ratio between the inter-arrival time of the streams and the time to distribute an element to the worker impacts on the computation behaviour.

If $T_A < T_{send}(g)$ a buffer will be needed in the emitters which become bottlenecks of the computation; its size must be decided carefully using queueing theory notions (expected number of users in the waiting line) and, in the worst case, can be up to all the input size (as the solution with total buffering).

Now we discuss the completion time. After the first inter-arrival time we have to consider the latency needed to distribute all the partitions of B in order to enable the computation on all the workers. The first part of the completion time therefore becomes: $L_{scatter} = nT_{send}(g)$.
When the last worker receives its partition of B it can start the computation utilizing the elements of A already received. For sure it does not have the last partition of A yet: it has to wait the forwarding of the message among the workers which takes $(n-1)T_{send}(g)$. As in the case of *map* with replication (cf. section 3.3.2) we can consider a maximum between three different times that overlap.

- The first is the time that the n-th worker waits to receive the last partition of A ($(n-1)T_{send}(g)$).

- The second time is the execution time of the n-th worker that have to produce gN results.

- Finally, we consider the time necessary that the worker takes to send all the results to the collector.

The worker sends a result as soon as possible obtaining a gather time equal to $nT_{send}(g^2)$. In any case we must consider that the computation (g^2T_F) and the collection ($T_{send}(g^2)$) of the last result cannot be overlapped.

$$
\begin{aligned}
&L_{execution+gather} = \\
&= \max((n-1)T_{send}(g), g(N-g)T_F, (n-1)T_{send}(g^2)) + g^2T_F + T_{send}(g^2) \\
&= \max(g(N-g)T_F, (n-1)T_{send}(g^2)) + g^2T_F + T_{send}(g^2)
\end{aligned}
$$

We can omit $(n-1)T_{send}(g)$ from the maximum because $(n-1)T_{send}(g) < (n-1)T_{send}(g^2)$.

If $T_A \geq T_{send}(g)$ then the *stencil* can keep up with the arrivals so, for the completion time, we have to consider the total input arrival time to the

emitters instead of the $L_{scatter}$. For the execution and the collection time the same reasoning of the first case still applies.

$$T_C = \begin{cases} T_A + nT_{send}(g) + L_{execution+gather}, & \text{if } T_A < T_{send}(g) \\ nT_A + L_{execution+gather}, & \text{if } T_A \geq T_{send}(g) \end{cases}$$

Now let us observe the difference between the solution with total buffering and the pure streaming one. If n is the same, we can compare the formulas of the completion time.
The mathematical difference between the completion time of the solution with total buffering and the solution with pure streaming represents the time gained (if positive) or lost (if negative) adopting the second solution.

For the case where $T_A < T_{send}(g)$, this difference is shown in table 3.2.

$L_{execution+gather}$	Completion time difference
$g(N-g)T_F$	$nT_A + T_{setup} + (nN + \frac{N^2(n-1)}{n^2})T_{transm}$
$(n-1)T_{send}(g^2)$	$nT_A + (2-n)T_{setup} + nNT_{transm} + \frac{N^2(n-1)}{n^2}T_F$

Table 3.2: Completion time values in function of $L_{execution+gather}$

We can see that, when $g(N-g)T_F$ is the maximum and we consider the same n, we always have a positive difference and in the streaming case we obtain a smaller T_C with respect to the solution with total buffering. On the other hand when $(n-1)T_{send}(g^2)$ is the maximum, the difference could also be negative and the solution on stream could be worse than the solution with the buffering of all the input data. This happens because the replication of the array A is obtained using n sends of grain g to the first worker instead of one send of grain N. In this way we pay nT_{setup} instead of T_{setup} and this factor could be greater than the time saved avoiding the buffering.

For the case where $T_A \geq T_{send}(g)$, the difference is shown in table 3.3.
We can see that, considering the same n, in the streaming case we always

$L_{execution+gather}$	Completion time difference
$g(N-g)T_F$	$(n+1)T_{setup} + (N + nN + \frac{N^2(n-1)}{n^2})T_{transm}$
$(n-1)T_{send}(g^2)$	$2T_{setup} + (N + nN)T_{transm} + \frac{N^2(n-1)}{n^2}T_F$

Table 3.3: Completion time difference values in function of $L_{execution+gather}$

have a positive difference and the computation has a smaller completion time.

Now we examine the memory requirements of the *stencil* computation with replication. Each worker needs $N + g$ bytes to store the whole array A and its partition of B. While in the data parallel case without streaming a worker needs gN bytes to store all the results, in this case just g^2 bytes are needed. This happens because in the streaming case we send results on the fly while in the data parallel case we wait to have all the results.

3.6 Comparisons and numerical examples

In this section we compare all the studied solution (figure 3.1) from the point of view of the memory occupation and the completion time.
For every performance parameter that we want to analyse, the ratio between the inter-arrival time T_A of the streams and $L_{scatter}$ is to be taken under consideration because it determines if the emitter is a bottleneck or not. At this point it is important to note that using the shedding and dropping techniques (cf. section 1.2.2) we can modify the inter-arrival time. In this way we can switch from the case $T_A < L_{scatter}$ to the other one (but not vice versa). Obviously, some kind of problems cannot accept the result's approximations that these techniques induce.

3.6.1 Memory occupation

First let us look at the memory consumption. We can affirm that with a pure streaming solution we are saving in the emitter at most the size of the

input data. This best scenario verifies when $T_A > L_{scatter}$ and the emitter needs only the receive buffer of size g for the input elements. In the other case ($T_A < L_{scatter}$), our parallel module becomes a bottleneck and a bigger buffer will be needed in the emitter. The size of this buffer depends on different factors (streams inter-arrival time distribution, latency of the scatter, execution time of the workers) and can be evaluated considering the module as a queueing system. If the buffer size is not big enough, it can force the emitter to drop some inputs.
The memory required by a worker for the computation in the streaming cases remains the same as in the data parallel case. What may change is the size of the receive and send buffers in the workers. In the streaming solutions we tried to minimize as much as possible the total memory of the parallel module by adapting the grain of the computation to the grain of the stream. An example of this can be seen in the *map* with replication (section 3.3.2) and *stencil* with replication (section 3.5.2). In the data parallel and streaming with buffering solutions we reduced the number of messages by replicating the array A in just one message per worker. In the pure streaming solutions a worker sends many smaller results to the collector instead of one bigger message containing all the results the worker computed, avoiding additional buffering in the workers (see tables 3.4, 3.5 and 3.6).

Map/Stencil	Memory occupation	
	DP and Buffering	Pure Streaming
Worker (n)	$2\frac{N}{\sqrt{n}} + \frac{N^2}{n}$	$2\frac{N}{\sqrt{n}} + \frac{N^2}{n}$
Emitter (2)	N	$\frac{N}{\sqrt{n}} \leq x \leq N$
Total	$2\sqrt{n}N + N^2 + 2N$	$2\sqrt{n}N + N^2 + 2x$

Table 3.4: Memory occupation comparison without using replication

Map with replication	Memory occupation	
	DP and Buffering	Pure Streaming
Worker (n)	$\frac{N}{n} + N + \frac{N^2}{n}$	$\frac{N}{n} + N + \frac{N^2}{n^2}$
Emitter (1)	$2N$	$2\frac{N}{n} \leq x \leq 2N$
Total	$3N + nN + N^2$	$N + nN + \frac{N^2}{n} + x$

Table 3.5: Memory occupation comparison using Map with replication

Stencil with replication	Memory occupation	
	DP and Buffering	Pure Streaming
Worker (n)	$\frac{N}{n} + N + \frac{N^2}{n}$	$\frac{N}{n} + N + \frac{N^2}{n^2}$
Emitter (2)	N	$\frac{N}{n} \leq x \leq N$
Total	$3N + nN + N^2$	$N + nN + \frac{N^2}{n} + 2x$

Table 3.6: Memory occupation comparison using Stencil with replication

3.6.2 Completion time

In this subsection we analyse the completion time of all the solutions adopted. As said in the opening of this chapter, we wanted to study the behaviour of a stream computation considering a single window because the completion time obtained can be seen as the service time of the computation operating on a stream of elements in the tumbling windows model.

In order to have some numerical validation to our ideas and have deductions from real data, we fix some values for our problem:
$T_{setup} = 10^3\tau$
$T_{transm} = 10^2\tau$
$N = 10^3\ bytes$
$T_F = 64 * 10^2\tau$
$T_{C_{seq}} = 6,4 * 10^9\tau$
where τ is the length of the clock cycle.
Tables 3.7, 3.8, 3.9 and 3.10 show the completion time for each form of parallelism studied: *map*, *stencil*, *map* with replication and *stencil* with replication.

In every case we first computed n_{buf} which is the optimal parallelism degree to minimize the completion time in case of total buffering. Then, in order to make a comparison, this parallelism degree was utilized to compute the completion time in the case with pure streaming.
For the pure streaming solution, we also tried to obtain the best completion time possible finding the optimal parallelism degree n_{min}. In some cases n_{buf} and n_{min} are the same (or very similar) and the completion time does not change.

Inter-arrival time	Parallelism degree	Map	
		Buffering	Pure Streaming
$T_A = 10^3$	$n_{buf} = n_{min} = 1681$	$9,6 * 10^6$	$9,6 * 10^6$
$T_A = 10^6$	$n_{buf} = 484$	$3,8 * 10^7$	$3,5 * 10^7$
	$n_{min} = 529$		$3,5 * 10^7$

Table 3.7: Completion times comparison using Map without replication

Inter-arrival time	Parallelism degree	Stencil	
		Buffering	Pure Streaming
$T_A = 10^3$	$n_{buf} = 26569$	$9,3 * 10^5$	$7,7 * 10^5$
	$n_{min} = 34596$		$7,6 * 10^5$
$T_A = 10^6$	$n_{buf} = n_{min} = 529$	$3,5 * 10^7$	$3,5 * 10^7$

Table 3.8: Completion time comparison using Stencil without replication

Inter-arrival time	Parallelism degree	Map with replication	
		Buffering	Pure Streaming
$T_A = 10^3$	$n_{buf} = 252$	$5,1 * 10^7$	
	$n_{min} = 130$		$7,9 * 10^7$
$T_A = 10^6$	$n_{buf} = n_{min} = 76$	$1,6 * 10^8$	$1,6 * 10^8$

Table 3.9: Completion times comparison using Map with replication

From the results obtained we can see a general behaviour in the buffering solutions. When the inter-arrival time is small ($T_A = 10^3$), the total buffering

Inter-arrival time	Parallelism degree	Stencil with replication	
		Buffering	Pure Streaming
$T_A = 10^3$	$n_{buf} = 252$	$5,1 * 10^7$	$2,5 * 10^7$
	$n_{min} = 2509$		$5,1 * 10^6$
$T_A = 10^6$	$n_{buf} = n_{min} = 76$	$1,6 * 10^8$	$1,6 * 10^8$

Table 3.10: Completion time comparison using Stencil with replication

time is negligible with respect to the completion time of the computation. By increasing the inter-arrival time, n_{buf} decreases. This happens because T_{buffer} becomes the main factor in the completion time and the parallel module slows down to adapt.

Comparing the pure streaming completion time with the buffering one, different cases can occur, as our examples highlighted.

- The two completion times are quite the same.
- One of the two completion times has a better scale factor than the other, but the order of magnitude remain the same.
- One of the two completion times outperforms the other by an order, or more, of magnitude

In the examples presented the second case is the more common, with the pure streaming solution performing better then the buffering one, however all the cases have occurred.

We discuss here the reasons to all the behaviours.
In the cases without replication, we utilized the same distribution and collection schemes in both the solutions with total buffering and pure streaming. However in the pure streaming solution, we computed the results in the workers as soon as the input elements were available. Therefore as predicted, if $T_A < L_{scatter}$ we save the time necessary to buffer all the input elements (T_{buffer}), otherwise we are able to overlap the distribution time with inter-arrival time. As pointed out many times previously, we always save the

smallest factor between the buffering time and the distribution time.

In the *stencil* with replication case, when $T_A < L_{scatter}$, comparing the solutions with their optimal n, we can see that the pure streaming solution saves an order of magnitude on the completion time with respect to the solution with total buffering. This happens because of the computation reorganization in the distribution and collection phases. Indeed we were able to overlap in every worker the computation with the internal communication obtaining a pipeline effect (cf. section 3.5.2).

In the solutions exploiting replication, the pure streaming case could be worse than the one with the buffering of all the input data. This happens because the replication of the array A is obtained using many sends of smaller grain instead of one send containing the whole array. In this way we pay a larger time for the setup of the communications and this factor could be greater than the time saved avoiding the buffering.
In particular, in the *map* with replication case when $T_A < L_{scatter}$, we obtain a completion time with the same order of magnitude (10^7) but a worst constant even if we are using about half the workers of the solution with buffering. Using the parallelism degree of the solution with total buffering we would obtain a worse completion time even if the parallelism degree is greater. As we know having a greater parallelism degree doesn't imply a smaller completion time. (In this case it is worth noting that also the grain g of the input data is changed and the total arrival time.)

Chapter 4

Parallel Stream Join

In this chapter we present the *join* problem over two streams of data. This kind of computation is very common in Data Stream Management System (cf. section 1.1).
The main strategy to evaluate stream join is still sequential, but some parallel solutions have been presented in literature [GBY09, TM11].
We are interested in stream join because the computation works with time-based sliding windows. We show how the methodology and the parallel paradigms proposed for count-based computations can be adapted to solve this problem.
We present the Data Stream Processing application developed during our thesis (*parJoin*) and a comparison with an existing solution. Our implementation targets shared memory architectures.

