CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
*Concurrency Computat.: Pract. Exper.* 2015; **00**:1–20
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

# Continuous Skyline Queries on Multicore Architectures

Tiziano De Matteis, Salvatore Di Girolamo and Gabriele Mencagli*

*Department of Computer Science, University of Pisa, Largo B. Pontecorvo 3, I-56127, Pisa, Italy*

SUMMARY

The emergence of real-time decision-making applications in domains like high-frequency trading, emergency management and service level analysis in communication networks, has led to the definition of new classes of queries. *Skyline queries* are a notable example. Their results consist of all the tuples whose attribute vector is not dominated (in the Pareto sense) by one of any other tuple. Because of their popularity, skyline queries have been studied in terms of both sequential algorithms and parallel implementations for multiprocessors and clusters. Within the *Data Stream Processing* paradigm, traditional database queries on static relations have been revised in order to operate on continuous data streams. Most of the past papers propose sequential algorithms for continuous skyline queries, whereas there exist very few works targeting implementations on parallel machines. This paper contributes to fill this gap by proposing a parallel implementation for multicore architectures. We propose: *i)* a parallelization of the *eager* algorithm based on the notion of *Skyline Influence Time*, *ii)* optimizations of the reduce phase and load-balancing strategies to achieve near-optimal speedup, *iii)* a set of experiments with both synthetic benchmarks and a real dataset in order to show our implementation effectiveness. Copyright © 2015 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Data Stream Processing; Skyline Queries; Sliding Windows; Multicore Programming

## 1. INTRODUCTION

In recent years our ability to produce information has been growing steadily, driven by an ever increasing computing power, communication rates and hardware/software sensor diffusion. Data are often available in the form of continuous streams, and the development of automated tools to gather and analyze information "on the fly" in order to extract insights and detect patterns has become a valuable opportunity. Consequently, an increasing number of applications [1] (e.g., financial trading, environmental monitoring, and network intrusion detection) need continuous processing of data flowing from several sources, in order to obtain promptly and timely responses to complex queries.

In this paper we study parallel solutions for *skyline queries* over data streams [2]. They are a particular class of preference queries that compute the set of Pareto-optimal points from a given set. Fig. 1 shows a representation of the skyline. In this example the points represent hotels with the attribute *price* per night and *distance* from the center of the town. The result of the query is the set of the most interesting hotels that cannot be bettered in both price and distance by any other. In the literature the skyline set is also referred to as *Pareto frontier* [3] or *maximum vector* [4].

Skyline queries have received considerable attention in the database and information retrieval community, owing to their importance in multiple-criteria optimization. They have been studied in several papers that can be classified according to two main factors: *i)* whether the query is executed in a centralized or in a distributed/parallel fashion, *ii)* whether it operates on static datasets or on continuous data streams. All the four combinations are possible.

*Correspondence to: G. Mencagli, Email mencagli@di.unipi.it, Phone +39-050-2213132.

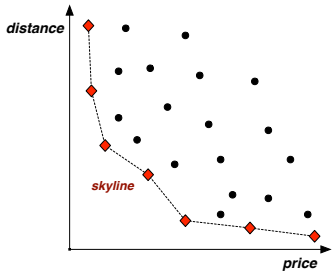



Figure 1. Example of a skyline query given a set of 2-dimensional points: red points are the ones in the skyline set, black points are dominated by at least one red point.

A large volume of papers have focused on sequential algorithms for computing the skyline on static datasets [2, 5, 6, 7, 8]. Most of these approaches present strategies to avoid scanning the whole dataset (e.g., using suitable index data structures). A summary will be provided in Sect. 2.3.

The problem of computing the skyline on data streams has additional difficulties. The skyline is a *blocking* operator that requires to scan the entire input before producing the first output [9]. Since the stream is possibly unbounded, the approach is to compute the skyline on a *sliding window* corresponding to the most recent points [10, 11, 12, 13, 14]. In this way the skyline comprises only the live data and the query produces a stream of skyline updates once new points arrive or expire.

Parallel implementations of continuous skyline queries have not been sufficiently investigated yet. The major contribution of this paper is to propose a parallel implementation targeting multicores. We use the *eager* algorithm proposed in [10] based on the concept of *Skyline Influence Time* (SIT). This algorithm is characterized by an intensive pruning activity, which allows the computation to save memory at the expense of a higher computational burden with respect to other algorithms. To the best of our knowledge, this is the first time that this algorithm has been parallelized and the performance evaluated on multicores. The other contributions are:

- a critical aspect for an efficient parallelization of the eager algorithm is to provide effective ways to deal with pruning, which may greatly impact on load balancing. To this end, we present some partitioning strategies to balance the workload and we assess their effectiveness both using synthetic distributions and in a real dataset;
- we design an *asynchronous reduce* to compute the global SIT of each point without blocking the threads of the implementation;
- we perform thorough experiments under different scenarios in terms of window length, arrival rate and we analyze our implementation by using data generated by a real-world application.

This work is the extended version of the conference paper published in [15], which covers only a small fraction of the results presented in this paper.

This paper is organized as follows. Sect. 2 provides a brief overview of skyline computations on static datasets and on continuous streams. Sect. 3 introduces the preliminary concepts and definitions of the skyline problem and describes the eager algorithm. Sect. 4 and 5 present our parallelization and the optimizations. Sect. 6 shows the experiments and Sect. 7 gives the conclusion of this work.

## 2. RELATED WORK

The skyline operator originates from the maximal vector problem in computational geometry [4] and it has been firstly introduced in the database context in [2]. With the emergence of data-intensive applications, skyline queries (and extensions like top-k queries) have obtained a renewed interest especially in application scenarios in which inputs are available as unbounded data streams. Skyline queries represent a computationally demanding phase of several environmental monitoring applications [16], in which streams of readings are produced by properly deployed sensors and their actual values interpreted by a real-time decision-making process (e.g., for early fire detection).



DOI: 10.1002/cpe

In this section we review the sequential algorithms for continuous skyline queries, some parallel implementations, and solutions for skyline queries on static sets of data.

### 2.1. Sequential Algorithms

Algorithms for continuous skyline queries maintain a sliding window of the most recent points [9]. Since the skyline changes over the time due to point expiration and arrival, the output is a sequence of *skyline updates*, i.e. which points are inserted/removed in/from the skyline at each time instant.

The first algorithm has been proposed in [11] for $n$-of-$N$ queries, which consist in computing the skyline for the most recent $n$ tuples ($\forall\, n \leq N$) where $N$ is an upper bound to the window size. The algorithm supports the pruning of *obsolete tuples*, i.e. non-expired tuples that cannot enter the skyline set and can be safely discarded. The algorithm is designed only for count-based windows.

More often sliding windows have a temporal semantics. The seminal work for time-based continuous skyline queries is represented by [10]. In this paper the authors propose a *lazy* and an *eager* algorithm. The first one performs most of the activities when a skyline point expires and the stored points that will appear in the skyline must be identified. In contrast, the eager variant performs an intensive preprocessing at the reception of each new point in order to: *i)* expunge tuples from the system as early as possible (*pruning*) by keeping only those tuples that may enter the skyline in the future; *ii)* minimize the actions performed when a skyline point expires.

Another algorithm is *LookOut* [14]. With respect to eager and lazy it performs better in the case of anticorrelated datasets. Furthermore, it adopts a more general model in which tuples may have a different lifetime. The algorithm exploits the *Quad Tree* index data structure that, though not balanced in principle, is more efficient in the insertion of new points than the R*-trees used in [10]. Most of the benefits achieved by LookOut with anticorrelated datasets are due to the use of quad trees rather than being an inherent feature of the algorithm.

### 2.2. Parallel Implementations

Continuous skyline queries have been studied in distributed environments. Data are often generated by data sites and streamed directly to a central server that evaluates the query. Various solutions require a pre-processing on data sites to reduce the data sent to the server. In [17] this class of skyline operators has been evaluated over time-based windows, while in [18] the authors assume intelligent data sites that notify the server only of the changes that may affect the global skyline.

All the aforementioned works assume the centralized processing of the query. Parallel implementations have been rarely investigated and they represent an open research issue. Parallel skyline queries over *uncertain streams* (i.e. in which the values of the attributes of each input point have a trustability threshold) have been discussed in [12] by proposing parallel solutions for multicore CPUs. The authors of that work argue that parallel solutions are mainly useful for uncertain streams, where the complexity of the skyline computation makes the whole problem more difficult to be solved in real-time. We agree with them. However, due to the recent explosion of data availability, the real-time execution of skyline queries over classic (certain) streams has also become a time-consuming task.

### 2.3. Traditional Approaches on Static Datasets

The problem of computing the skyline on a static set of data (e.g., finite relations) is different than the counterpart on data streams. In fact, the set of data relevant to the skyline computation are all available before starting the processing, and the output is a single skyline set rather than a sequence of incremental updates. Therefore, the problem requires different procedures and algorithms.

Solutions have been proposed for various architectures such as multicores [19], P2P systems [20, 21], shared-nothing machines [22] and FPGAs [23]. Most of these approaches share the idea of partitioning the whole dataset into subsets, computing the local skylines in parallel, and composing the final result by merging the local results. An important work is the BBS algorithm presented in [24], based on a Branch & Bound approach that traverses a R*-tree using a best-first search and inserts entries ordered by a specified distance function.



DOI: 10.1002/cpe

One of the most critical aspects is to define effective partitioning strategies of the dataset among the processors. Various solutions have been proposed [25]. The most common is a *grid-based* partitioning [22, 21]. The idea is to create a grid on the data such that each cell has roughly the same number of points. The local skyline of each cell is computed in parallel by different processors. The main drawback is that many partitions may not contribute to the final skyline. A partitioning technique applied in [26] is based on a *random* partitioning that distributes the points uniformly among the partitions. Also in this case many points may belong to the local skyline but not to the final skyline. An *angle-based* partitioning scheme has been proposed by [27]. The algorithm transforms the points using hyper-spherical coordinates. In this way the skyline points are equally spread in all the partitions and their contribution to the global skyline is approximately the same.

## 3. SKYLINE COMPUTATION OVER DATA STREAMS

The goal of the skyline query is to determine the points received in the last $T_w$ time units that belong to the skyline set. Each point $x$ is represented as a tuple of $d \geq 1$ attributes $x = \{x_1, x_2, \ldots, x_d\}$. Given two points $x$ and $y$, we say that $x$ *dominates* $y$ (denoted by $x \prec y$) if and only if $\forall i \in [1, d]\ x_i \leq y_i$ and $\exists j \mid x_j < y_j$. The skyline of a given set $\mathcal{S}$ is the subset of all the points not dominated by any other point in $\mathcal{S}$. The output is a stream of *skyline updates* expressed in the format `(action, p, t)`, where *action* indicates whether the point $p$ enters (`ADD`) or exits (`DEL`) the skyline at time $t$. The computation has three main phases:

- *Point expiration*: a skyline point $x$ will be removed from the system when it reaches its expiration time;
- *Point maintenance*: if a new point $x$ is not a skyline point at its arrival time, it cannot be discarded immediately because it could become a skyline point when all its dominators expire;
- *Pruning*: once a point $x$ arrives, the older points dominated by it (*obsolete*) can be discarded.

The received points that cannot be pruned (*non-obsolete points*) must be stored in a data structure denoted by $\mathcal{DB}$. This data structure implements an abstract data type with the following operations: the *insertion* of a new point, the *removal* of an existing point, and the *search* of the critical dominator of a given point $p$, i.e. the youngest point $r$ which dominates $p$.

In this work we study the parallelization of the *eager* algorithm described in [10]. The algorithm computes for each incoming tuple $p$ the minimum time in which $p$ may become a skyline point. This time is referred to as the *Skyline Influence Time* of $p$ (briefly, $\texttt{SIT}_p$). When the system time reaches $\texttt{SIT}_p$, if point $p$ is still in the system the algorithm immediately adds the point to the skyline and produces an `ADD` update onto the output stream. Otherwise, if a younger tuple $r$ dominating $p$ is received before the system time reaches $\texttt{SIT}_p$, $p$ is discarded (*pruned*).

There are several reasons that make the eager algorithm very attractive. First, it requires a lower memory occupancy in most of the situations because it applies pruning systematically. Second, it is characterized by a more stable processing time per point than the lazy variant [10]. Furthermore, it maintains a single data structure for the points instead of two separated data structures for the skyline set and for the candidate points as in lazy. However, it requires a higher computational cost.

### *3.1. SIT-based Algorithm*

The main concept exploited by the eager algorithm is the Skyline Influence Time of a point:

**Definition 3.1.** The Skyline Influence Time of a point $p$ (denoted by $\texttt{SIT}_p$) is the expiring time of the youngest point $r \in \mathcal{S}$ dominating $p$ (its critical dominator).

The algorithm maintains an *event list* $\mathcal{EL}$ (typically indexed by a B-tree [10]). Two types of events are supported: *i) skytime(p,t)* indicates that point $p$ will enter the skyline at time $t$, *ii) expire(p,t)* indicates the exit of point $p$ from the skyline at time $t$. The rationale of the eager algorithm is to perform most of the work during the reception of a new point in order to update the event list. Then,




DOI: 10.1002/cpe

the events are processed chronologically by emitting the changes in the skyline set through ADD and DEL updates transmitted onto the output stream of the computation.

At the reception of a new point $p$ the algorithm executes the following steps:

1. *Pruning phase*: all the points in $\mathcal{DB}$ dominated by $p$ must be removed and their associated events cleared from $\mathcal{EL}$. If a removed point was part of the skyline, a DEL update must be emitted onto the output stream;
2. *Insertion phase*: the new point $p$ is added to $\mathcal{DB}$;
3. *Search phase*: the critical dominator $r$ of $p$ in $\mathcal{DB}$ must be found. If it exists, we add the event *skytime(p, $r.t^{exp}$)* into $\mathcal{EL}$ where $r.t^{exp}$ is the expiring time of $r$, i.e. $\mathtt{SIT}_p = r.t^{exp}$. Otherwise, if $r$ does not exist, $p$ becomes a skyline point immediately and we add the event *expire(p, $p.t^{exp}$)* into the event list.

The algorithm processes the events in the chronological order. When the time of an event has been reached, there are two possibilities:

- in the case of a *skytime(p,t)* event, the point $p$ is added to the skyline and an ADD update is emitted. A new event *expire(p,$p.t^{exp}$)* is added to $\mathcal{EL}$;
- in the case of an *expire* event, the associated point (which is part of the actual skyline) is removed from $\mathcal{DB}$ and a DEL update is transmitted onto the output stream.

This behavior allows us to define an important property (the proof can be found in [10]):

**Lemma 3.1.** If a point $p$ reaches its expiration time, that point is part of the skyline at that time.

Therefore, for each non-obsolete point $p$ there are two possible situations. If a new point $r$ that dominates $p$ is received before the system time reaches $\mathtt{SIT}_p$, $p$ is pruned and its *skytime* event cleared from $\mathcal{EL}$. Otherwise, when the system time reaches $\mathtt{SIT}_p$ the corresponding *skytime* event is executed, $p$ enters the skyline and an ADD update is emitted. When point $p$ reaches its expiration time, for Lemma 3.1 it is part of the skyline. The point is deleted and a DEL update is emitted. The expiration time of $p$ matches the skyline influence time of the points critically dominated by it, that are exactly the ones that have to be inserted into the skyline as a consequence of p's expiration (each one of them will have a corresponding *skytime* event in $\mathcal{EL}$).

## 4. PARALLEL IMPLEMENTATION ON MULTICORES

Our parallelization is based on the *data-parallel* paradigm, in which the data structures $\mathcal{DB}$ and $\mathcal{EL}$ are partitioned among a set of parallel *workers* $(W_1, W_2, \ldots, W_n)$. We adopt a partitioning method in which each point is owned by a unique worker for its entire validity time. A discussion and a comparison of different methods will be proposed in Sect. 5.3.

Two other functionalities are part of our parallel implementation: *i)* an *emitter* $(E)$ receives points from the input stream and distributes them to the workers; *ii)* a *collector* $(C)$ receives the skyline updates from the workers and produces a single, time ordered, output stream.

Once a point $p$ has been received by the emitter, it assigns the ownership of $p$ to a specific worker (denoted by $W_k$) according to a specific *partitioning strategy*. Then, the point and the owner identifier are transmitted to *all* the workers. Each worker $W_i$ performs the following actions:

1. discard all the points dominated by $p$ in its local partition $\mathcal{DB}_i$ (*pruning*);
2. calculate its local $\mathtt{SIT}_p^i$ ($\mathtt{SIT}_p$ of $W_i$) by accessing the points in its local partition $\mathcal{DB}_i$;
3. if $i = k$ the worker is the owner of $p$ and stores the point in its $\mathcal{DB}_i$.

To determine the global value of $\mathtt{SIT}_p$ the workers perform a *reduce* phase, i.e. $\mathtt{SIT}_p = \max_{i=1}^{n} \mathtt{SIT}_p^i$ and the event *skytime(p, $\mathtt{SIT}_p$)* is added to $\mathcal{EL}_k$, i.e. the event list of the owner $W_k$, which is the worker needing the result of the reduce operation. Owing to the fine-grained nature of the reduce, it is centralized in the collector: $C$ receives the values of $\mathtt{SIT}_p^i$ from the workers, calculates the global $\mathtt{SIT}_p$ and sends the result to the worker $W_k$ owning the point, as shown in Fig. 2. It is worth noting



DOI: 10.1002/cpe

that the owner of a point has only a slightly greater computation burden than the others, since it needs to wait for the reduce result on that point and to update its event list correspondently. Most of the work (the pruning and the search of the critical dominator) is executed on the same point by all the workers on their partitions in parallel.

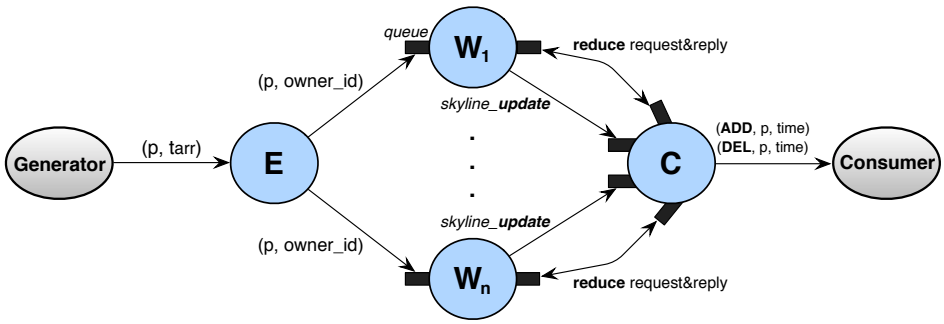



Figure 2. Scheme of the parallel implementation: *emitter*, *worker* and *collector* functionalities.