4.1 Stream Join semantics

The problem we want to address is the *join* on two streams of elements. Given two elements x and y, we denote as join the generic computation $z = F(x, y)$. For example the typical case of an *equi-join* is defined as follows:

$$F(x,y) = \begin{cases} x \circ y & \text{if } x.attr = y.attr \\ nil & \text{if } x.attr \neq y.attr \end{cases}$$

that is, the result of the *join* is the concatenation of the two elements if and only if the attribute *attr* is equal for the two tuples; no result is produced otherwise. Obviously it is possible to apply more complex operators to the computation.
In the DaSP context, the join operation between elements of two streams R and S (potentially infinite) is realized using the concept of sliding windows. In the count-based version the window size and the slide are expressed in number of elements (tuples). We will focus instead on the time-based version, in which the window size T and the slide are expressed in temporal duration (every element x of the stream has associated a time-stamp attribute t_x representing the instant of reception). Therefore the number of elements that belong to a window is not fixed: it is dependent on the rate of arrival of the stream.

When the join is computed over two sliding-windows streams the computation is performed on all the tuples contained in the last window of the two streams. With last window we mean the window containing all the elements received in the last period of duration T (window size).
The exact semantics for sliding-windows joins can be derived by the three-step procedure presented in [KNV03] as described in [TM11].
For $x \in R$ and $y \in S$, the tuple $x \circ y$ belongs to the join result $R \bowtie_p S$ iff:

- x arrives after y ($t_x > t_y$) and y is in the current S window when x arrives (i.e. $t_x < t_y + T_S$)
- x arrives earlier than y ($t_y > t_x$) and x is still in the current R window when y arrives (i.e. $t_y < t_r + T_R$)

and x and y pass the join predicate p.

Figure 4.1 shows two streams R and S with sliding windows of size T_R and T_S respectively. Following the semantics explained above, the join computation over the elements of the two streams will produce the following matchings (if they pass also the predicate p):

- $x_1 \circ y_1$ because x_1 arrives after y_1 and y_1 is in the current S window when x_1 arrives (rule *(a)*)

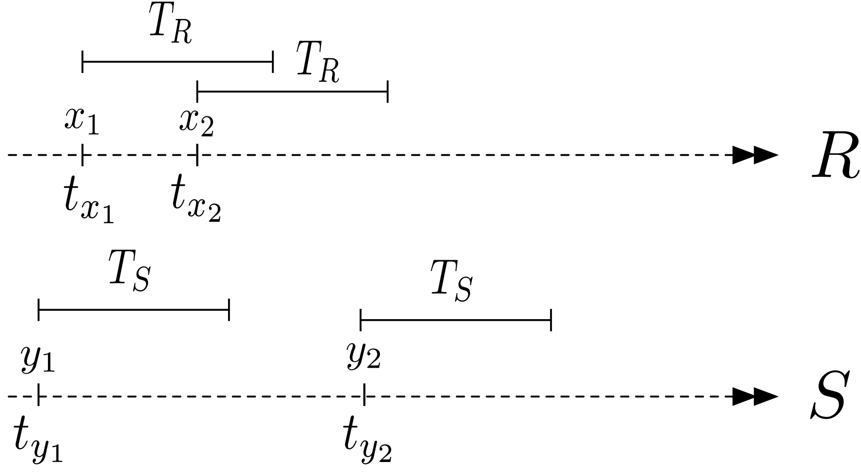


Figure 4.1: Stream Join example

- $x_2 \circ y_1$ because $t_{x_2} > t_{y_1}$ and $t_{x_2} < t_{y_1} + T_S$ (rule *(a)*)
- $x_2 \circ y_2$ because x_2 arrives earlier than y_2 and x_2 is still in the current R window when y_2 arrives (rule *(b)*)

$x_1 \circ y_2$ cannot be a result because $t_{y_2} > t_{x_1}$ but $t_{y_2} > t_{x_1} + T_R$.

4.2 Sequential Algorithm

Stream join algorithm is composed by three main steps (Kang et al. [KNV03]). In order to explain these phases, we suppose to have just received an element x from the stream R (elements of the stream S are named y):

1. **insert** x in stream R window.
2. **scan** S window and for every y if $t_x < t_y + T_S$ (meaning that the element is not expired) **compute** the join of x and y.
3. **remove** all the expired elements in S window.

Kang's original algorithm removes expired elements from stream R window, instead of S window. Note that elements are always compared by their timestamps before computing the join, therefore the two solutions are semantically equivalent (cf. section 4.1). However our solution presents an advantage: both, the join and remove operations (steps **2** and **3**), can be performed in the same scan of the S window. If the elements are inserted in the window ordered by their time-stamp and we begin to scan from the most recent item, when we find an expired element we can discard also all the successive because they have been received earlier. In this way we avoid useless comparisons and useless scans too: Kang's algorithm would scan S window to compute the join and then the R window to remove expired elements while we just scan S window once. If an element y from the stream S is received, the operations performed are symmetrical and equivalent.

4.3 DaSP module formalization

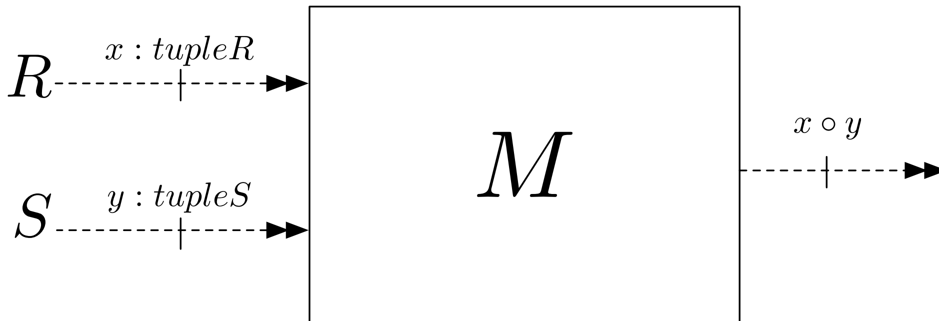


Figure 4.2: The Module M: its inputs and its output

We can imagine the DaSP module M that has two input streams, R and S respectively; the elements coming from R will have type *tupleR*, and the ones coming from S will have type *tupleS* (figure 4.2). The two input streams are characterized by two inter-arrival times λ_R and λ_S.
M produces an output stream that consists of pair of tuples, one *tupleR* and one *tupleS*. We describe the behaviour of the module as follows:

```
TupleR windowR[]; TupleS windowS[]; channel in ch_R,ch_S;
  channel out ch_res;
TupleR x; TupleS y;
alternative{
  receive(ch_R,x) do
    {
      windowR = windowR ∪ x;
      for each y ∈ windowS {
          if(F(x,y,WINSIZER,WINSIZES))
            send(ch_res ,(x,y));
        }
    }
  or receive(ch_S,y) do
    {
      windowS = windowS ∪ y;
      for each x ∈ windowR {
          if(F(x,y,WINSIZER,WINSIZES))
            send(ch_res ,(x,y));
        }
    }
}
```

windowR and *windowS* are the windows of the two streams containing non expired tuples.
The stream join sequential algorithm, described in section 4.2, is implemented by the function F. From now on, we just assume that this function outputs true with probability p (also called *hit-rate*), and false with probability $1-p$. The F function takes four parameters which are two tuples, coming from different streams, and the size in seconds of the two windows. These two parameters are user-defined fixed value, so we just described them as constant in the pseudo-code. We will discuss the detailed implementation of F in the next chapter, because here we just want to focus on the module M.
We ask ourselves if it is possible to find the ideal maximum output rate B_{id_M} that M can reach. In the classical theory of graph computations [Van09, Van12], the simplest case is the one in which there is only one stream and to every input corresponds exactly one output. In that case the ideal maximum output rate of the module is equal to the input rate.

The stream join problem is more complex because we have two streams, the number of outputs is in general quadratic with respect to the input elements and it depends on the window sizes and the *hit-rate*. For this reason we are going to find a mathematical formula to evaluate B_{id_M}.
Intuitively, given a time interval $[t_0, t_1]$, the maximum output rate is the one that allows the module to output all the results computed for $[t_0, t_1]$ just at the end of the time interval. So, in order to find B_{id_M}, we must look at the number of results generated by the inputs received in $[t_0, t_1]$. However this number depends on many parameters like λ_R, λ_S, p, $t_1 - t_0$ and the average number of elements that the windows will contain.
Let's start from this last parameter: we will name $||W_R||$ ($||W_S||$) the average number of *tupleR* (*tupleS*) present in *windowR* (*windowS*) at any moment, and $|W_R|$ the size in seconds of *windowR*(*windowS*). Therefore:

$$||W_R|| = |W_R| \cdot \lambda_R$$

Now we are ready to define the maximum number of outputs that M can compute in $[t_0, t_1]$ supposing both windows have respectively $||W_R||$ and $||W_S||$ elements at t_0:

$$\begin{aligned} \Omega_{t_0,t_1} &= [\lambda_R(t_1 - t_0)||W_S|| + \lambda_S(t_1 - t_0)||W_R||)] \times p \\ &= \lambda_R\lambda_S(t_1 - t_0) \cdot (|W_S| + |W_R|) \times p \end{aligned} \tag{4.1}$$

$\lambda_R \cdot (t_1 - t_0)$ is the average number of *tupleR* received in $[t_0, t_1]$; multiplying this number for $||W_S||$ we obtain the expected number of execution of F triggered by elements received from R. We can set up a similar reasoning for the second addend in the formula for *tupleS*. At this point it is clear that the first factor in Ω_{t_0,t_1} represents the average number of F computations triggered by the tuples received in $[t_0, t_1]$. By multiplying it for the *hit-rate* p we find the average number of outputs that M will produce for the inputs received in $[t_0, t_1]$.
Now we can easily found B_{id_M}:

$$B_{id_M} = \frac{\Omega_{t_0,t_1}}{t_1 - t_0} = \lambda_R\lambda_S \cdot (|W_S| + |W_R|) \times p \tag{4.2}$$

Using (4.2) we are able to know whether or not M is a bottleneck: if the

output rate of M is smaller than B_{id_M} then the module is a bottleneck and it can be parallelized internally, if otherwise the output rate is equal to B_{id_M} then M is not a bottleneck. We will validate in section 4.7 all the formulas showed in this section.

4.4 Existing implementations

4.4.1 CellJoin

An interesting scalable parallel solution to the stream join problem for the Cell Processor has been described in [GBY09]. The Cell processor is a single chip multiprocessor with nine processing elements that share a coherent memory. Eight of the cores, called Synergistic Processing Elements (SPEs), are 128-bit RISC processor specialized for big-data, compute-intensive SIMD applications. Each SPE has a 256KB private local cache to hold both instruction and data; the element accesses the memory through direct memory access commands. The ninth PE is the Power Processor Element (PPE): a general purpose dual-threaded 64-bit RISC Processor that is connected to the main memory through two level of caches. In addition to coherent accesses to the main memory, there are several other ways for the PPEs and the SPEs to communicate with each other, such as mailboxes and signals. For an extensive description of the Cell processor see [Gsc07, Gsc06].
In [GBY09] the stream join is parallelized by replicating the last received tuple to each SPE and partitioning the join window of the other stream among the SPEs. This happens dynamically for each SPE when it receives tuples regardless of the stream. PPE takes care of storing received tuples in windows in the main memory and of their removal. In order to increase the grain of the computation, the authors exploited batching on partitioned tuples. In this way a join window is transferred once from the main memory to the local stores of SPEs for each unit of job. Each SPE is assigned a partition of the join window in parallel and the authors claim that the load is well-balanced, independently of the number of tuples fetched. Finally, the results are collected at the PPE-side.
This solution is designed for a specific architecture (Cell processor) and it

would be inappropriate to compare the results obtained on a different system. Furthermore, the Cell processor has only eight SPEs and it is not possible to evaluate the scalability of the solution using more processing units.

4.4.2 Handshake join: How soccer players would do stream joins

In [TM11], Teubner et. al present *handshake join*, a parallel scheme for time window-based stream join.

The basic idea behind this solution is illustrated in figure 4.3 (adopted from

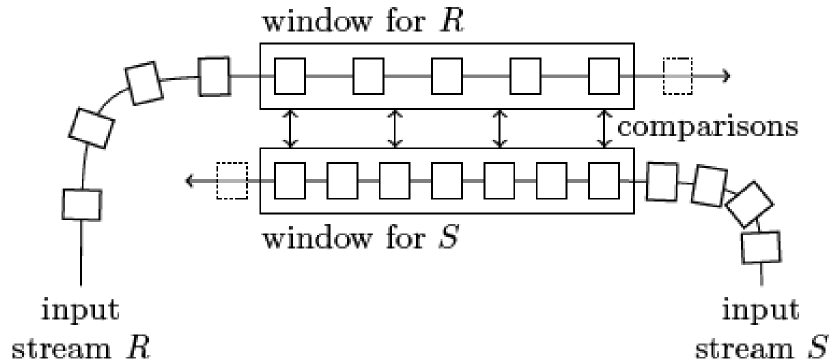


Figure 4.3: Handshake join

[TM11]). The two streams of tuples flow in opposite directions and the comparisons happen in the middle, as the tuples from different sources encounter each other. The authors compare this behaviour with soccer players walking by each other and shaking hands with every player of the opposite team at the beginning of a game.

This solution, like our proposal, produces out of order outputs. Nevertheless, the comparisons performed follow the semantics of the stream join presented in section 4.1. At the arrival, a tuple enters the comparisons area (corresponding to the window of the stream) and it exits when expires.

Each tuple that flows in a window pushes the others toward the end. When a tuple of a stream encounters a tuple of the other stream (moving in the

opposite direction) the comparison is performed. These comparisons can be parallelized because more than one *handshake* happens at the same time.