Our implementation uses the runtime support of `FastFlow` [28], a programming environment designed for streaming applications. The functionalities of the parallel program are written in `C++` using standard `POSIX` threads. Communications between threads are performed using `pop` and `push` operations on *lock-free* `FastFlow` queues [29]. According to the `FastFlow` model, queues are used to exchange memory pointers to shared data structures. In addition, we have two other threads: a *generator* in charge of producing a stream of tuples according to a certain arrival rate and distribution, and a *consumer* responsible for collecting skyline updates. The emitter and the collector functionalities can be alternatively executed by two special worker threads. This can be useful in multicores with a limited number of cores. In the sequel we will assume that they are implemented as separated threads.

### *4.1. Time Synchronization*

The sequential algorithm is characterized by time-driven internal activations, i.e. when a skyline point expires or the `SIT` of a non-obsolete point is reached. On shared-memory machines we have a unified system time. In the sequential implementation the system time is used in two different cases: *i)* to assign a timestamp to each new point, *ii)* to check if the time of a *skytime* or *expire* event has been reached. In the parallel implementation the first case is managed by the emitter, which assigns the timestamp of a new point using the current system time. In particular, if point $p$ arrives before $r$ we have that $p.t^{arr} < r.t^{arr}$.

Each point $p$ is transmitted to all the workers to compute their local `SITs` and to prune the obsolete points from their partitions. These two operations depend on the timestamps assigned to the points by the emitter. Skyline updates are produced by the execution of the *skytime* and *expire* events (second case). Each worker must process these events consistently with the received timestamps: if a worker receives a point $p$ with timestamp $p.t^{arr}$, it must execute all the events with timestamp less than $p.t^{arr}$ before processing the new point. This technical aspect of the implementation will be clarified in the sequel.

Furthermore, skyline updates from different workers may arrive at the collector *unordered*. Therefore, a timestamp-based ordering is required in the collector, see the next section.

### *4.2. Implementation Description*

As explained in Sect. 3, the sequential computation is activated in two cases: *1)* when it receives a new point; *2)* when an event from the event list is raised. In the parallel version the two activations must be properly handled. In particular, the first is handled by the emitter, while the second is handled directly (and independently) by the workers for the points belonging to their partitions. In the sequel we will describe the behavior of the emitter, collector and worker functionalities.



DOI: 10.1002/cpe

***Emitter.*** The emitter assigns a timestamp to each received point, selects the owner according to some partitioning strategy, and transmits each point to *all* the workers. According to the `FastFlow` model, the broadcast to the workers is performed *by reference*, i.e. by passing the initial pointer of $p$ to the workers. Therefore, the emitter service time is independent from the point size. The push and pop primitives on the `FastFlow` queues are very efficient and require few hundreds of nanoseconds on cache-coherent multicore architectures [29]. This allows the emitter to be very fast in performing the broadcast. Therefore, it is never a bottleneck in our test cases.

***Worker.*** At the arrival of a new point $p$ all the workers execute the following steps on their partition of $\mathcal{DB}$. In particular each worker $W_i$:

1. prunes from $\mathcal{DB}_i$ all the points dominated by $p$; the private event list $\mathcal{EL}_i$ is cleared from the events related to the pruned points;
2. for each skyline point pruned at the previous step, $W_i$ generates a `DEL` update to the collector;
3. calculates the local $\texttt{SIT}_p^i$. If $p$ has no dominator in $\mathcal{DB}_i$, then $\texttt{SIT}_p^i$ is zero. $W_i$ sends the value of $\texttt{SIT}_p^i$ to the collector for the reduce phase.

After these steps, the owner (denoted by $W_k$) adds $p$ to its partition and then waits for the completion of the reduce phase. Once $\texttt{SIT}_p$ has been received, the owner:

4. sends an `ADD` update to the collector if $\texttt{SIT}_p = 0$, i.e. $p$ must be added to the skyline immediately;
5. updates $\mathcal{EL}_i$ by adding: a *skytime* event with time $\texttt{SIT}_p$ if $\texttt{SIT}_p > 0$, or an *expire* event with time $p.t^{exp}$ if $\texttt{SIT}_p = 0$.

As previously discussed, each worker must produce chronologically ordered updates. To this end, at each arrival of a point $p$ we must guarantee that: *i)* all the events with a timestamp smaller than $p.t^{arr}$ have been raised; *ii)* no event with timestamp greater than $p.t^{arr}$ has been raised. In our implementation the workers process their event lists according to the timestamps of the new received points. When worker $W_i$ receives a new point $p$, before starting the computation related to this new point it evaluates *all* the events in $\mathcal{EL}_i$ with timestamp smaller than $p.t^{arr}$. That is, all the fired events are executed (unrolled) in the correct order at the beginning of the processing on each new received point. This solution may represent a problem if the input stream is excessively slow. To avoid this problem, the emitter transmits periodically a *punctuation* [30] (a null point) marked with the current time to the workers in order to go on in the processing of their event lists.

***Collector.*** The collector is in charge of performing two important tasks: the *reduce* phase of the algorithm, and the *time-based ordering* of skyline updates received from different workers.

For the reduce phase, each worker calculates its $\texttt{SIT}_p^i$ and sends it to the collector. When all the local `SIT` values have been received, the collector sends the value of the global $\texttt{SIT}_p$ to to the owner. The collector is able to manage multiple reduce requests for different points simultaneously.

The collector must preserve the chronological order of the updates on the output stream. To do that the collector buffers the updates and keeps them ordered by timestamp using a priority queue $\mathcal{Q}$. It also keeps, for each worker $W_i$, the timestamp $last\_T_i$ of the last update received from that worker. By assuming that each worker produces ordered updates to the collector, $W_i$ cannot generate next updates with a timestamp lower than $last\_T_i$. Therefore, the collector can safely forward all the buffered updates with a timestamp $T$ such that $T \leq \min_{i=1}^{n} last\_T_i$.

## 5. OPTIMIZATIONS

In this section we focus on three aspects of the parallel implementation: *i)* how to store the set of non-obsolete points in memory, *ii)* how to optimize the reduce phase, *iii)* how to keep the size of the partitions as similar as possible in order to balance the workload among workers. In the first part of the description we show results with five dimensions per tuple, which is a typical size used in past works [11, 12, 14]. Experiments with greater dimensionalities will be shown at the end of the paper.



DOI: 10.1002/cpe

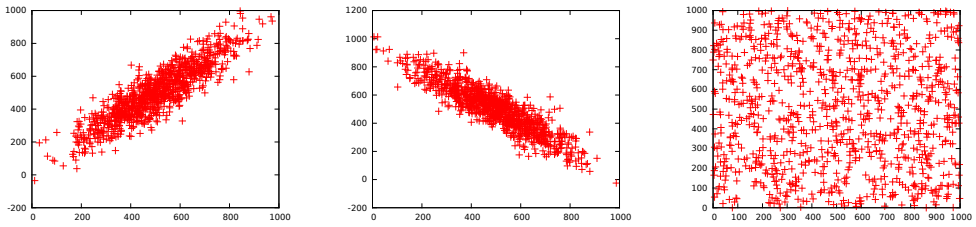
### 5.1. Spatial Distribution and Data Structures

Each partition $\mathcal{DB}_i$ is an abstract data type with the following operations: *insertion* and *removal* of a point, and $youngestDominator(p)$ to find the youngest point $r \in \mathcal{DB}_i$ that dominates the point $p$. In this paper we will use arrays, which allow the insertion of a new point and the removal of an existing one to be performed in constant time (points do not need to be stored in the arrival order), while the search of the youngest dominator requires to scan the entire set at the worst case. In a single scan of the array we are able to:

- find the points dominated by the new point $p$ and delete them from the data structure;
- find the youngest dominator of $p$ and calculate the $\texttt{SIT}_p^i$;
- insert $p$ in the database (if the partition is the one of the $p$'s owner).

To improve spatial locality, we can use an *attribute-oriented organization* [31]: $d$ separated arrays store the values of the same coordinate of different points contiguously in memory. This layout enables the vectorization of the code of the youngest dominator search.

The use of arrays is justified by the specific properties of the SIT-based algorithm, which requires to maintain only the set of non-obsolete points. The size of $\mathcal{DB}$ depends on the pruning probability that in turn depends on the spatial distribution of data. We will consider three *synthetic datasets* generated according to spatial distributions popular in the skyline literature [10, 14]: *1)* the *independent* distribution in which the coordinates of the points are uniformly distributed in their domain; *2)* the *anticorrelated* distribution in which the points exhibit a strong anticorrelation; *3)* the *correlated* distribution in which the coordinate values are generated with a high correlation coefficient. These distributions are sketched in Fig. 3 for bidimensional datasets. As we can see, the correlated distribution has the smallest number of skyline points. The anticorrelated distribution maintains the largest skyline whereas the independent one represents an intermediate case.



(a) Correlated. (b) Anticorrelated. (c) Independent.

Figure 3. Correlated, anticorrelated and independent distributions on a bidimensional space.

Fig. 4 shows the number of non-obsolete points ($|\mathcal{DB}|$) as a function of the points received per window $|W| = \lambda T_w$, where $\lambda$ denotes the arrival rate. The pruning probability $p$, equal to $p = 1 - |\mathcal{DB}|/|W|$, is in general high, with the highest values with the correlated dataset. With millions of points received per window, the SIT-based algorithm needs to maintain in memory few thousands of points, leading to scenarios in which the array-based implementation is a good solution. This consideration is further emphasized by the data parallel paradigm, in which the partitions are smaller with higher parallelism degrees. In very pessimistic cases of very large sets of non-obsolete points (in the order of millions), arrays could lead to a too high processing time per tuple, and the use of index structures like R-trees and R*-trees [10, 32] could be beneficial. The use of such index structures in our parallelization will be studied in our future works. In our implementation the $\mathcal{DB}$ is implemented by dynamically resizable arrays (the basic size is of 100 tuples).

### 5.2. Asynchronous Reduce

The skyline influence time of a new point $p$ is calculated through a reduce phase involving all the workers and the collector. At the end of this phase, the value of $\texttt{SIT}_p$ is transmitted by the collector to the worker $W_k$ that has the ownership of that point. In the basic version the reduce is performed in a *synchronous* fashion: $W_k$ waits explicitly for the value of $\texttt{SIT}_p$. The waiting time depends on



DOI: 10.1002/cpe

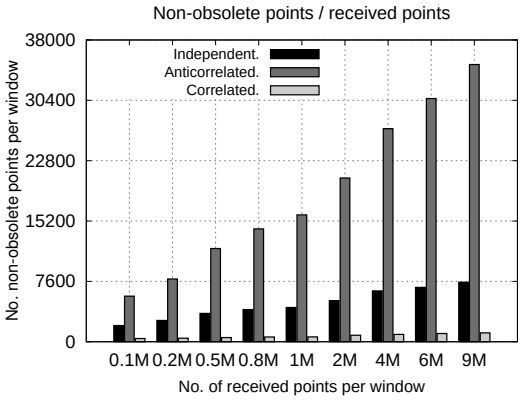



Figure 4. Number of non-obsolete points: correlated, anticorrelated and independent distributions, $d = 5$.

the *latency* of the reduce operation, which in turn depends on the computation time of the pruning phase and the search of the youngest dominator of $p$ performed in parallel by the workers on their partitions. If the partitions are not evenly sized, this latency is the one of the slowest worker.

The idea is to make the reduce *asynchronous*: $W_k$ does not have to wait for the completion of the reduce phase, but it can process the subsequent points received from the emitter while the reduce result is not available yet. This behavior is correct because the computation on the subsequent points does not depend on the result of the reduce phase on point $p$. In fact, both the search of the youngest dominator and the pruning phase use only the spatial coordinates of the points and their expire time. The asynchronous reduce works as follows:

- when a new point $p$ is received by a worker, whether it is the owner or not it participates to the reduce phase without waiting for the result;
- the owner $W_k$ non-deterministically waits for the reception of a message either from the emitter (a new point) and from the collector (the result of the *pending* reduce on $p$).

Although the pruning and the search of the youngest dominator do not depend on the value of $\texttt{SIT}_p$, the workers must pay attention to how skyline updates are generated and transmitted to the collector. In fact, the asynchronous reduce is a sort of out-of-order processing in which the end of the computation on point $p$ is postponed until the value of $\texttt{SIT}_p$ is available. During the time interval from the beginning of the reduce computation on $p$ and its completion, other points can be received by $W_k$ and processed and some skyline updates possibly generated. In particular:

- all the skyline updates generated during such time interval (i.e. the ones raised by the triggering of *skytime* or *expire* events) will be stored and maintained in the chronological order into a local *reordering buffer* of worker $W_k$;

Furthermore, let $r$ be the first point received by the worker during that time interval, we can recognize the following possibilities:

- $r$ does not dominate $p$. The reception of $r$ has no effect on the updates related to point $p$;
- $r$ dominates $p$ and $\texttt{SIT}_p < r.t^{arr}$: $p$ is marked as deleted, and when the reduce result is available an `ADD` update and a `DEL` update for $p$ are inserted in the pending buffer with time $\texttt{SIT}_p$ and $r.t^{arr}$ respectively. It is worth noting that any successive point $s$ that dominates $p$ does not have any effect on the updates generated on $p$, because $s.t^{arr} \geq r.t^{arr}$;
- $r$ dominates $p$ and $\texttt{SIT}_p \geq r.t^{arr}$: $p$ is marked as deleted and no skyline update related to $p$ is produced after the reception of $\texttt{SIT}_p$ from the collector.

When the reduce result has been computed by the collector, the owner worker flushes to the collector (in the temporal order) all the updates in the pending buffer if no other pending reduce is present, or flushes only the updates with timestamp lower or equal than $\texttt{SIT}_p$ otherwise. It is worth noting



DOI: 10.1002/cpe

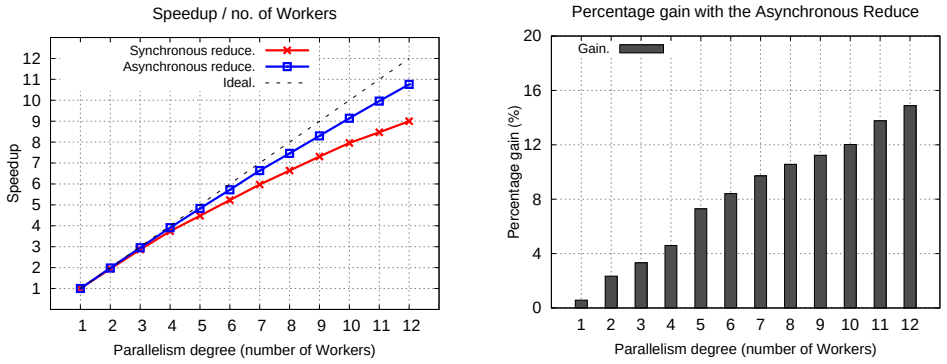
that although a worker can have more pending reduce requests on different points, they will be completed by the collector serially and the results transmitted in that order.

This optimization produces an average gain about 10%-20% under various configurations in terms of arrival rate, window length and point distributions on the Intel multicore described in Fig. 5. Fig. 6 shows the results with a configuration with a window length of 20 seconds and an arrival rate of $100K$ points/sec generated according to the independent distribution ($d = 5$). We map at most one thread per core, i.e. 12 is the maximum number of workers. In this setting the maximum parallelism degree is not sufficient to sustain the input rate, i.e. the so-called *offered bandwidth* is lower than the *required bandwidth* (the input rate). Fig. 6a shows the *speedup* measured as the ratio between the offered bandwidth with $n$ workers and the one of the single-threaded program[†].

| INTEL MULTICORE | AMD MULTICORE |
|---|---|
| A multiprocessor composed of two Xeon Sandy Bridge E5-2650 CPUs for a total of 16 cores working at 2Ghz. Each core is equipped with a private L1d cache of 32KB and a private L2 cache of 256KB. Groups of 8 cores share a L3 cache of 20MB. | A multiprocessor composed of two Opteron 6176 CPUs with 32GB of RAM. Each CPU is composed of two multicore chips featuring six single-threaded cores (for a total of 24 cores). Each core operates at 2.2 GHz with private L1d and L2 cache of size 128KB and 512KB per core and a L3 cache of 6MB shared among the six cores on the same chip. |

Figure 5. Multi-core architectures used in the experiments.

With the asynchronous reduce the speedup is closer to the ideal one. With high parallelism degrees the benefit is greater (see Fig. 6b) because the partitions of $\mathcal{DB}$ are smaller and their relative unbalance is higher. With the synchronous reduce the workers are blocked waiting for the completion of the reduce whose latency depends on the slowest worker. With the asynchronous reduce we are able to mitigate this by achieving a better offered bandwidth.




(a) Speedup. (b) Relative gain.

Figure 6. Asynchronous reduce: $\lambda = 100K$ points/sec, $T_w = 20$ sec and independent distribution.

### 5.3. *Load-balancing Strategies*

In order to balance the workload we need to keep the size of the partitions as similar as possible. We tackle the problem in three different angles:

- the ownership of the points must be assigned in order to balance the size of the partitions;
- the processing of a new point could prune several stored points that might be not uniformly distributed among the workers. Therefore, the strategy must take into account the actual load of the workers instead of distributing the same amount of new points to them;
- we do not assume any knowledge about the point distribution. Therefore, we can not apply static rules that exploit solely the spatial coordinates of the points to assign the ownership.

[†]We consider the *relative speedup* by excluding the emitter/collector threads from the total parallelism degree.



DOI: 10.1002/cpe

In the following we present three heuristics: *On-demand* (OD), *Least Loaded Worker* (LLW) and *Least Loaded Worker with Ownership* (LLW+).

***On-demand strategy.*** The idea of this heuristic is the following: each point is broadcast to the workers and the first able to accommodate the new point in its input queue becomes the owner. This heuristic works well when we have a small number of enqueued points on average. In fact, it does not distinguish between a queue which is almost full and one which is almost empty. In principle, if the computation is not a bottleneck the queues between the emitter and the workers are empty on average. However, due to possible high variance of the stream arrival rate and/or of the internal processing time per point, it is possible that for some transient periods some points are enqueued waiting to be processed. If the number of enqueued points is small, the difference between "almost full" and "almost empty" is negligible and load balancing is good. Otherwise this difference may become very large, thus hampering load balancing.

The experiments on the Intel multicore confirm this behavior. Fig. 7a shows the difference $\Delta$ between the number of non-obsolete points owned by the most loaded worker and the one of the least loaded worker . In the experiment we use 12 workers and the independent distribution ($d = 5$). Similar results are achieved with different configurations.


(a) OD strategy. (b) LLW strategy.

Figure 7. Load difference with the OD (a) and the LLW (b) strategy and different queues sizes. Configuration: $\lambda = 100K$ points/sec, $T_w = 10$ seconds and independent distribution.

With this configuration the computation is a bottleneck, and the input queues in front of the workers grow up to reaching their maximum capacity (queue size). We consider queue sizes of 1, 5 and 30 points. The experiment shown in Fig. 7a consists in 50 seconds of execution in which we measure the factor $\Delta$ with a sampling interval of 100 $\mu$sec. As we can observe, the smaller the queues size the lower the load difference. The best results are achieved with queues of one single position: in this case each new point is assigned to a worker that is immediately ready to process it. We can note a transient period in which the load difference is relatively high (near to 250 points) and then it falls rapidly near to zero. This is because at the beginning of the execution the ownership of the points is assigned to the first worker as long as it is able to sustain the input stream rate.