In order to parallelize the handshake join over the available computing

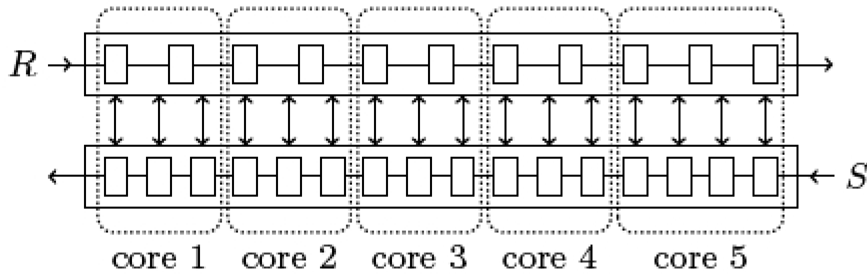


Figure 4.4: Handshake join parallelization

resources, R and S windows are partitioned among the processing units as shown in figure 4.4 (adopted from [TM11]).
Basically this parallel scheme can be seen as two pipelines paradigms flowing in opposite directions. The tuples comparisons are performed locally inside each worker. The communications are static, fixed and happen only among neighbours.
This solution proposed by Teubner et al. arises a synchronization problem. If two tuples flowing in opposite directions are sent on the communication channels by neighbouring workers at the same time, they might never encounter each other inside a worker. In this case the two tuples will never be compared and there would be some outputs missing. In order to solve this problem, *handshake join* implements a communication protocol based on acknowledgement messages sent between workers in addition to the tuples.
Another issue addressed by *handshake join* is the load balancing problem. In order to keep even the amount of tuples in each worker, tuples forwarding is based on local windows size. Each worker checks its own local window size and, when it becomes bigger than the successive worker window size, forwards a tuple. This strategy ensures that all the workers perform roughly the same number of comparisons.
This solution is not designed for a specific system and the source code is available at [TM]. Therefore it is possible and meaningful to compare the

results obtained using a different solution on the same architecture. Furthermore, it is also possible to evaluate the scalability of the solution using any number of core available.

4.5 Proposed parallel solutions

In this section we propose our parallel solution for the stream join algorithm. A possible way to parallelize the computation consists in utilizing a partitioning and/or a replication of the windows over the available processing units. To do that, we adopted the so-called *Virtual Processors* approach [Van09, Van12] that, starting from the sequential computation, is able to derive the basic characteristics of a data parallel computation such as the identification of a *stencil* and its shape.
Using the virtual processor approach we derive an abstract representation of the equivalent data parallel computation which is characterized by the maximum theoretical parallelism degree compatible with the computation semantics. Because it is a maximal parallelism degree version, the grain size of data partitions is the minimal one. However we must not forget that data stream processing have some important differences with respect to the standard data parallel paradigm: our data structures, consisting of the two input streams, are potentially infinite and we must consider the concept of window too.
The minimal computational grain is the join of two tuples (figure 4.5).

Each virtual processor is the computation of one result starting from two tuples. These tuples must have *compatible* timestamps, i.e. they should not have been expired. If we suppose that all the input tuples are available at the beginning of the computation, all the virtual processors can be executed in parallel because there are no data dependencies for the computations of different join results.
Actually the tuples arrive at a certain time and each comparison becomes possible after a specific moment. We have to assign each virtual processor (one comparison) to a real worker, and this assignment has to be dynamic because all the input tuples are not immediately available. Furthermore,

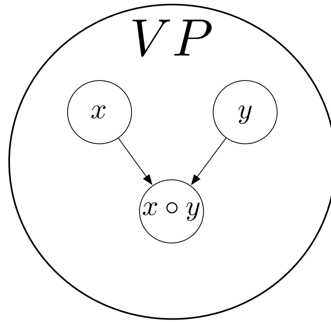


Figure 4.5: VP with minimal grain of data

the stream is potentially infinite, as the number of windows; this implies an infinite number of comparisons and thus an infinite number of VPs to be assigned to a finite number of real workers. Each virtual processor will be assigned to one and only one worker.

It is important to note that many VPs share the same data and some input

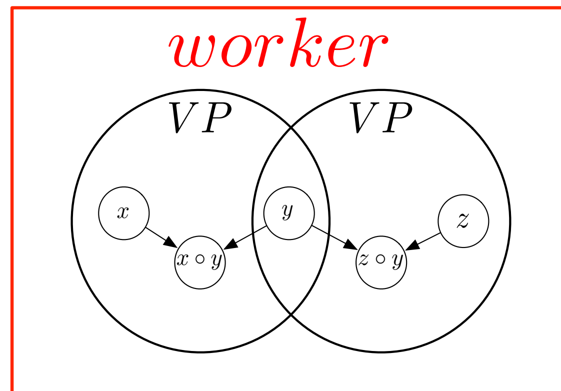


Figure 4.6: Two VPs sharing a tuple executed in the same worker

elements have to be replicated among different VPs. When mapping the virtual processors on real workers, we must consider this fact in order to reduce the replication and increase the locality of data partitions in the workers (figure 4.6). For this reason we will try to allocate on the same worker VPs that shares some data.

We can utilize a *stencil* solution similar to the one proposed in section 3.4

and adapt it for the *time-based* sliding-window join problem. The input elements needed for the join computations in a single worker must be carefully chosen in order to reduce the replication of input tuples.
The solution proposed in section 3.4 was intended to be implemented in each macro worker of the DaSP module. The distributor was used to transform the input stream in order to have tumbling windows from the point of view of the workers. Each worker internally was implemented as a *stencil*.
For the parallel stream join instead, we utilize just one macro-worker implemented as a *stencil*.

Figure 4.7 summarizes what we are going to describe in the next sections.

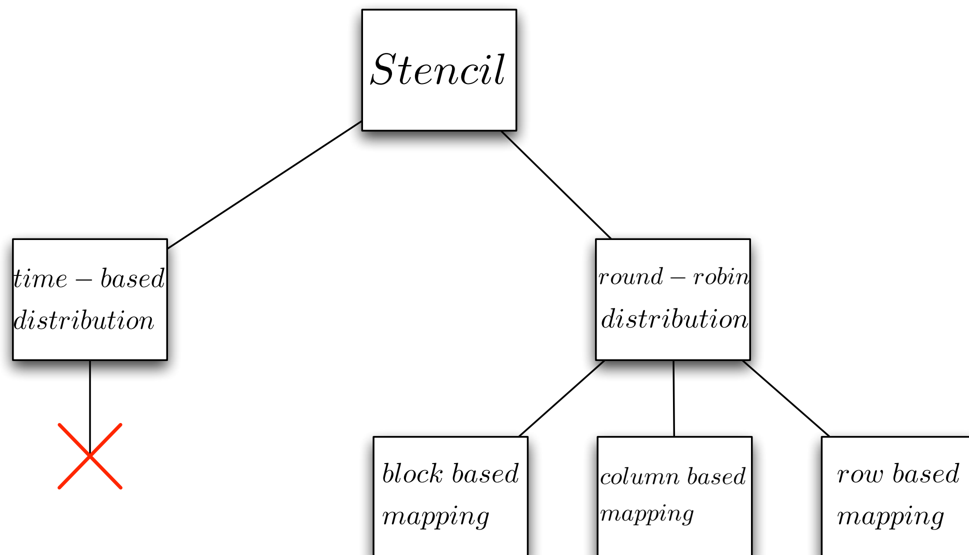


Figure 4.7: parJoin different implementations

First we are going to present a solution for the distribution of the input elements which is simple but suffers of performance issues. Then we describe a second distribution technique, based on a round-robin strategy, that is the one we actually implemented in parJoin.
We utilized different mappings of input data onto workers, represented by the leaves of the tree.

4.5.1 Time-based distribution solution

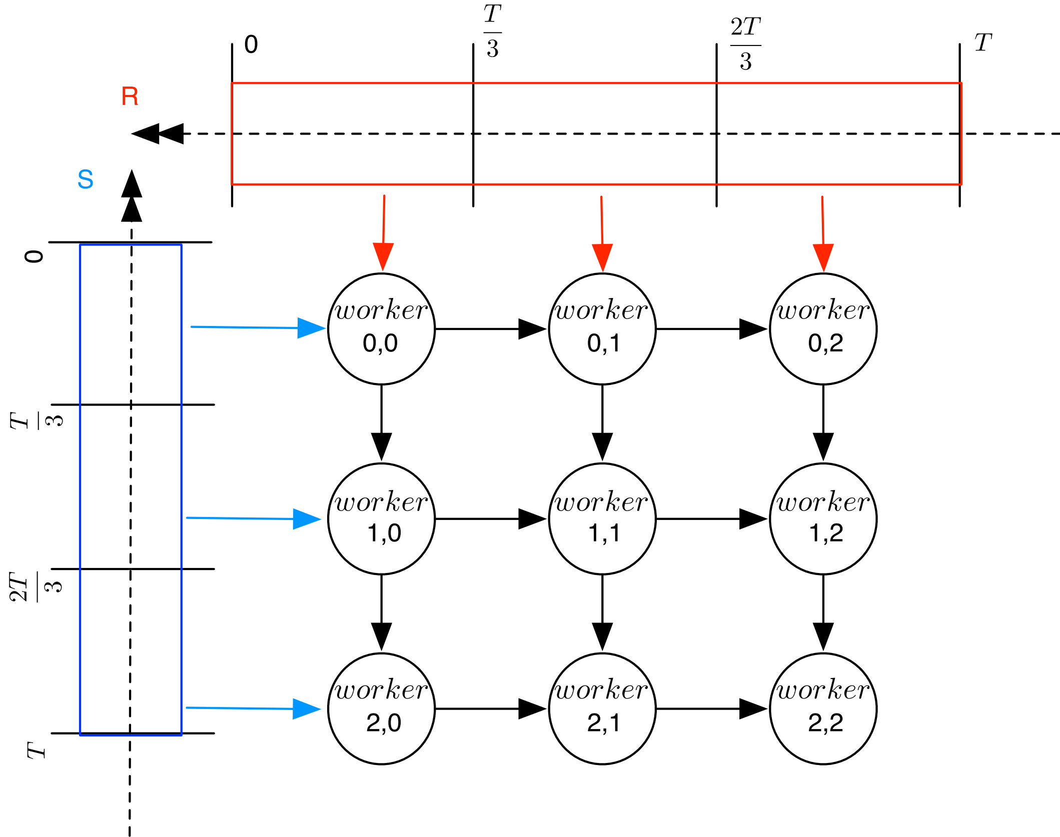


Figure 4.8: Partitioning of a window among workers

In the *count-based* version of the problem the number of elements in each window is fixed and well-known because it is a user-defined input of the problem itself: W.

The main difference in the *time-based* case is that we cannot predict the number of elements that will be present in a window at any given time. This happens because the window is bounded by time and not by a number of elements. The rate of arrivals is also unknown to the module or can change over time.

A possible solution is to directly start with the workers, setting their number arbitrarily (e.g. N^2). As shown in figure 4.8, we can logically split the

time-based window, whose size is T, in N same sized slot.
As already discussed in chapter 2, the distribution of the input elements is a crucial phase of the parallelization because it can impact on the efficiency and the resources utilization.
Consider the R stream. The distribution will partition the input elements over the columns and utilize replications over the rows of workers. $worker_{*,j}$ will receive elements of the stream that belongs to $[Tk + j\frac{T}{N}, Tk + (j+1)\frac{T}{N})$ time-slots with $k \in \mathbb{N}$. Therefore a certain number of contiguous elements will be received by the same worker. The *sliding* behaviour of the window has to be implemented by each worker with the removal of expired elements. Unfortunately, this distribution does not guarantee a balanced partitioning of the tuples; the number of elements that a worker receives in different slots is not necessarily the same because the inter-arrival time may suffer of some variations. Moreover, as a consequence of the time-based distribution, if the slot size is very large with respect to the computation time of the workers this parallel paradigm may not exploit its full potential. In the worst case, at every time instant, this computation would have only one row and one column of workers active in parallel.

The behaviour of a generic worker is clarified by the following pseudo-code:

```
set Join&Remove(Tuple x, Set Y){
  init W = ∅;
  for each y ∈ Y {
    if (x.timestamp < y.timestamp){
      if(y.timestamp − x.timestamp < T_R)
        W = W ∪ Join(x, y);
    }
    else{
       if(x.timestamp − y.timestamp < T_S)
        W = W ∪ Join(x, y);
      else
        Y = Y \ y;   //remove the element
      }

  }
```

```
  return W;
}

Worker i,j::
  init: X = ∅ , Y = ∅, Z = ∅;

  alternative{
    guard_1, receive(IN[i-1, j], x) {
      X = X ∪ x;               //INSERT in ORDER
      send(OUT[i+1, j], x);  //FORWARD
      Z = Join&Remove(x, Y);//JOIN and REMOVE
      send(RIS[i, j], Z);
    }

    guard_2, receive(IN[i, j-1], y) {
      Y = Y ∪ y;               //INSERT in ORDER
      send(OUT[i, j+1], y);  //FORWARD
      Z = Join&Remove(y, X);//JOIN and REMOVE
      send(RIS[i, j], Z);
    }
  }
```

The code describes faithfully the algorithm previously presented in section 4.2; we added the communications between the workers. As can be seen in the comments, we suppose that the insertion of the elements in the windows is performed in order.
We can note that every worker needs to store in memory its local windows X and Y. The size of these buffers cannot be known a priori because it depends on the inter-arrival times of the streams and on the number of unexpired received elements.

4.5.2 Round-robin distribution solution

In order to solve load balancing and under-usage issues of the previous parallel paradigm, we propose a modification. What we are going to change is the distribution scheme without altering the workers behaviour.
The emitter will now distribute the elements from the stream to the worker

in a simple yet effective round-robin fashion (figure 4.9). At any given instant of time, the number of elements sent to each worker will be the same ($\pm$ one element) and will not be influenced by the inter-arrival times of the input streams. Furthermore, all the workers computing on the same number of elements will have very similar service times because the join function does not have a great variance in time. In this way, the distribution will ensure load balancing.