***LLW strategy.*** This strategy gives to the emitter the knowledge of the number of points owned by each worker. The owner selection is done by selecting the worker with the smallest partition. To know the actual size of the partitions, workers and the emitter share atomic integer counters (instances of the `std::atomic` class of the standard C++ library). Fig. 7b shows the results of the same experiment of Fig. 7a in which we use the LLW strategy instead of OD.

Also in this case the number of enqueued points plays an important role. Consider the case in which at time $t$ the emitter assigns the ownership of a new point $p$ to worker $W_i$ that is the least loaded one. This worker will start the computation on $p$ at time $t + \tau$, where $\tau$ is the time interval needed by the worker to dequeue and execute all the points already in its input queue before point



DOI: 10.1002/cpe

$p$. When $p$ is extracted from the queue, $W_i$ might no longer be the least loaded worker, since the execution of the points during the interval $\tau$ can modify the size of its partition.

This behavior is confirmed by the experiments in Fig. 7b. With a queue size of 5 elements, the load difference is smaller than the one achieved with the same queue size and the `OD` strategy. We also measure the load difference with bigger queue sizes of 100 and $10K$ elements.

***LLW+ strategy.*** To reduce the error in the selection of the least loaded worker, the emitter counts for each worker the number of enqueued points for which it has been designed as the owner. Fig. 8 shows the results under the same configuration of Figs. 7a and 7b. `LLW+` is also affected by the length of the queues, however, the load difference $\Delta$ is smaller than for `LLW`.

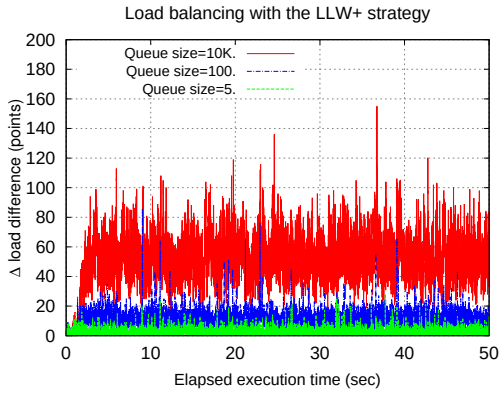



Figure 8. Load difference with `LLW+` strategy and different queues sizes. Configuration: $\lambda = 100K$ points/sec, $T_w = 10$ seconds and independent distribution.

***Comparison.*** We conclude this analysis with a comparison among the aforementioned strategies. In addition, we study a strategy unaware of the actual load of the workers, i.e. the *Round-Robin* strategy (`RR`) in which the ownership is simply assigned in an interleaved manner.

Fig. 9a shows the result of an experiment using small queue sizes (i.e. few positions). We can observe that the `LLW` and `LLW+` strategies provide very similar results. This is an expected result, since with few enqueued points the two strategies approximate the worker load similarly.

Fig. 9b shows the results with more enqueued points. We omit the `OD` strategy because the load unbalance is too high with respect to the other strategies. `LLW+` performs better than the others. As confirmed by the data reported in Tab. I, this strategy is able to maintain the lowest mean load difference and variance under various scenarios in terms of queue sizes.

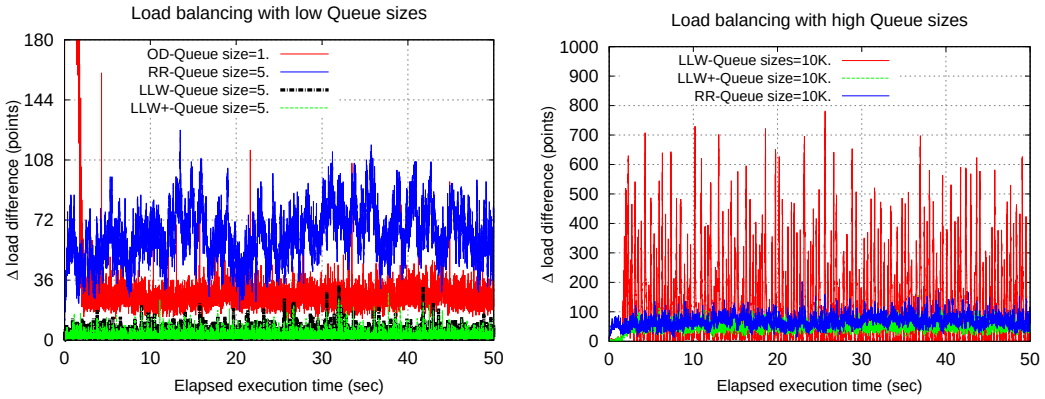
In conclusion, all the strategies are influenced by the average number of enqueued points between the emitter and the workers. In general, the load difference gradually gets worse with many enqueued points, that is in cases in which the computation is near being a bottleneck with the actual stream rate and window length. In scenarios in which the implementation is broadly able to sustain the input stream pressure, few points are present in the input queues on average, and all the strategies behave similarly. The comparison shows that the `LLW+` heuristic is clearly the winner, as it is able to reach the best results in terms of load balancing even in cases in which we have many enqueued points waiting to be served by the workers. This is a good property, because large buffers are useful in general to sustain irregular streams without blocking the computation.

## 6. EXPERIMENTS

Our parallel implementation has been evaluated on the Intel and the AMD multicore architectures described in Fig. 5. For the experiments we use the `gcc` compiler version 4.8.2 with the `-O3`



DOI: 10.1002/cpe




(a) Comparison with small queues.

(b) Comparison with large queues.

Figure 9. Load difference with different strategies. Configuration: $\lambda = 100K$ points/sec, $T_w = 10$ seconds and independent distribution.

| Queue size | RR | | OD | | LLW | | LLW+ | |
|---|---|---|---|---|---|---|---|---|
| | Mean | Var. | Mean | Var. | Mean | Var. | Mean | Var. |
| 1 | 64.02 | 229.18 | 34.13 | 1790.67 | 3.15 | 4.28 | 2.55 | 4.05 |
| 5 | 64.20 | 230.88 | 581.45 | 3948.31 | 3.64 | 4.60 | 2.61 | 4.33 |
| 100 | 64.54 | 246.26 | * | * | 12.93 | 136.26 | 10.89 | 23.06 |
| 10K | 70.06 | 424.65 | * | * | 104.60 | 16483.29 | 50.90 | 280.42 |

Table I. Load difference between the most loaded worker and the least loaded one with difference strategies. *Very large values.

optimization flag. All the threads are mapped onto the cores in an exclusive fashion, i.e. "at most one thread per core". The use of hyperthreading on Intel produces noisy results with cases in which we measure a slight benefit and, more often, cases with lower performance than exclusive mapping. This is justified by the fact that `FastFlow` makes use of an aggressive busy-waiting spinloop to access the lock-free queues [29]. This intensively consumes important CPU resources. Therefore, for the sake of uniformity in all the experiments we set the maximum parallelism degree to 12 and 20 workers on the two architectures respectively (4 cores are used for the generator, emitter, collector and consumer threads, see Sect. 4). All the results shown in this section are the average values of 10 runs. The variance is always small and we can omit to show the error bars in the plots.

### *6.1. Varying the Number of Stored Points*

In the first set of experiments we study the performance of the parallel skyline implementation by varying the number of points received during the timespan $T_w$ of a sliding window. We show two classes of experiments. In the first one we fix the arrival rate $\lambda$ and we vary the window length $T_w$. In the second we do the opposite.

We refer to the *service time* (denoted by $T_S$) as the average time interval between the beginning of the executions on two consecutive points received from the input stream. In the sequential single-threaded implementation, the service time is equal to the processing time per point:

$$T_S^{(1)} = T_F \times T_w \times \lambda \times (1 - p) \tag{1}$$

where $T_F$ is the calculation time per point, $T_w$ is the window length, $\lambda$ is the stream arrival rate and $p$ is the pruning probability, i.e. the probability that a point $p$ received at time instant $p.t^{arr}$ is dominated by a younger point $r$ received before $p$ expires, i.e. s.t. $r.t^{arr} < p.t^{arr} + T_w = p.t^{exp}$.

If the workload is balanced among workers, the service time with parallelism degree $n$ is roughly equal to $T_S^{(n)} \simeq T_S^{(1)}/n$. The inverse of the service time is the offered processing bandwidth $B_w^{(n)} = 1/T_S^{(n)}$, i.e. the average number of points that the implementation can serve in a time unit.



DOI: 10.1002/cpe

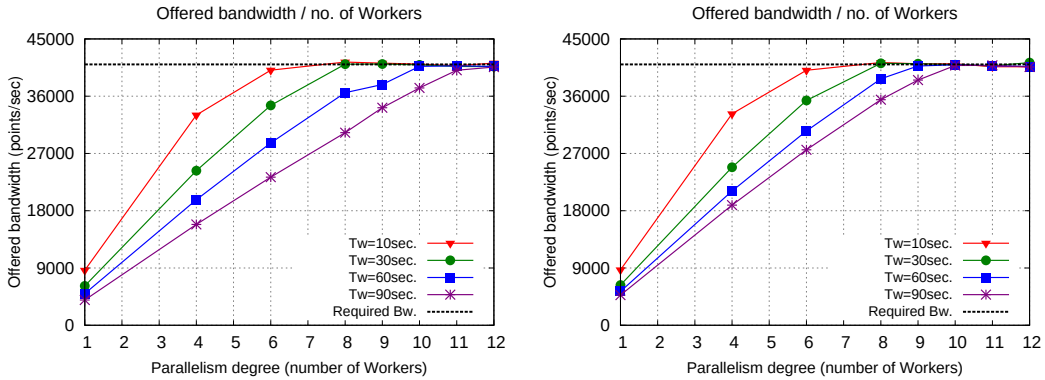
If the offered bandwidth is greater or equal to the stream arrival rate (the *requested processing bandwidth*), we are able to sustain the stream rate. Otherwise, the computation is a *bottleneck*.