The final distribution technique utilized is the one with multiple streams presented in section 2.2.2. There are two logically independent distributors, one for each stream, utilizing a round-robin strategy. Each worker still has two input channels as in the sequential case.

We now want to be sure that the semantics of the computation is still cor-

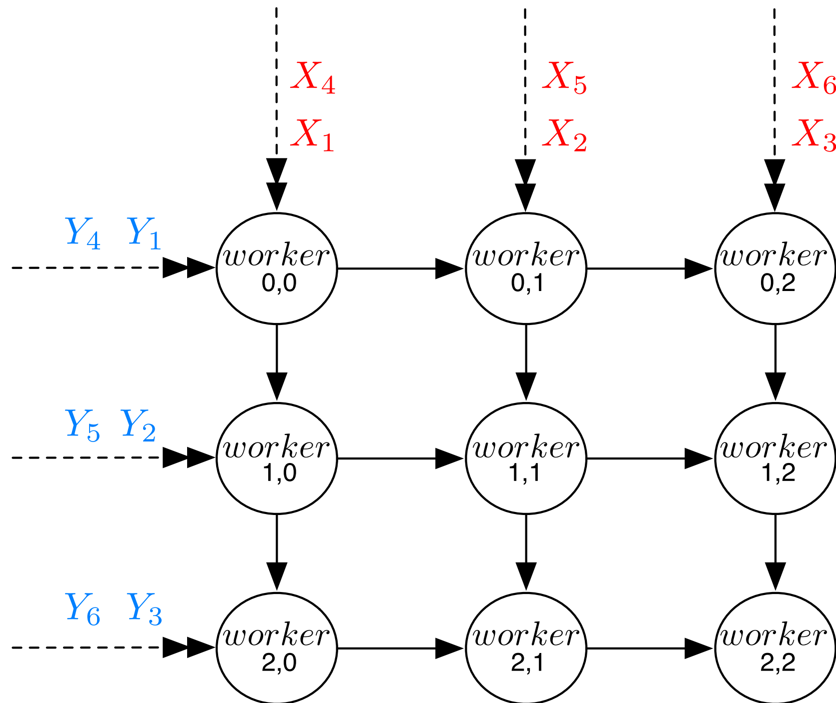


Figure 4.9: Round-robin distribution of elements

rect (cf. section 4.1). In order to prove it, we set up the following reasoning. Take in consideration one window of the stream R: its elements will be parti-

tioned over the columns of the workers matrix, and replicated over the rows. On the contrary, tuples in a S-window will be partitioned over the rows and replicated over the columns. In this way, each pair of tuple $(x \in R, y \in S)$ will be received by only one workers in the matrix and all the pairs of tuple will be processed eventually.
The only difference with the previous version is that elements of the same stream received by a given worker are not contiguous. This strategy will lead to have out of order outputs, but the correctness is still guaranteed. Worker's sequential algorithm ensures that every join computation is executed thanks to the check of the timestamps at step **2**.

4.5.3 Possible mappings of input data onto workers

Let us now discuss about some possible implementation schemes of our last proposed parallel paradigm. We suppose to have three kinds of entities: *emitter*, *collector* and *worker*. There will be more than one worker. Each worker will perform the computations that belong to a set of VPs (i.e. a set of join computation). We want to focus on the mapping of the input data onto the workers, so we will suppose to have N real processors executing our matrix of infinite virtual processors.
We can imagine, in general, a rectangular matrix of workers. The two streams elements are replicated and/or partitioned over rows and/or columns of the matrix. In particular, we can consider two limit cases: square matrix of workers (*block based mapping*), one row/column of workers (*row/column based mapping*).

- **Block based mapping**. We have a square matrix of workers as shown in figure 4.10.
  This solution performs partitioning and replication on both streams. The advantage of this mapping is that both the stream windows are evenly partitioned over the same number of workers, therefore this configuration can be used when the two input rates are similar. The total number of workers is a square number, therefore we could be forced to

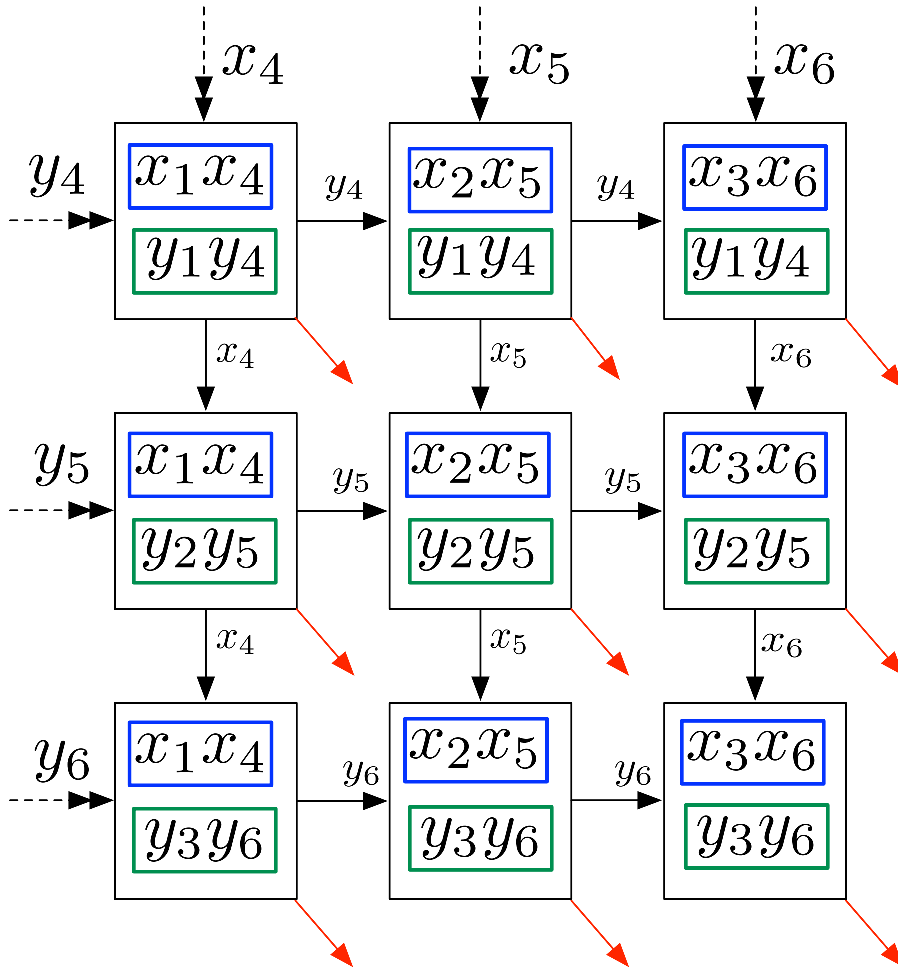


Figure 4.10: Block based mapping

under-utilize the available processing units.
The number of communication channels of this solution is $O(N)$: every worker has two input channels (i.e. from left and up) and three output channels (i.e. towards right, down and for the results) except for the workers on the sides of the matrix that have two or one output channels; the emitter has $2\sqrt{N}$ output channels; the collector has N input channels.

- **Row based mapping**. We have only one row of workers as shown in

figure 4.11.

In this configuration we perform total replication of the current win-

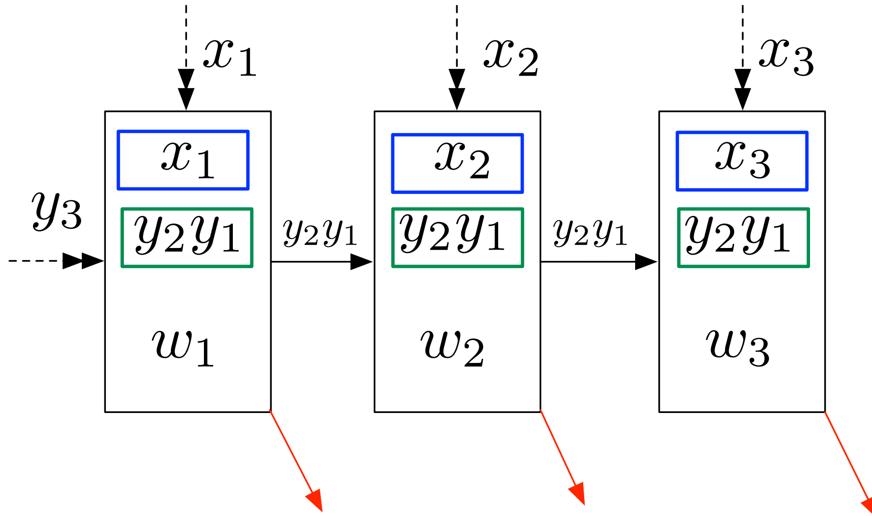


Figure 4.11: Row based mapping

dow of stream S on every worker, while the current window of R is partitioned among all the workers.

This mapping may be very useful when the stream S is characterized by a bigger inter-arrival time with respect to the one of R. Under these hypothesis the number of elements received from S in every window will be smaller than the one received from R. It is more convenient to replicate the elements of the slower streams (less elements in the current window) and to partition the tuples that comes from the faster stream.

Removal of elements from a window stream happens in the workers only at the arrival of an element from the other stream: this may cause the workers to keep in memory many expired items.

The number of communication channels of this solution is $O(N)$: every worker has two input channels (i.e. from left and up) and two output channels (i.e. towards right and for the results) except for the last worker that has only one output channel; the emitter has $N+1$ output

channels; the collector has N input channels.

4.6 Implementation

4.6.1 Assumptions

In order to implement our application we made some important assumptions. The solution has been designed to target shared memory architectures. We supposed that the inter-arrival times of the input streams are unknown to *parJoin*. The two streams, named R and S, are composed by fixed length *time-based* windows. We call T_R and T_S the lengths of the R window and the S window respectively.
Every element x in each stream is marked with a *time-stamp* t_x that determines its position in the stream. The time-stamp is put on the tuples by the generator of the stream, that is an external module to *parJoin*.

4.6.2 Tuples attributes and join predicate

We aimed at comparing our application with already existing solutions, so we decided to use the same tuples of CellJoin [GBY09] and *handshake join* [TM11] (cf. section 4.4).

- Tuples of stream R have attributes: `attr1`:int, `attr2`:float, `attr3`:char[20].

- Tuples of stream S have attributes: `attr1`:int, `attr2`:float, `attr3`:double, `attr4`:bool.

Both type of tuples have also a `timestamp` attribute used to check when they are expired.
The tuples are joined using the following predicate:
`tupleS.attr1` $\in$ `(tupleR.attr1 - 10, tupleR.attr1 + 10)`
`AND tupleS.attr2` $\in$ `(tupleR.attr2 - 10, tupleR.attr2 + 10)`
The two attributes utilized by the join predicate are generated with a uniform

random distribution in the interval $[1, 10000]$. Therefore the *hit-rate* becomes $p = \frac{20}{10000} \cdot \frac{20}{10000} = \frac{1}{250000} = 4 \cdot 10^{-6}$

4.6.3 Tuples memory organization

As pointed out in [GBY09], there are two basic types of memory organizations for storing tuples in memory, namely *row-oriented* (tuple oriented) and *column oriented* (attribute oriented).

In the first approach, different tuples are stored within contiguous regions

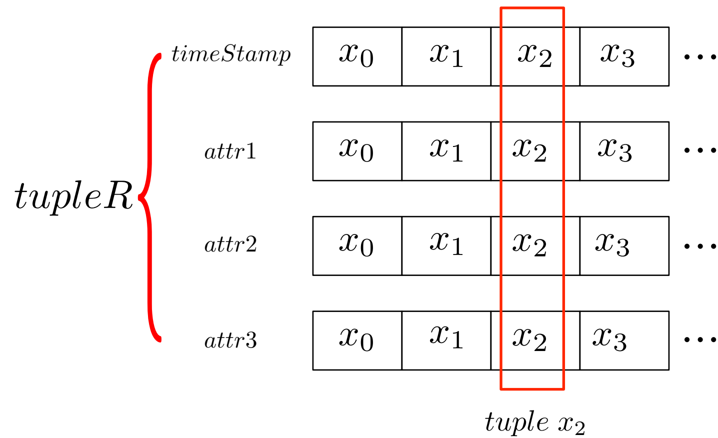


Figure 4.12: Column-oriented memory organization for tupleR

of memory. On the opposite side, column oriented organization stores the attributes contiguously as it can be seen in figure 4.12 for tupleR. This is more convenient for a memory-read intensive application like the stream join: we can exploit both spatial and temporal locality by prefetching the attributes in lower levels of the memory hierarchy and by not deallocating them until the corresponding tuples expire. Note that the stream join algorithm needs to pair only few attributes of tuples, in our case two.

The row-oriented solution is commonly used by traditional DBMS, whereas column-oriented organization is exploited by read-optimized databases [SAB+05]. Like the Cell-Join and the Handshake-Join, we utilized the column-oriented organization for our software.

In figure 4.13 it is shown how we adapted the column-oriented memory

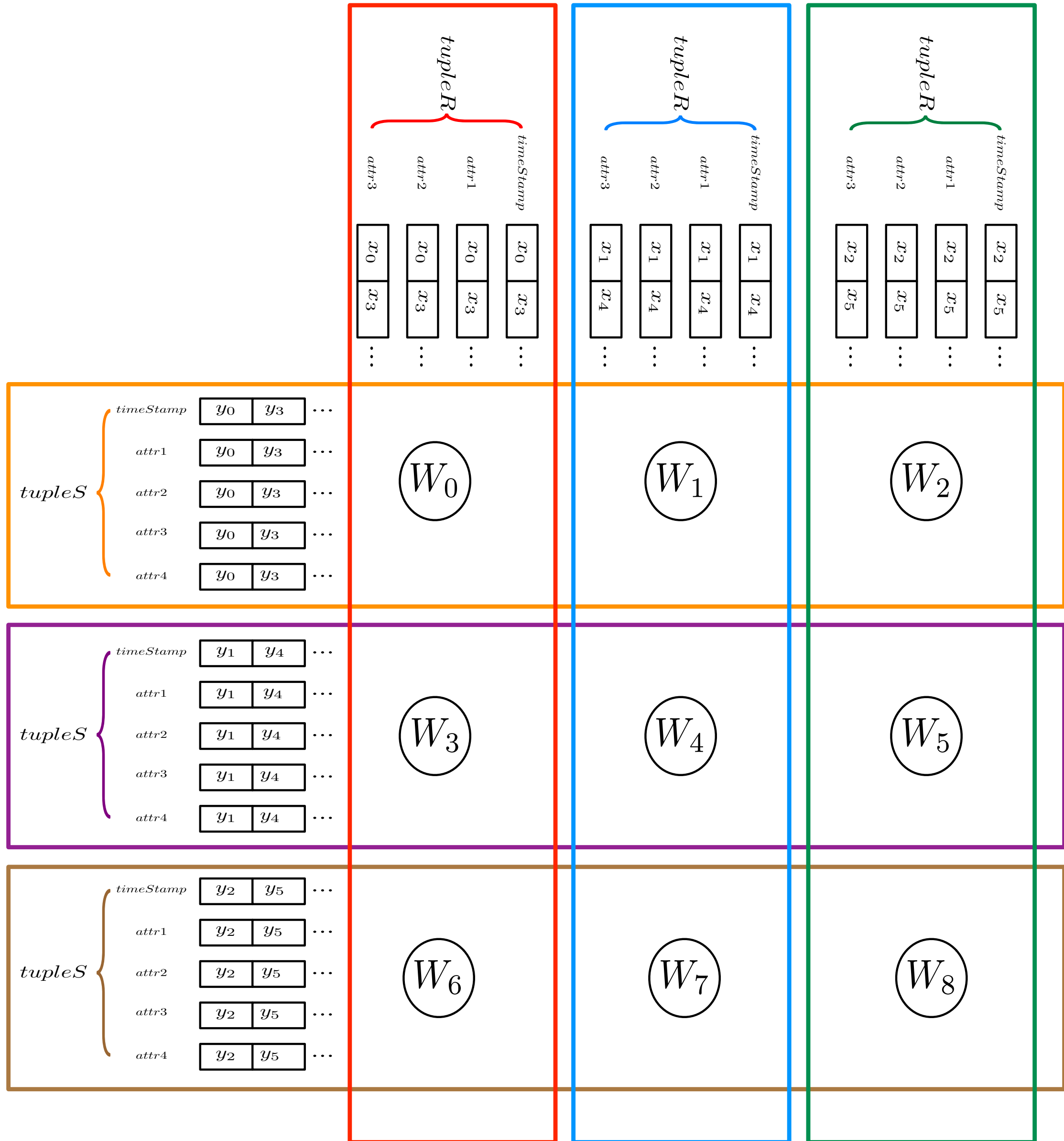


Figure 4.13: *parJoin* memory organization

organization to our data partitioning and replication scheme. In the example we can see a matrix of nine workers (three columns and three rows); generally speaking we have a number of column-oriented structures for stream R equal to the number of columns in the matrix and as many column-oriented struc-

tures for stream S as the number of rows. The partitioning is exploited by dividing the single column-oriented structure in many smaller ones, while the replication can be seen in the fact that one small column-oriented structure is shared by the workers on the same row (or column).

4.6.4 Inter-thread communications: FastFlow queues

FastFlow (FF) [ADKT13] is a C++ framework for structured parallel programming targeting both shared memory and distributed memory architectures. It is composed by several layers which abstract the underlying architecture. The abstraction provided by these layers is twofold: to simplify the programming process offering high-level constructs for data parallel and stream parallel skeletons creation and, at the same time, to give the possibility to fine tune the applications using the mechanisms provided by the lower layers.
At the base level FF offers single producer/single consumer (SPSC) queues [ADK+12] which can be used as communication channels between software threads. These queues are characterized by the absence of locking mechanisms. SPSC queues can be classified in two main families: bounded and unbounded. Bounded SPSC queues, typically implemented using a circular buffer, are used to limit memory usage and avoid the overhead of dynamic memory allocation. Unbounded queues are mostly preferred to avoid deadlock issues without introducing heavy communication protocols in the case of complex streaming networks, i.e. graph with multiple nested cycles.
We exploited FastFlow SPSC bounded queues to implement the communication channels needed by our application. The communication channels were used to connect the emitter and the collector with the workers and the workers with each other.
The elements passed in the queues are just pointers to the real data that are stored in shared memory. The tuples, or their attributes are not copied from a memory location to another.

4.7 Tests

In this section we present the tests we performed and the results obtained. Our experiments were made to study the performance of our solution and to compare it with *handshake join* [TM11].

We utilized two types of shared memory architectures for our experimental evaluations. The first type is an Intel Xeon Processor E5-2650 at 2 GHz. The number of physical cores is 16, but exploiting the available Hyper-Threading they become 32 virtual cores. We used two machines (`pianosa, pianosau`) of this type linked together with InfiniBand: one to generate the streams, the other to implement the module performing the stream join.
The second type of architecture utilized during the tests is an AMD Opteron Processor 6176 at 2.30GHz with 24 cores. We used one machine of this type: `titanic`.

We have performed different types of experiments. Initially, we studied only the behaviour of our own solution in order to evaluate its scalability and to understand if it can reach the ideal output rate. Then we have compared our solution with *handshake join*, utilizing the source code available at [TM]. The comparisons were done for symmetric input rates utilizing *parJoin* with block mapping and for asymmetric input rates utilizing *parJoin* with column mapping (cf. section 4.5.3).

For all the experiments that we will present in this section we computed also the ideal output rate of the stream join module. In order to compute its value we divided the number of generated outputs for the generation time: $\frac{\Omega_{t_0,t_1}}{(t_1 - t_0)}$.
The number of outputs generated in a time interval $[t_0, t_1]$ can also be foreseen, before running the experiment, utilizing formula (4.1) already presented in section 4.3.

$$\Omega_{t_0,t_1} = \lambda_R \lambda_S (t_1 - t_0) \cdot (|W_S| + |W_R|) \times p$$

This formula holds when the two windows are already full of elements at time t_0. If we want to evaluate the total number of outputs generated from the beginning of the computation, we can ignore all the comparisons done in the filling phase state, when the two windows are not completely filled yet. The formula becomes:

$$\Omega_{t_0,t_1} = \lambda_R \lambda_S (t_1 - t_0) \cdot (|W_S| + |W_R|) \times p - \lambda_R |W_R| \cdot \lambda_S |W_S| \times p$$

During the experiments we measured the actual *hit-rate* that resulted to be $p = 3.6 \cdot 10^{-6}$. It is very similar, but not equal, to the theoretical one: $p = 4 \cdot 10^{-6}$. Using this *hit-rate* in the formula to predict the total number of generated outputs, we obtained a percent error $\leq 0.34\%$ as shown in table 4.1.

$\|W_S\|, \|W_R\|$	λ_R	λ_S	Generation time	Predicted	Actual	Percent error
300	400	400	900	259200	259595	0.15
300	600	600	900	583200	583775	0.09
300	800	800	900	1036800	1037995	0.11
300	1000	1000	1200	2268000	2275781	0.34
300	2000	2000	900	6480000	6492230	0.18
300	800	1200	900	1555200	1558455	0.20
300	800	1600	900	2073600	2077731	0.19
300	800	2000	900	2592000	2598837	0.26
300	200	800	900	259200	259799	0.23
600	1000	1000	1200	3888000	3899042	0.28
600	1500	1500	1200	8748000	8766142	0.20
600	2000	2000	1200	15552000	15578172	0.16

Table 4.1: Percent errors for the predicted numbers of outputs generated

Therefore we can say that our formula is a very good approximation for the number of generated outputs and it is possible to predict the ideal output rate before running the tests and collecting the results. This can be useful during the computation to understand if the parallel module is still a bottleneck and can be further parallelized adding other workers or if it has already reached the maximum bandwidth.

4.7.1 parJoin

Our first set of experiments focus just on *parJoin* in order to evaluate its behaviour with different input rates. These tests were run utilizing two machines (`pianosa, pianosau`). On `pianosa` we executed the generator sending the tuples of the two streams, while on `pianosau` we executed the parallel module implementing the stream join. The two machines are connected with InfiniBand and the communications are implemented using standard TCP sockets. This is as close as possible to a real stream join application in which the streams are received from external sources.
We created two set of tuples, one for each stream, and utilized the same two sets for all the computations performed. This means that all the different experiments produce exactly the same results because the tuples attributes, the timestamps and the window sizes utilized are always the same. In these tests the generator does not send the tuples according to their timestamps, but it generates the stream with different input rates. This is done in order to study the changes in the parallel module behaviour and to study its scalability.
The timestamps of the tuples are generated in order to have an average input rate of 1000 tuples per second. The window sizes are fixed to 100 seconds, therefore on average, in steady state, there are 100000 tuples in each stream window.
The number of tuples sent for each stream is 240000, therefore the generation time is 240 seconds if the tuples are sent according to their timestamps. Actually, in this set of experiments, as already said, we send the tuples with input rates different from the one dictated by the timestamps.
By sending the same tuples with increasing input rates, we can observe how our solution performance changes and see if it reaches the ideal output rate with a certain parallel degree or if it remains a bottleneck. We can also study the scalability of the module in presence of different input rates.

From the following figures we can see that our parallel module performs very well utilizing input rates in the range $[4000 tuples/s, 8000 tuples/s]$. We utilized very high input rates in order to stress the application and to exploit

bigger parallel degrees.
The scalability obtained in all the cases is almost ideal as shown in figure 4.17. Furthermore, with lower input rates we reach the ideal output rate utilizing 16 workers, while in the remaining ones the module is still a bottleneck and could be further parallelized. As expected, with higher input rates we need a greater parallel degree in order to reach the ideal output rate.