To measure the offered bandwidth in cases in which the parallelization is a bottleneck, we change the behavior of the emitter that does not set the timestamps of the points, but they are filled by the generator by reading them from the dataset file. In this way the timestamps always correspond to the real stream rate, even if the computation is a bottleneck and the emitter is periodically blocked due to the backpressure on the workers' queues. We have used this configuration for all the experiments of Sect. 6.1 and Sect. 6.2 (and also for the results of Fig. 6).

Unless otherwise noted, we adopt the `LLW+` heuristic discussed in Sect. 5.3 and we use a fixed dimensionality of $d = 5$.

***Varying the window length.*** In this experiment we fix the arrival rate from the input stream to $\lambda = 40K$ points/sec and we show how the offered bandwidth increases by adding more workers. When the offered bandwidth equals the requested one, i.e. $B_w^{(n)} = \lambda$, the computation is not a bottleneck and the offered bandwidth stops growing if we add more workers.

Fig. 10a shows the offered bandwidth as a function of the parallelism degree on the Intel architecture with the anticorrelated distribution. Analogously to past research works [10, 12], we use window lengths of few tens of seconds (10, 30, 60 and 90 seconds).




(a) Anticorrelated. (b) Independent.

Figure 10. Effect of the window length on the offered bandwidth as a function of the parallelism degree on the Intel architecture. Anticorrelated (a) and independent (b) distributions with input rate of $40K$ points/sec.

The *optimal parallelism degree* $\overline{n}$ is the minimum number of workers such that the offered bandwidth equals the requested one. The optimal parallelism degree is greater with longer windows, since more points are stored in the partitions. In the experiments, 8 workers are sufficient to remove the bottleneck with a window length of 10 and 30 seconds. For longer windows of 60 and 90 seconds, 10 and 12 workers are needed. Fig. 10b shows the results with the independent distribution. As expected, the computation is able to sustain the same stream arrival rate with fewer workers.

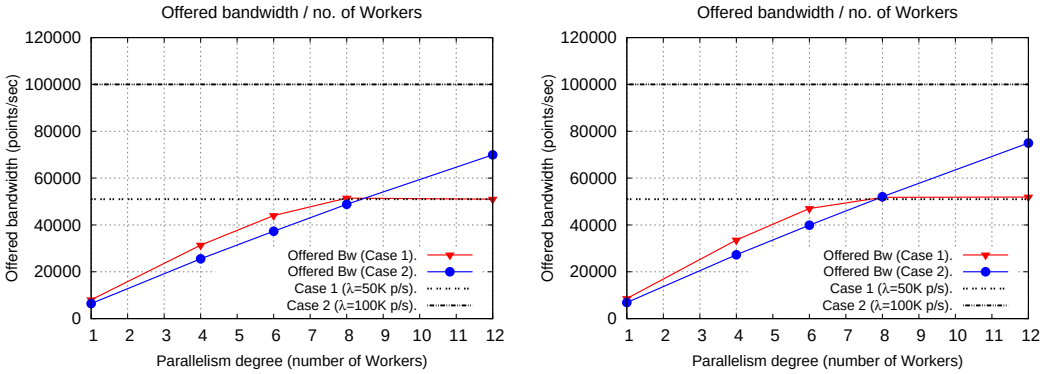
We omit the results of the correlated distribution because, due to the very intensive pruning, the bottleneck can be removed with very low parallelism degrees (1 or 2 workers).

***Varying the input stream rate.*** We study the processing bandwidth by varying the arrival rate and fixing the window length. Fig. 11a shows two distinct scenarios. In the first one with a stream rate of $50K$ points/second, we need 8 workers to reach the required bandwidth. In the second scenario with a rate of $100K$ points/second the bottleneck cannot be removed.

An interesting aspect can be observed by comparing the results of Fig. 10a with the ones of Fig. 11a. In the former, with a rate of $40K$ points/second and a window length of 30 seconds ($1.2M$ points received per sliding window), the bottleneck is removed with $\overline{n} = 8$ workers. In the latter, with a rate of $100K$ points/second and a window length of 10 seconds ($1M$ points per window), we are not able to remove the bottleneck with the available number of physical cores. This behavior



DOI: 10.1002/cpe




(a) Anticorrelated. (b) Independent.

Figure 11. Effect of the input stream rate on the offered bandwidth. Anticorrelated (a) and independent (b) distributions with window length $T_w = 10$ seconds on the Intel architecture.

can be explained by studying the cost model of the skyline computation. By assuming the workload perfectly balanced, the offered bandwidth with $n$ workers is:

$$B_w^{(n)} = \frac{1}{T_S^{(n)}} = \frac{n}{T_S^{(1)}} = \frac{n}{T_F \times T_w \times \lambda \times (1-p)} \tag{2}$$

The optimal parallelism degree can be determined as follows:

$$\overline{n} = \lambda T_S^{(1)} = T_F \times T_w \times \lambda^2 \times (1-p) \tag{3}$$

Even if $\lambda \times T_w$ is fixed, the contribution of the two terms (arrival rate and window length) to the optimal parallelism degree is different, since $\overline{n}$ grows as the square of $\lambda$ and linearly with $T_w$.

Fig. 11b shows the results with the independent distribution. In this case there is a smaller number of stored points with respect to the anticorrelated distribution and the offered bandwidth is higher.

### 6.2. *Evaluation of the Load-balancing Strategies*

In this section we analyze the impact of the partitioning strategies on the speedup. We choose for each data distribution a set of parameters ($\lambda$, $T_w$) such that the computation is a bottleneck. The speedup is measured in relation to the offered bandwidth of the sequential implementation, hence independently from the partitioning strategy.

Tab. II shows the configuration parameters of the experiments for the three distributions ($d = 5$), the number of non-obsolete points and the measured pruning probability. For the correlated distribution we consider a longer window length and a higher stream rate in order to have a sufficient computation grain. The size of all the `FastFlow` queues has been configured to a maximum capacity of $10K$ points (only references are stored, for a total of about 80 KB per queue). In the on-demand strategy we set the queue size to the minimum value of 1 position.

| Distribution | $T_w$ | $\lambda$ | Non-obs. points | Pruning prob. |
|---|---|---|---|---|
| Anticorrelated | 10 sec. | $80K$ point/sec | $13,968$ | 0.9825 |
| Independent | 10 sec. | $100K$ point/sec | $4,296$ | 0.9957 |
| Correlated | 60 sec. | $250K$ point/sec | $1,308$ | 0.9999 |

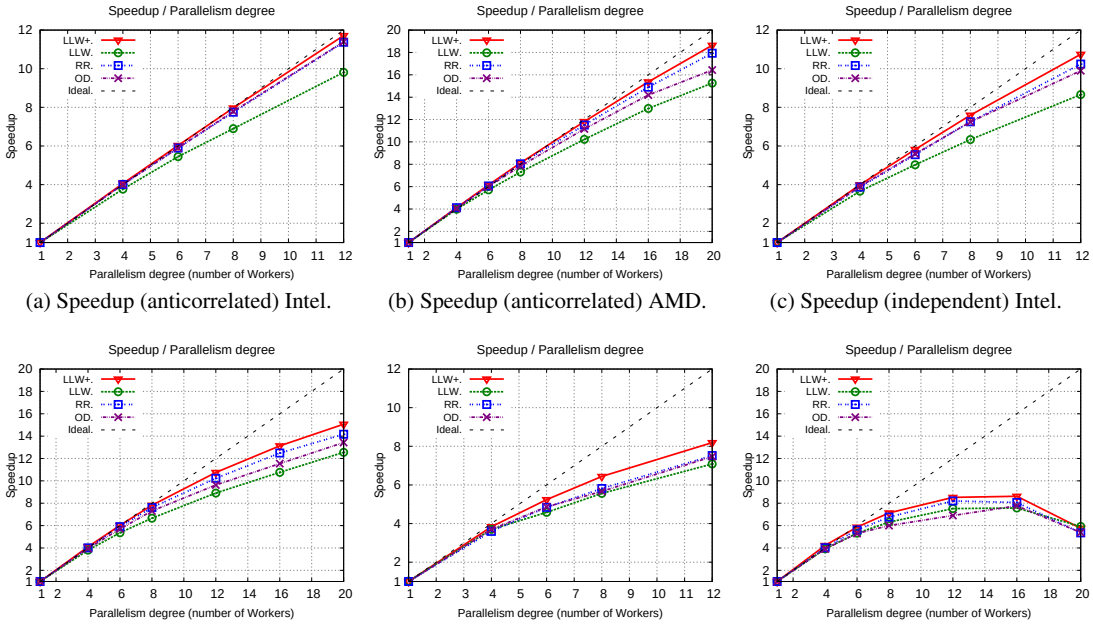
Table II. Configuration parameters, number of non-obsolete points and pruning probability.

Figs. 12a, 12b show the speedup in the anticorrelated case on the Intel and the AMD architectures. As we can observe, the `LLW+` strategy achieves the highest speedup. With $10K$ points in the worker's queues, `LLW` is less effective than the other strategies, as shown in Fig. 9b.