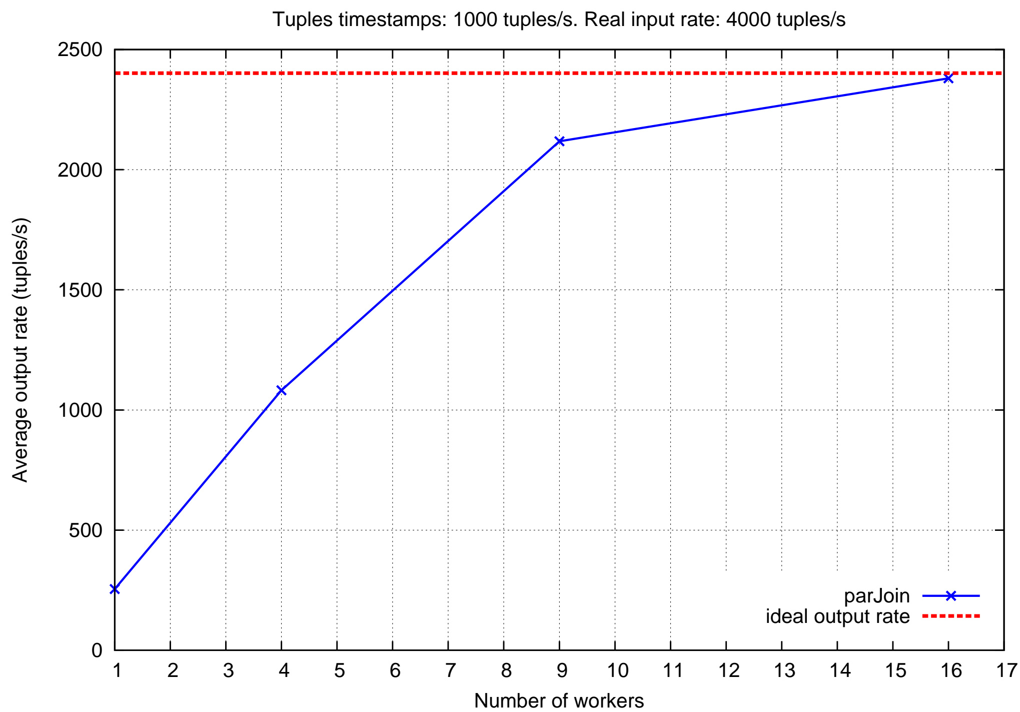
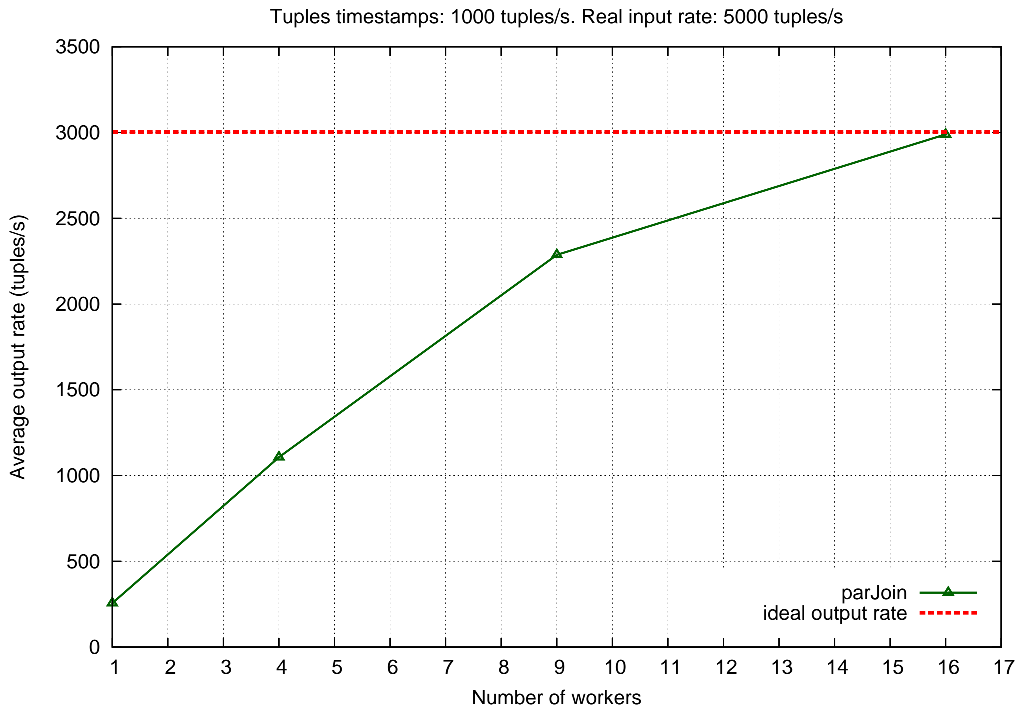
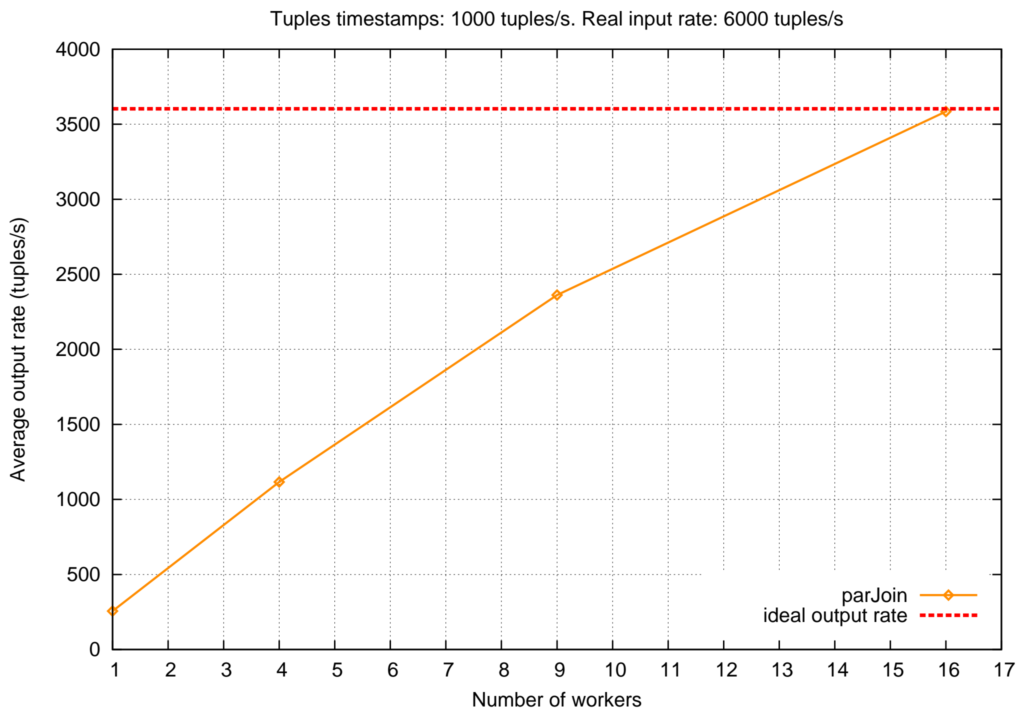


Figure 4.14: parJoin with input rates of 4000 tuples/s

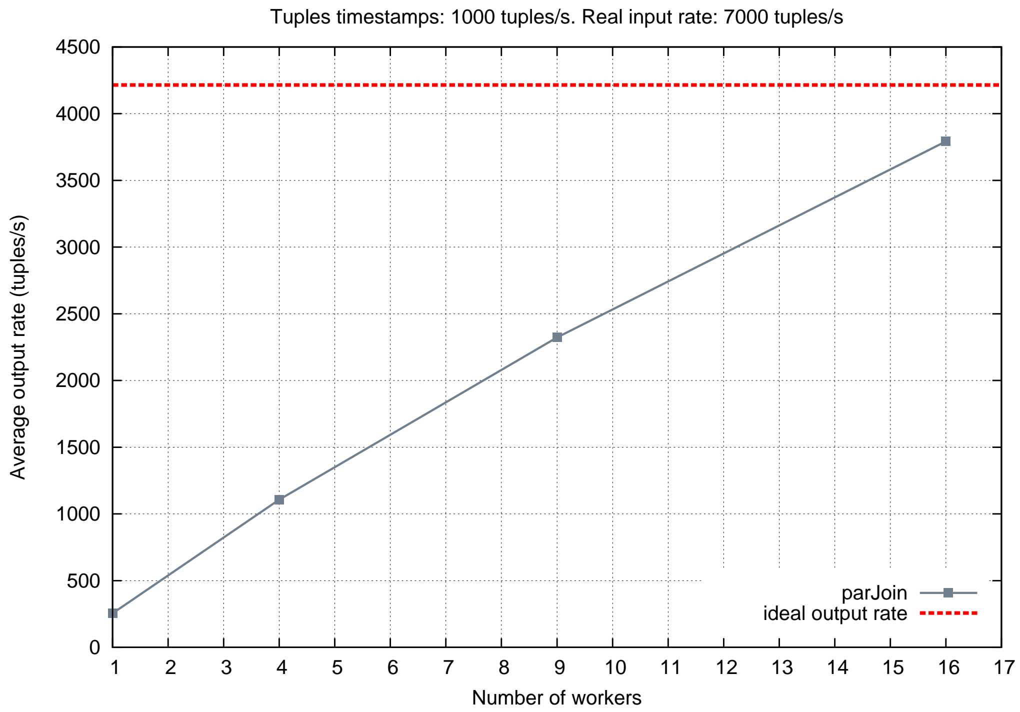


(a) parJoin with input rates of 5000 tuples/s

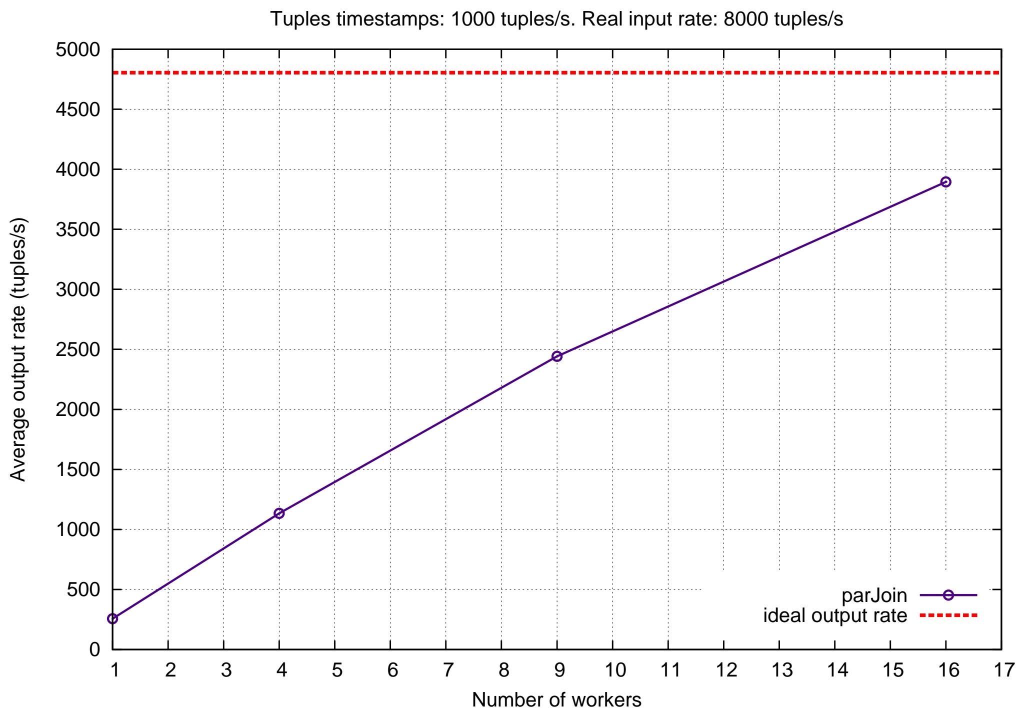


(b) parJoin with input rates of 6000 tuples/s

Figure 4.15



(a) parJoin with input rates of 7000 tuples/s



(b) parJoin with input rates of 8000 tuples/s

Figure 4.16

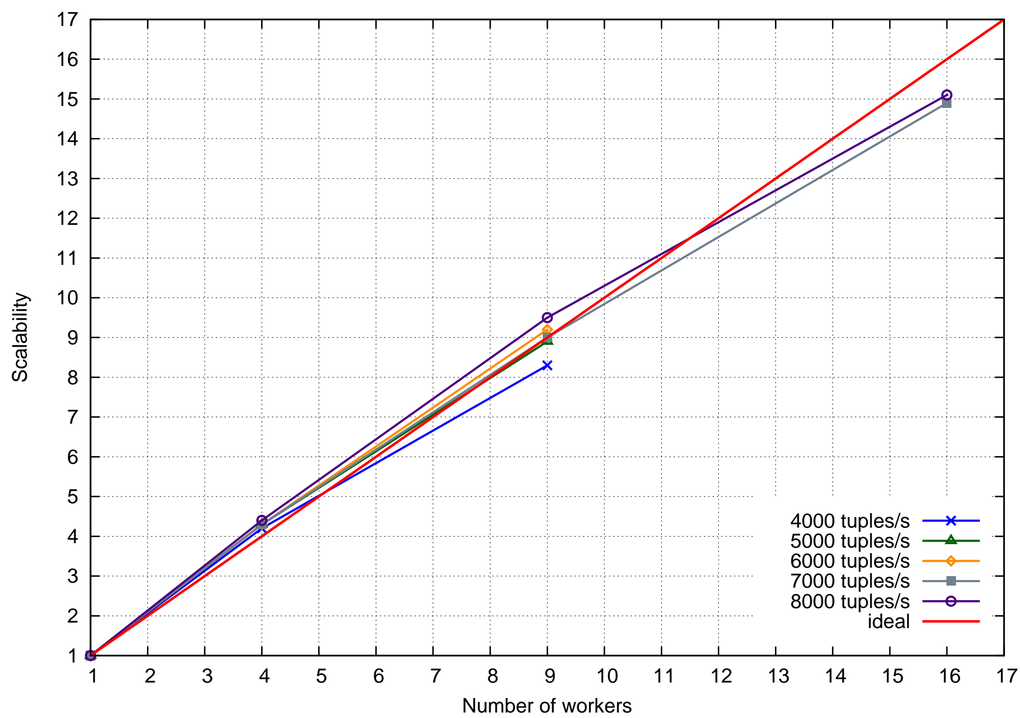


Figure 4.17: parJoin scalability with input rates ∈ [4000 tuples/s, 8000 tuples/s]

4.7.2 Comparison with literature using symmetric input rates

The second set of experiments focus on comparing our *parJoin* with the existing *handshake join*, the best published result for stream join performed in parallel. In order to make a fair comparison with *handshake join* we needed to have the generator of the streams on the same machine. Indeed *handshake join* utilizes additional threads for sending the tuples instead of an external process. The tuples are retrieved directly from memory instead of being sent on TCP sockets as done in the previous set of experiments. These experiments were performed utilizing one machine (`titanic`).
In these test we utilized the code of *handshake join* to create the set of tuples, stored them in files and given them in input to our solution. In this way we have a significant comparison because the tuples attributes, the timestamps and the window sizes utilized are always the same for both the applications. Furthermore the input rate utilized is the same for *handshake join* and our solution.
The generation of the tuples timestamps is done (by *handshake join*) with a continuous uniform distribution. Timestamps start from zero and are increased conveniently in order to obtain the desired average input rate.
The timestamps of the tuples are generated in order to have different average input rates for each experiment. The window sizes are instead always fixed to 300 seconds.
In these experiments the tuples are always sent according to their timestamps. We wanted to study the differences between *handshake join* and our solution to see if they both reach the ideal output rate with a certain parallel degree.

From figures 4.18, 4.19a and 4.19b we can see that our parallel module performs better than *handshake join* utilizing input rates in the range [$800 tuples/s$, $1200 tuples/s$] for a 5 minutes window. Our module tends to scale very well in all these cases.
As expected, with greater input rates our solution needs a greater parallel degree in order to reach the ideal output rate. Specifically, with 800 tuples/s

we reach the ideal output rate utilizing 9 workers, while in the remaining cases our module is still a bottleneck and could be further parallelized with more than 16 workers. Nevertheless, in all these cases we never stabilize on a non-optimal output rate.
On the other hand, *handshake join* does not increase substantially its output rate with more than 4 workers. With small parallel degrees, *handshake join* performs better, because it utilizes some optimization techniques such as batching that can be useful to improve the performance also for sequential computations. The lack of scalability may be due to the communication protocol based on acknowledgement messages between workers.