DOI: 10.1002/cpe

Figs. 12c, 12d show the results for the independent distribution. `LLW+` performs worse than in the anticorrelated case. However, it is still the best strategy. Finally, Figs. 12e, 12f show the results with the correlated distribution. In this case the set of non-obsolete points is very small on average. Due to the small size of the worker partitions (few tens of points), a higher relative difference in the partition sizes leads to a significant load unbalance that prevents to achieve a near-optimal speedup. This behavior is more evident on the AMD multicore because we can use higher parallelism degrees.




(a) Speedup (anticorrelated) Intel. (b) Speedup (anticorrelated) AMD. (c) Speedup (independent) Intel.

(d) Speedup (independent) AMD. (e) Speedup (correlated) Intel. (f) Speedup (correlated) AMD.

Figure 12. Effect of the partitioning strategy on the speedup.

### *6.3. Maximum Sustainable Input Rate*

In this section, we measure the maximum input rate $\lambda_{max}^{(n)}$ sustained with $n$ workers. By increasing the input rate more points are maintained in the window and the computation scans more data to produce the outputs. Each arrival rate $\lambda$ such that the following condition is satisfied:

$$\lambda \leq B_w^{(n)} = \frac{n}{T_F \times \lambda \times T_w \times (1-p)}$$

can be sustained by the computation with $n$ workers. Given a parallelism degree $n$, the maximum sustainable input rate can be determined as follows:

$$\lambda_{max}^{(n)} = \sqrt{\frac{n}{T_F \times T_w \times (1-p)}}$$

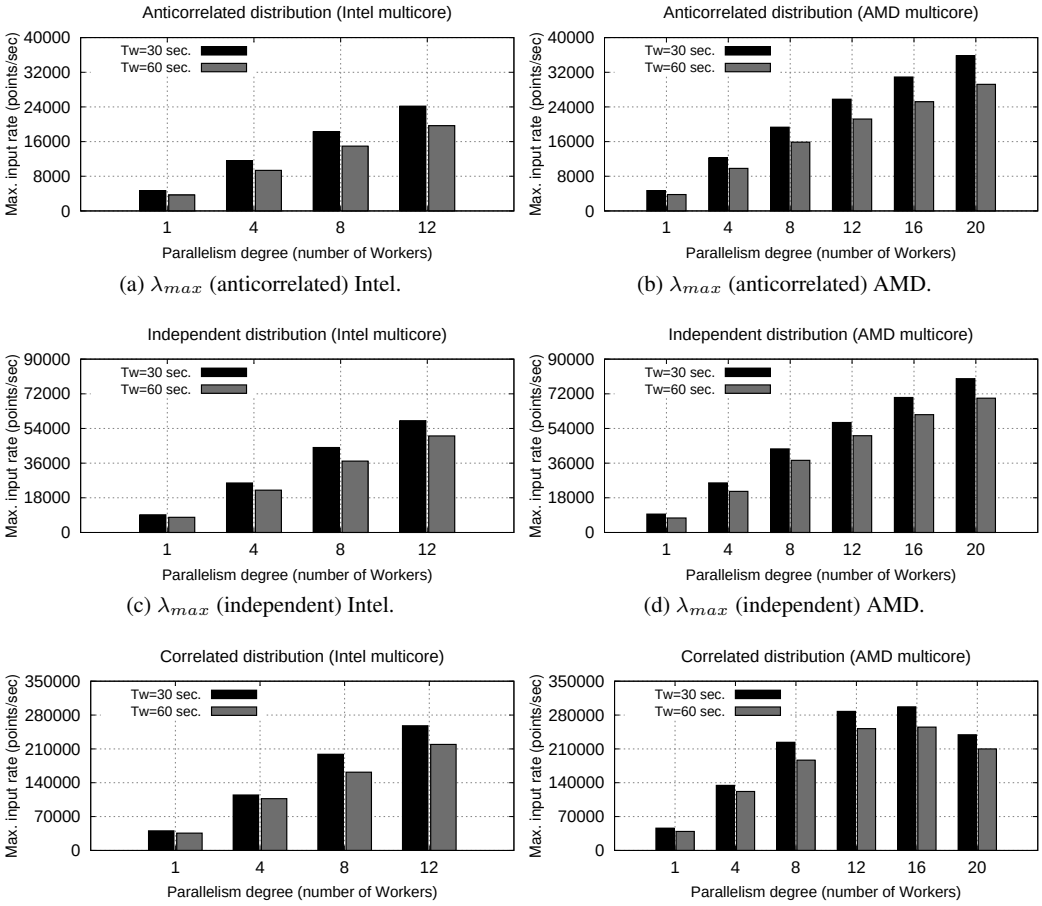
which grows as the square root of the parallelism degree.

The results of Fig. 13 are consistent with this analysis. We consider two window lengths of 30 and 60 seconds. To find the maximum rate we use a bisection method in which we execute several times the experiment by varying the input rate. A bottleneck is detected if the emitter tries to perform a `push` on a full destination queue more than 5 times consecutively. With $T_w = 60$ seconds the maximum rates are lower (about 20%) because more points are in the window. On average up to 12 workers the maximum rates on the AMD multicore are about 1%-13% higher than on Intel.

The highest rates are achieved with the correlated distribution because the computation is more fine grained. On the AMD architecture the correlated case achieves worse results with 20 workers than with 16 (see Fig. 13f) because of a slight load unbalance.



DOI: 10.1002/cpe




(a) $\lambda_{max}$ (anticorrelated) Intel.
(b) $\lambda_{max}$ (anticorrelated) AMD.
(c) $\lambda_{max}$ (independent) Intel.
(d) $\lambda_{max}$ (independent) AMD.
(e) $\lambda_{max}$ (correlated) Intel.
(f) $\lambda_{max}$ (correlated) AMD.

Figure 13. Maximum sustainable input rate per parallelism degree. Window lengths $T_w = 30$ and $60$ seconds. Independent, anticorrelated and correlated distributions. `LLW+` partitioning strategy.

### *6.4. Varying the Dimensionality*

In the previous experiments we have used a fixed dimensionality equal to $d = 5$. In this section we analyze the maximum sustainable input rate with the maximum number of workers as a function of the dimensionality of the points. Fig. 14 shows the results by changing the dimensionality from $4$ to $16$ in order to have a wide spectrum of values. For the sake of brevity we shows the results only for $T_w = 30$ seconds on the Intel multicore. For all the distributions the pruning probability is smaller with higher dimensionalities, and more points are maintained in the partitions leading to lower maximum sustainable input rates.

### *6.5. Evaluation on a Real Dataset*

We analyze our parallelization on a real dataset from the *Lausanne urban canopy experiment* (LUCE). Data were collected from a wireless sensor network within the project SensorScope[‡]. show the results with $T_w = 30$ seconds on the Intel multicore with $12$ workers.

The LUCE project was aimed at understanding the meteorological properties in the urban environment. It consists in high temporal and spatial density measurements. From the available data (radio duty cycle, radio transmission power, radio transmission frequency, primary buffer voltage,

[‡]The dataset can be downloaded from:
http://lcav.epfl.ch/files/content/sites/lcav/files/research/Sensorscope/sensorscope-monitor.zip.



*Prepared using* cpeauth.cls
DOI: 10.1002/cpe

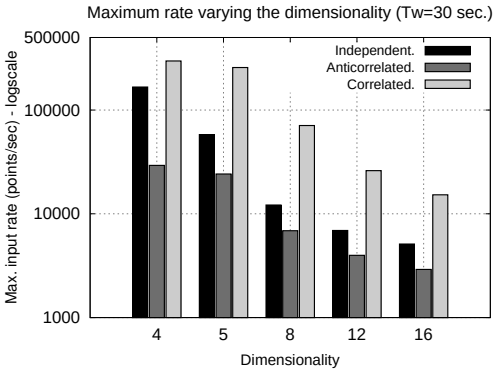
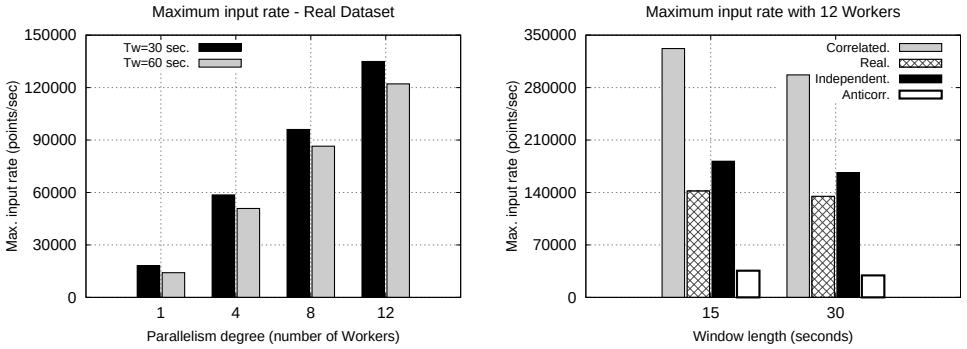



Figure 14. Maximum sustainable input rate on the Intel multicore by varying the dimensionality.

secondary buffer voltage, solar panel current, global current, and energy source), we extract the solar panel current, global current, primary buffer voltage and secondary buffer voltage to create a stream of $d = 4$ points by sorting the sensor readings acquired from different sources (as done in [16]). We have chosen this subset of dimensions because their values exhibit a greater variability (the others have quite constant values in the dataset).

This experiment is important because the data point distribution changes over the time. Fig. 15a shows the maximum sustainable input rate with different parallelism degrees on the Intel architecture. The point coordinates are generated following the real dataset while their timestamps are configured in order to vary the input stream speed by using a uniform timestamp distribution. Due to the size of the dataset (4.7M) we consider window lengths of 15 and 30 seconds.




(a) $\lambda_{max}$ real dataset. (b) Comparison with the synthetic distributions.

Figure 15. Maximum sustainable input rate with the real dataset (a) and comparison with the synthetic distributions (b).

Fig. 15b compares the maximum input rate sustained with 12 workers and $d = 4$ in the correlated, anticorrelated, independent distributions and with the real dataset. As we can see, the highest rate sustained with the real distribution is closer to the one with the independent distribution and higher than the anticorrelated case. Fig. 16 confirms this behavior by showing the total number of points maintained in the worker partitions against the total number of points received per window. The amount of points with the real dataset are slightly more than the independent case.

As we can see, the maximum sustainable input rate with the real dataset is slightly lower than the one with the independent distribution as more points are maintained in the partitions. In conclusion, this experiment confirms that our implementation performs well also with a real dataset generated.



DOI: 10.1002/cpe

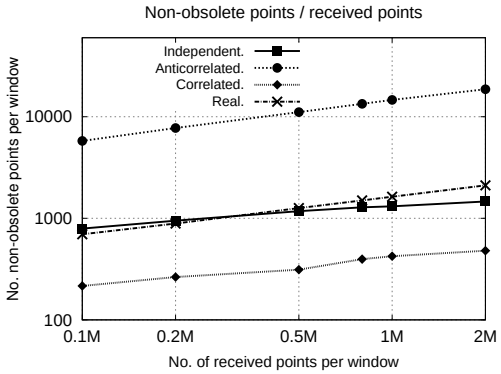



Figure 16. Pruning effect in the real dataset compared with the synthetic datasets, $d = 4$.

## 7. CONCLUSIONS AND FUTURE WORK

In this paper we studied parallel skyline queries based on the eager algorithm. We discussed optimizations of the reduce phase and we proposed several strategies to achieve load balancing. The result is a parallelization that provides good speedup in almost all the most important data distributions, as well as in a real dataset extracted from an environmental monitoring application.

In the future we plan to provide a comparison between our parallelization and other sequential algorithms and their possible parallelizations. Interesting is the analysis of our parallelization with other data structures (indexes) used to maintain the partitions of the points. The comparison will focus on both absolute performance and memory occupancy, possibly by deriving cost models [33]. Furthermore, other load balancing strategies can be studied in the future, such as strategies in which the size of the partitions is estimated using statistical forecasting tools. We expect that these strategies will allow us to achieve good speedup with highly correlated datasets, which still remain a challenging case. Finally, our work deserves to be investigated by providing autonomic supports to dynamic execution conditions (e.g., workload levels, input rates) [34] by changing the number of workers and the partitioning strategy at runtime.

## ACKNOWLEDGMENTS

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

## REFERENCES

1. Andrade H, Gedik B, Turaga D. *Fundamentals of Stream Processing*. Cambridge University Press, 2014. Cambridge Books Online.
2. Börzsönyi S, Kossmann D, Stocker K. The skyline operator. *Proceedings of the 17th International Conference on Data Engineering*, IEEE Computer Society: Washington, DC, USA, 2001; 421–430.
3. Papadimitriou CH, Yannakakis M. Multiobjective query optimization. *Proceedings of the Twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '01, ACM: New York, NY, USA, 2001; 52–59, doi:10.1145/375551.375560.
4. Kung HT, Luccio F, Preparata FP. On finding the maxima of a set of vectors. *J. ACM* Oct 1975; **22**(4):469–476.
5. Tan KL, Eng PK, Ooi BC. Efficient progressive skyline computation. *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 2001; 301–310.
6. Chomicki J, Godfrey P, Gryz J, Liang D. Skyline with presorting. *Data Engineering, 2003. Proceedings. 19th International Conference on*, 2003; 717–719.
7. Papadias D, Tao Y, Fu G, Seeger B. An optimal and progressive algorithm for skyline queries. *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, ACM: New York, NY, USA, 2003; 467–478.
8. Kossmann D, Ramsak F, Rost S. Shooting stars in the sky: An online algorithm for skyline queries. *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, VLDB Endowment, 2002; 275–286.



DOI: 10.1002/cpe

9. Babcock B, Babu S, Datar M, Motwani R, Widom J. Models and issues in data stream systems. *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, ACM: New York, NY, USA, 2002; 1–16.
10. Tao Y, Papadias D. Maintaining sliding window skylines on data streams. *Knowledge and Data Engineering, IEEE Transactions on* March 2006; **18**(3):377–391.
11. Lin X, Yuan Y, Wang W, Lu H. Stabbing the sky: efficient skyline computation over sliding windows. *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, 2005; 502–513.
12. Li X, Wang Y, Li X, Wang Y. Parallelizing skyline queries over uncertain data streams with sliding window partitioning and grid index. *Knowl. Inf. Syst.* Nov 2014; **41**(2):277–309.
13. Zhu L, Li C, Chen H. Efficient computation of reverse skyline on data stream. *Computational Sciences and Optimization, 2009. CSO 2009. International Joint Conference on*, vol. 1, 2009; 735–739.
14. Morse M, Patel J, Grosky WI. Efficient continuous skyline computation. *Data Engineering, 2006. ICDE '06. Proceedings of the 22nd International Conference on*, 2006; 108–108.
15. De Matteis T, Di Girolamo S, Mencagli G. A multicore parallelization of continuous skyline queries on data streams. *Euro-Par 2015: Parallel Processing*, *Lecture Notes in Computer Science*, vol. 9233, Träff JL, Hunold S, Versaci F (eds.). Springer Berlin Heidelberg, 2015; 402–413.
16. Pripužić K, Žarko IP, Aberer K. Time- and space-efficient sliding window top-k query processing. *ACM Trans. Database Syst.* Mar 2015; **40**(1):1:1–1:44.
17. Sun S, Huang Z, Zhong H, Dai D, Liu H, Li J. Efficient monitoring of skyline queries over distributed data streams. *Knowl. Inf. Syst.* 2010; **25**(3):575–606.
18. Lu H, Zhou Y, Haustad J. Efficient and scalable continuous skyline monitoring in two-tier streaming settings. *Inf. Syst.* Mar 2013; **38**(1):68–81.
19. Im H, Park J, Park S. Parallel skyline computation on multicore architectures. *Inf. Syst.* Jun 2011; **36**(4):808–823.
20. Hose K, Vlachou A. A survey of skyline processing in highly distributed environments. *The VLDB Journal* Jun 2012; **21**(3):359–384.
21. Wang S, Ooi BC, Tung AKH, Xu L. Efficient skyline query processing on peer-to-peer networks. *ICDE*, Chirkova R, Dogac A, özsu MT, Sellis TK (eds.), IEEE, 2007; 1126–1135.
22. Wu P, Zhang C, Feng Y, Zhao BY, Divyakant A, Abbadi AE. Parallelizing skyline queries for scalable distribution. *EDBT*, *Lecture Notes in Computer Science*, vol. 3896, Springer, 2006; 112–130.
23. Woods L, Alonso G, Teubner J. Parallel computation of skyline queries. *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on* 2013; **0**:1–8.
24. Papadias D, Tao Y, Fu G, Seeger B. Progressive skyline computation in database systems. *ACM Trans. Database Syst.* Mar 2005; **30**(1):41–82.
25. Afrati FN, Koutris P, Suciu D, Ullman JD. Parallel skyline queries. *Proceedings of the 15th International Conference on Database Theory*, ICDT '12, ACM: New York, NY, USA, 2012; 274–284.
26. Cosgaya-Lozano A, Rau-Chaplin A, Zeh N. Parallel computation of skyline queries. *Proceedings of the 21st International Symposium on High Performance Computing Systems and Applications*, HPCS '07, IEEE Computer Society: Washington, DC, USA, 2007; 12–.
27. Vlachou A, Doulkeridis C, Kotidis Y. Angle-based space partitioning for efficient parallel skyline computation. *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, ACM: New York, NY, USA, 2008; 227–238.
28. The fastflow (ff) parallel programming framework 2015. `http://calvados.di.unipi.it/`.
29. Aldinucci M, Danelutto M, Kilpatrick P, Meneghin M, Torquati M. An efficient unbounded lock-free queue for multi-core systems. *Proceedings of the 18th International Conference on Parallel Processing*, Euro-Par'12, Springer-Verlag: Berlin, Heidelberg, 2012; 662–673.
30. Tucker PA, Maier D, Sheard T, Fegaras L. Exploiting punctuation semantics in continuous data streams. *IEEE Trans. on Knowl. and Data Eng.* Mar 2003; **15**(3):555–568.
31. Stonebraker M, Abadi DJ, Batkin A, Chen X, Cherniack M, Ferreira M, Lau E, Lin A, Madden S, O'Neil E, *et al.*. C-store: A column-oriented dbms. *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, VLDB Endowment, 2005; 553–564.
32. Theodoridis Y, Sellis T. A model for the prediction of r-tree performance. *Proceedings of the Fifteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '96, ACM: New York, NY, USA, 1996; 161–171.
33. Bertolli C, Mencagli G, Vanneschi M. Analyzing memory requirements for pervasive grid applications. *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, 2010; 297–301, doi:10.1109/PDP.2010.71.
34. Mencagli G, Vanneschi M, Vespa E. Control-theoretic adaptation strategies for autonomic reconfigurable parallel applications on cloud environments. *High Performance Computing and Simulation (HPCS), 2013 International Conference on*, 2013; 11–18, doi:10.1109/HPCSim.2013.6641387.