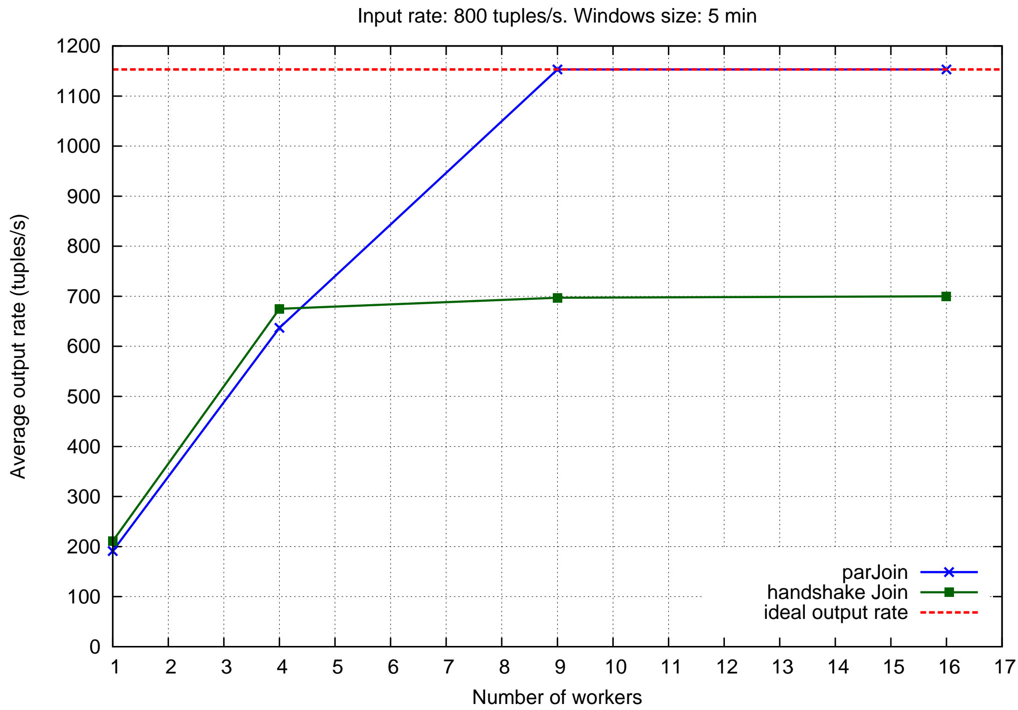
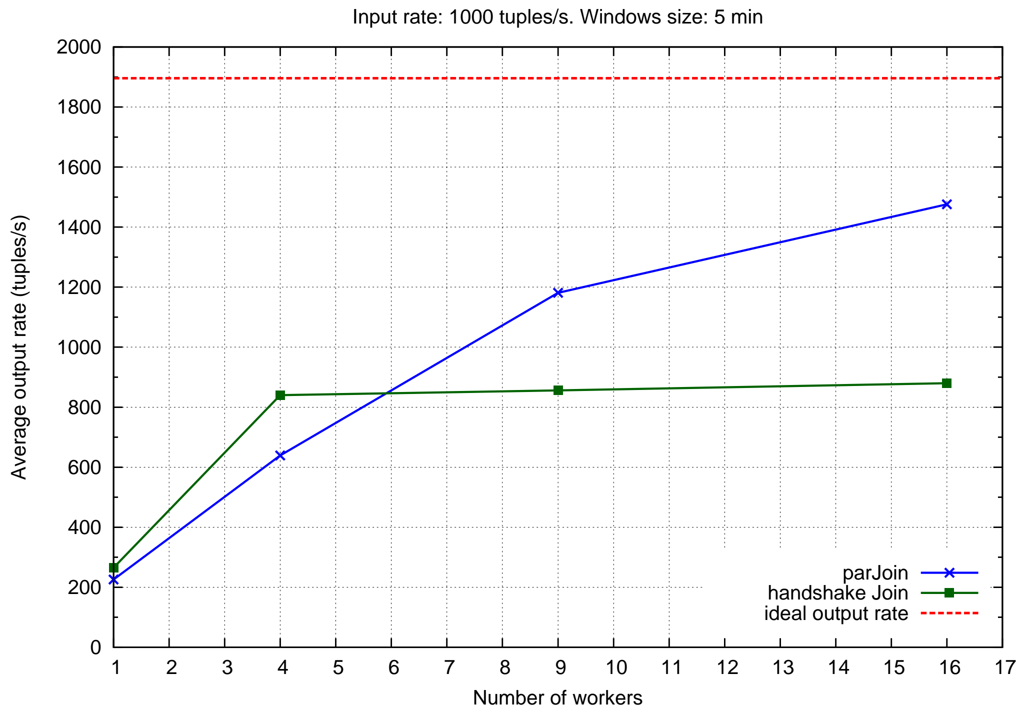
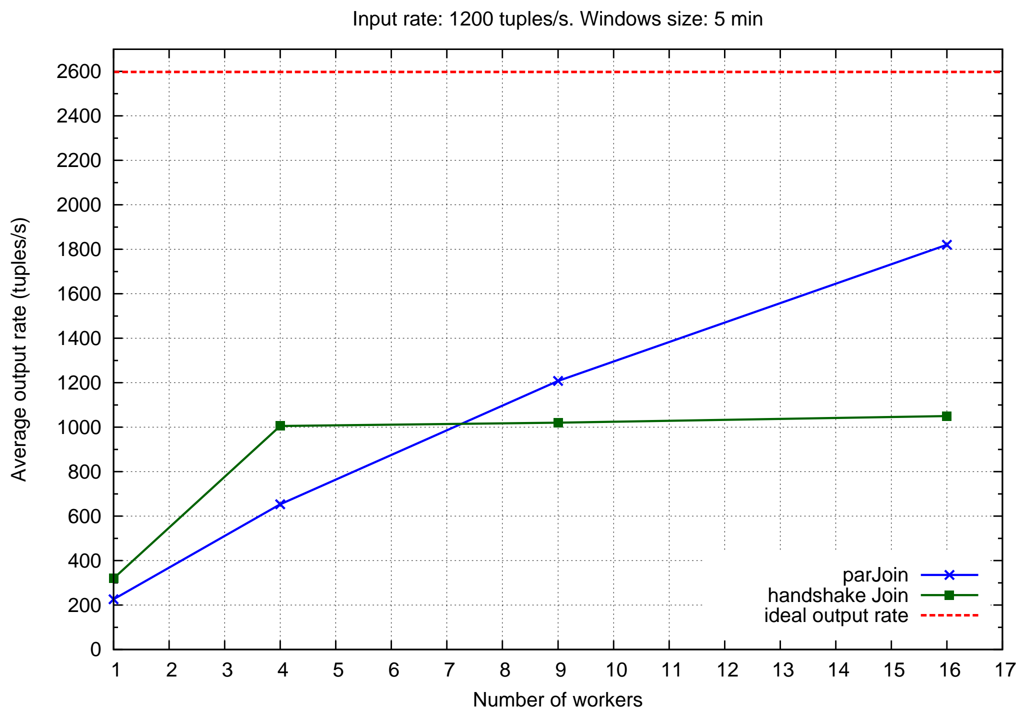


Figure 4.18: Comparison with input rates of 800 tuples/s



(a) Comparison with input rates of 1000 tuples/s



(b) Comparison with input rates of 1200 tuples/s

Figure 4.19

Figures 4.20 and 4.21 show comparisons done utilizing very high input rates: $4000 tuples/s$ and $5000 tuples/s$ with 5 minutes windows. We want to point out again that in these experiments the tuples are sent according to their timestamps. Increasing the input rates, also the number of comparisons to be done and the number of results will increase substantially. In order to fit the ideal input rates in the plots we utilized a logarithmic scale.
With these input rates both the solutions remain bottlenecks despite increasing the parallel degrees. The output rates reached are not very significant because they are very far from the ideal ones and the applications are not able to keep pace with the arrivals from the streams.
Handshake join performs better than our parallel module but does not increase its output rate with more than 9 workers and remains far from the ideal output rates. The shapes of the output rate functions suggest that adding more workers would not be useful.

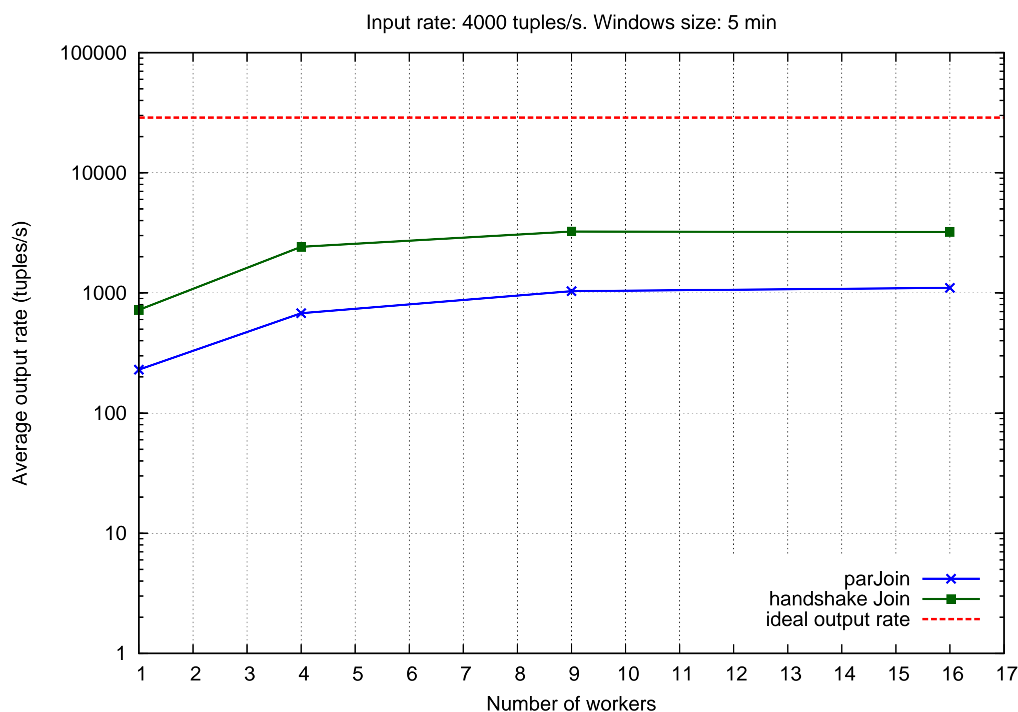


Figure 4.20: Comparison with input rates of 4000 tuples/s

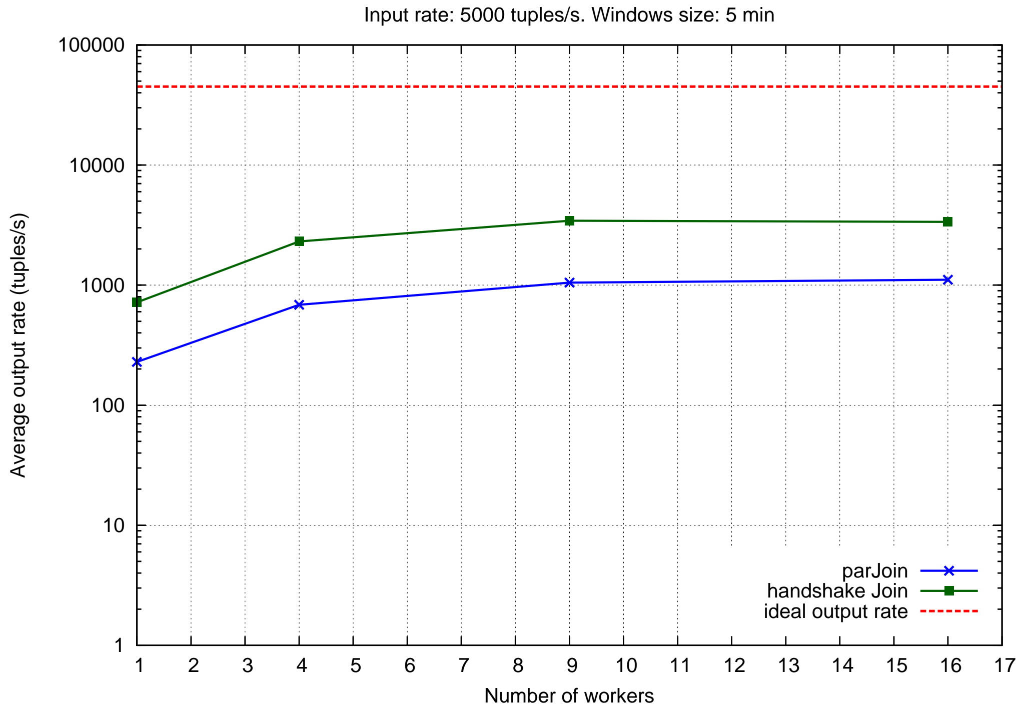


Figure 4.21: Comparison with input rates of 5000 tuples/s

4.7.3 Comparison with literature using asymmetric input rates

The last set of experiments still focus on comparing our solution with *handshake join*, but utilizes the column mapping of the windows data onto the workers. We wanted to test this mapping in presence of asymmetric input rates. Furthermore, with this configuration we can exploit more parallel degrees and we are not fixed to utilize only square numbers of workers.
These experiments are similar to the previous ones: they are run utilizing one machine (`titanic`) and the generator of the streams is on the same machine of the parallel module.
The tuples are created by *handshake join*, stored in files and then given in input to our application.
In these experiments the timestamps of the tuples are generated in order to have different average input rate for each stream. The input rate of stream

R is always fixed to 800 tuples/s while input rate of stream S is different for each experiment. The window sizes are always fixed to 300 seconds in all the tests.
Like the previous set of experiments, the tuples are sent according to their timestamps.

From the following figures we can see that our parallel module performs better than *handshake join* utilizing input rates for stream S in the range $[1200 tuples/s, 2000 tuples/s]$ and input rate of 800 tuples/s for stream R.
In all the cases *handshake join* does not increase substantially its output rate utilizing more than 4 workers.
Our module tends to scale very well with an input rate of stream S equal to 1200 tuples/s and 1600 tuples/s and also reaches the ideal output rate with 9 or 16 workers.
When the input rate of stream S is very high (2000 tuples/s) our module still performs better than *handshake join* but doesn't scale ideally. Nevertheless, our solution does not stabilize on a non-optimal output rate and continues to increase utilizing more workers.

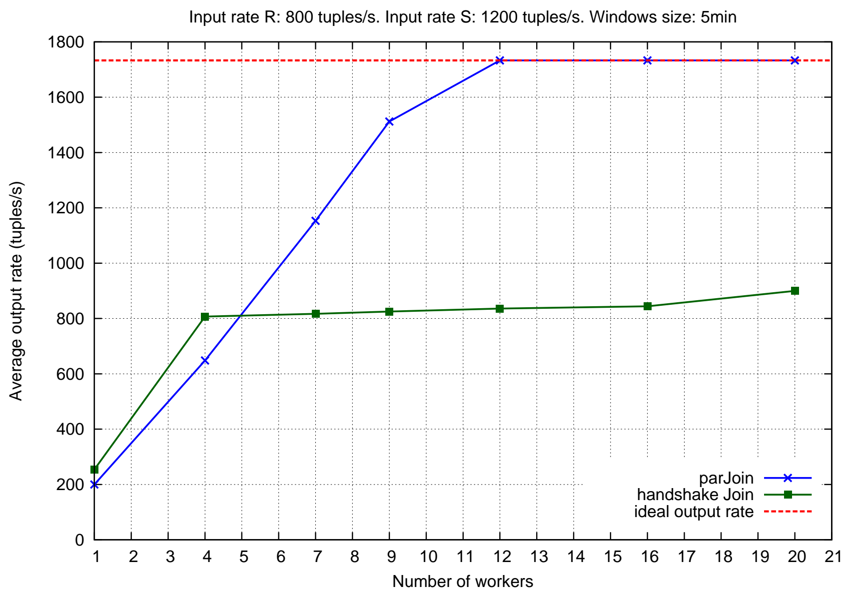
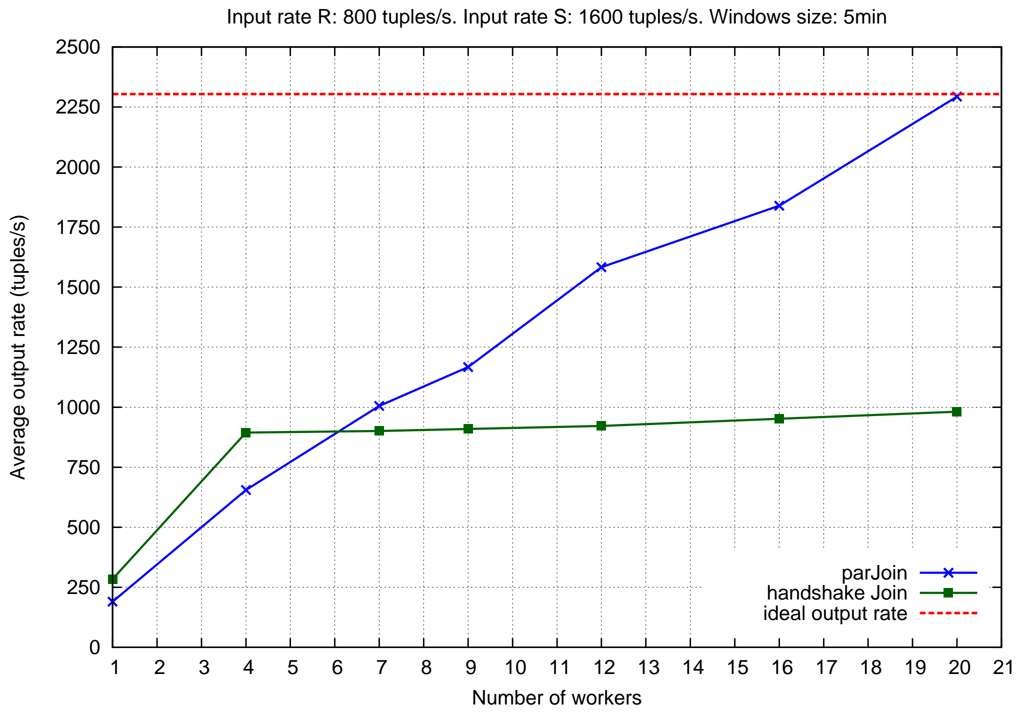
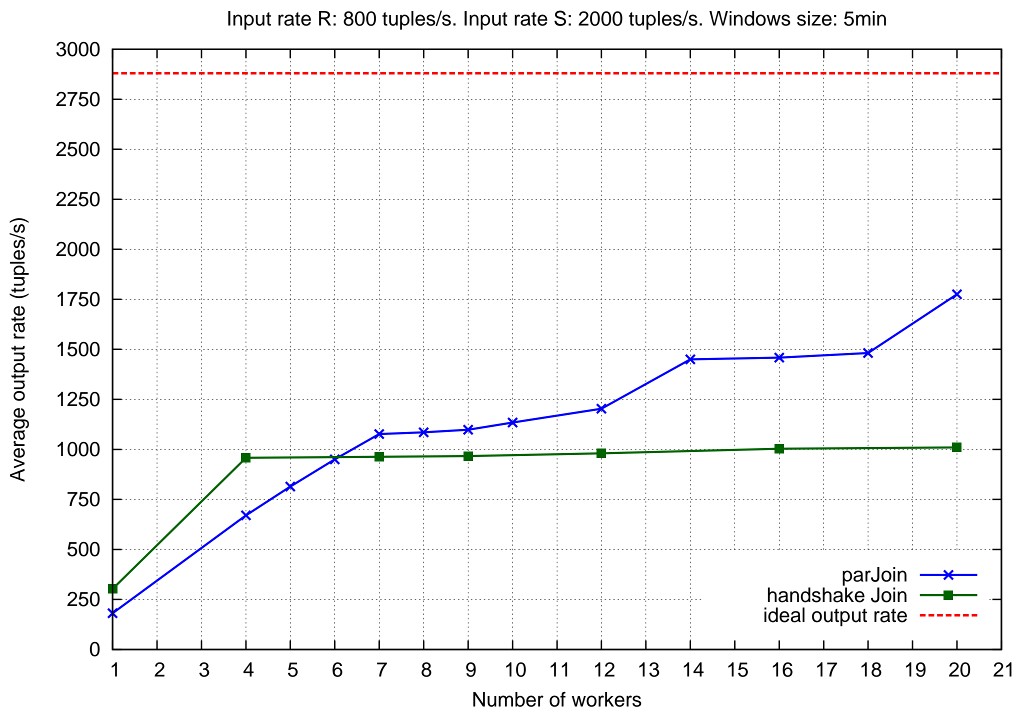


Figure 4.22: Comparison with input rates of 800 tuples/s and 1200 tuples/s



(a) Comparison with input rates of 800 tuples/s and 1600 tuples/s



(b) Comparison with input rates of 800 tuples/s and 2000 tuples/s

Figure 4.23

Chapter 5

Conclusions

In this chapter we summarize the thesis work and the results obtained. As final remarks, we offer possible suggestions to further develop the research on parallel DaSP.

Thesis summary

During the thesis we studied large part of the existing literature about Data Stream Processing, in particular for the parallel case. Starting from the standard model for DaSP we developed the concept of parallel DaSP module and discussed the main issues about the distribution of input elements to macro-workers. We were able to formalize different distribution techniques for DaSP modules operating on one ore more streams. The results are significant also because we showed that, using proper distribution techniques, many sliding windows computations can be transformed in tumbling windows one.
We showed how to adapt existing parallel paradigms for tumbling windows computations. This enables to exploit the results obtained in structured parallel programming about performance predictability for this kind of computations.
We analysed a significant streaming problem called *Stream Join* that is time-based and utilizes sliding windows. Even if we could not give a formal method to parallelize this kind of computations, we gave a specific solution to the

problem that is strongly based on an existing parallel paradigm (*stencil*). We implemented this solution and compared its performance with an existing parallel solution using the same test-bed. The results obtained where quite good, both in terms of scalability and comparing it to the solution in literature. These tests showed how structured parallel programming can be adapted to DaSP with very good results:

scalability results Part of the tests were aimed to show that our solution to the stream join problem scales very well. We achieved scalability really close to the ideal one for the tests performed.

comparison with literature Other tests were done to compare our solution with the fastest one in literature. Fixing the window size, for sustainable input-rates, we performed better than the existing solution often reaching the ideal output rate.

Future works

Here we propose some future works to advance our research on structured parallel programming for DaSP.

Tumbling windows In chapter 3, we showed how to adapt some skeletons to DaSP on tumbling windows, but we did not implemented any of the proposed solutions. It would be interesting to solve some real problems with the techniques proposed, and compare the theoretical results with the experimental ones. A possible problem to address could be the packet inspection in a router that in some way remembers the Basic Counting.

parJoin Our application can be further improved. In the sequential algorithm we can add many optimizations like vector (or *SIMD*) instructions. From the point of view of the parallel implementation further investigations can be done. For example, individuating a good batch size for the input streams can improve the performance. Other tests with different windows sizes and input rates can be performed for any

configuration of parJoin (matrix, columns and rows). A new implementation, based on a rectangular matrix of workers, can be studied and realized.

Sliding windows We still lack a parallel programming model for the streaming problems in this class.

Bibliography

[AAB+05] 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.

[ABB+04] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R Motwani, U. Srivastava, and J Widom. Stream: The stanford data stream management system. Technical report, 2004.

[ACC+03] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, J. h. Hwang, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, and S. Zdonik. Aurora: A data stream management system. In *In ACM SIGMOD Conference*, page 666, 2003.

[ADK+12] 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 of *LNCS*, pages 662–673, Rhodes Island, Greece, August 2012. Springer.

[ADKT13] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Fastflow: high-level and efficient streaming on multi-core. In Sabri Pllana and Fatos Xhafa, editors, *Programming Multi-core and Many-core Computing Systems*, Parallel and Distributed Computing, chapter 13. Wiley, January 2013.

[BBD+02] 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.

[BDM07] Brian Babcock, Mayur Datar, and Rajeev Motwani. Load shedding in data stream systems. In Charu C. Aggarwal, editor, *Data Streams - Models and Algorithms*, volume 31 of *Advances in Database Systems*, pages 127–147. Springer, 2007.

[BT11] Cagri Balkesen and Nesime 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.

[CCD+03] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[CGRS01] Kaushik Chakrabarti, Minos N. Garofalakis, Rajeev Rastogi, and Kyuseok Shim. Approximate query processing using wavelets. *VLDB J.*, 10(2-3):199–223, 2001.

[DM07] Mayur Datar and Rajeev Motwani. The sliding-window computation model and results. In CharuC. Aggarwal, editor, *Data Streams*, volume 31 of *Advances in Database Systems*, pages 149–167. Springer US, 2007.

[GBY09] 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.

[Gol06] Lukasz Golab. Sliding window query processing over data streams. Phd, University of Waterloo, 2006.

[Gsc06] Michael Gschwind. Chip multiprocessing and the cell broadband engine. In *Proceedings of the 3rd conference on Computing frontiers*, CF '06, pages 1–8, New York, NY, USA, 2006. ACM.

[Gsc07] Michael Gschwind. The cell broadband engine: exploiting multiple levels of parallelism in a chip multiprocessor. *Int. J. Parallel Program.*, 35(3):233–262, June 2007.

[Gul12] Vincenzo Massimiliano Gulisano. *StreamCloud: An Elastic Parallel-Distributed Stream Processing Engine*. PhD thesis, Universidad Politécnica de Madrid, December 2012.

[IP99] Yannis E. Ioannidis and Viswanath Poosala. Histogram-based approximation of set-valued query-answers. In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *VLDB*, pages 174–185. Morgan Kaufmann, 1999.

[KNV03] Jaewoo Kang, Jeffrey F. Naughton, and Stratis D. Viglas. Evaluating window joins over unbounded streams. In *ICDE*, pages 341–352, 2003.

[LMT+05] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Rec.*, 34(1):39–44, March 2005.

[NDM+01] Jeffrey Naughton, David Dewitt, David Maier, Ashraf Aboulnaga, Jianjun Chen, Leonidas Galanis, Jaewoo Kang, Rajasekar Krishnamurthy, Qiong Luo, Naveen Prakash, Ravishankar Ramamurthy, Jayavel Shanmugasundaram, Feng Tian, Kristin Tufte, and Stratis Viglas. The niagara internet query system. *IEEE Data Engineering Bulletin*, 24:27–33, 2001.

[oO] University of Oldenburg. Odysseus: An open source java based framework for data stream management systems. `http://odysseus.informatik.uni-oldenburg.de:`

8090/display/ODYSSEUS/Odysseus+Home [Online; accessed 15-Nov-2012].

[RD09] Florin Rusu and Alin Dobra. Sketching sampled data streams. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, ICDE '09, pages 381–392, Washington, DC, USA, 2009. IEEE Computer Society.

[SAB+05] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: a column-oriented dbms. In *Proceedings of the 31st international conference on Very large data bases*, VLDB '05, pages 553–564. VLDB Endowment, 2005.

[TM] Jens Teubner and Rene Mueller. How soccer players would do stream joins - source code. http://people.inf.ethz.ch/jteubner/publications/soccer-players/handshake-join-0.1.tar.gz [Online; accessed 15-Nov-2012].

[TM11] 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.

[UoWM] Portland State University University of Wisconsin-Madison. Niagarast: A research data stream management system. http://datalab.cs.pdx.edu/niagara/ [Online; accessed 15-Nov-2012].

[Van09] Marco Vanneschi. *Architettura degli elaboratori.* Pisa University Press, 2009.

[Van12] Marco Vanneschi. Course notes of high performance systems and enabling platforms. Master Program in Computer Science and Networking, 2012.