A Multicore Parallelization of Continuous Skyline Queries on Data Streams

Tiziano De Matteis, Salvatore Di Girolamo and Gabriele Mencagli

Department of Computer Science, University of Pisa
Largo B. Pontecorvo, 3, I-56127, Pisa, Italy
{dematteis, digirolamo, mencagli}@di.unipi.it

Abstract. Skyline queries are preference queries frequently used in multi-criteria decision making to retrieve interesting points from large datasets. They return the points whose attribute vector is not dominated by any other point. Over the last years, sequential and parallel implementations over static datasets have been proposed for multiprocessors and clusters. Recently, skyline queries have been computed over continuous data streams according to *sliding window* models. Although sequential algorithms have been proposed and analyzed in the past, few works targeting modern parallel architectures exist. This paper contributes to the literature by proposing a parallel implementation for window-based skylines targeting multicores. We describe our parallelization by focusing on the cooperation between parallel functionalities, optimizations of the reduce phase, and load-balancing strategies. Finally, we show experiments with different point distributions, arrival rates and window lengths.

Keywords: Continuous queries, Skyline queries, Sliding window, Parallel programming, Multicores.

1 Introduction

Skyline queries are a particular class of preference queries that compute the set of Pareto-optimal points from a given set. They have become commonplace in real-time applications working on input data on-the-fly, such as network monitoring, sensor networks, stock market trading and social media. Usually, data are available in the form of *continuous streams* [1], i.e. sequences, possibly of unlimited length, of points (tuples) received from heterogeneous sources.

Most of the existing research works have focused on centralized [2] or parallel solutions [3, 4] for traditional skyline queries over static datasets. Computing the skyline over data streams is more challenging [5]. Due to the unbounded stream length, the query is evaluated on substreams (*windows*) corresponding to equal-sized time intervals. Tuples enter the window at their arrival and expire after a fixed time interval called *window length* (denoted by T_w). In the literature a *lazy* algorithm and an *eager* variant [5] have been proposed to maintain window-based skylines with different features in terms of space and time efficiency.

Parallel implementations of continuous skyline queries raise critical issues: *i)* how to partition the window among a set of parallel Workers, and *ii)* how to keep the partitions evenly sized in response to new point arrivals, pruning activities, and the expiration of old points. Existing works have only partially studied these problems. In [6] the authors have described an approach in which the computational load is moved from a centralized server to a set of data sites that interact with the server to notify changes in their local skyline. In [7] the authors have presented a parallel approach in the domain of uncertain streams, in which the associativity of the skyline operator does not hold. Both the approaches do not take into account any pruning phase of obsolete points, which is crucial to reduce memory occupancy at the expense of a harder load balancing.

In this paper we propose a parallelization of continuous skyline queries on multicores. We describe our parallelization as a *MAP* pattern with an asynchronous *reduce*. We study optimizations related to the reduce phase, and we show the effect of different load-balancing strategies. Then, we study the performance of our parallelization with different point distributions, arrival rates and window lengths. The results show good performance which proves the efficiency of our implementation and the effectiveness of our load-balancing strategy.

This paper is organized as follows. Sect. 2 describes related works. Sect. 3 introduces some prerequisites and a description of the parallelized sequential algorithm. Sect. 4 shows our parallelization which will be evaluated on a multicore architecture in Sect. 5. Finally, Sect. 6 concludes this paper.

2 Related Work

Skyline queries have been originally designed for static datasets by focusing on index structures to cope with high dimensional data (B-Trees, R-Trees, R*-Trees) [2]. Existing parallel solutions [3] share the idea of partitioning the datasets into regions processed in parallel and finally merging the results thanks to the associativity of the skyline operator.

On data streams a first work [8] has addressed n-of-N skyline queries, i.e. the skyline is computed over a count-based window of the last n received tuples (N is an upper bound to the window size). Time-based windows have been firstly used for continuous skyline queries in [9], with the algorithm *LookOut*. This solution does not perform any pruning strategy yielding to high memory occupancy. More recently in the work [5] the authors have proposed the *lazy* and the *eager* sequential algorithms for computing the skyline over streams with a time-based sliding-window semantics. The former method delays most of the computational work until the expiration of a skyline point. The latter, instead, performs a pre-computation phase at each new point arrival in order to minimize memory consumption at the expense of a higher processing burden.

Most of the previous approaches are centralized. Works proposing parallel solutions are [6, 10]. In the first one a centralized server collaborates with intelligent data sites that notify the server of the changes affecting the global skyline. The role of data sites is to avoid sending useless data by reducing the bandwidth usage. A similar idea has been applied in [10] for continuous skylines over wireless sensor networks. Part of the logic is moved to the sensors that filter input data saving network bandwidth and energy. Parallel skyline queries over uncertain data streams (with imprecise stream elements) have been discussed in [7] by proposing a parallel implementation for multicores. This solution is highly dependent from the case of uncertain streams, with parallel servers maintaining skyline probabilities without performing pruning actions. This solution is different from our work, which focuses on time-based sliding-window skylines over certain data streams with the pruning of obsolete points.

3 Continuous Skyline Queries: Eager Algorithm

The goal of the skyline operator 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 of the skyline operator is a stream of *skyline updates* expressed in the format `(action, p, t)`, where *action* indicates whether point p enters (`ADD`) or exits (`DEL`) the skyline at the specified time t. The continuous skyline operator is characterized by three properties:

- *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;
- *Point expiration*: a skyline point x will be definitely removed from the system when it reaches its expiration time;
- *Pruning*: once a point x arrives, the older points dominated by it (*obsolete*) can be discarded since they will not be able to enter the skyline in the future.

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 (see Def. 1). Several implementations can be used. Common solutions are arrays and index structures for spatial searching such as R-trees and R*-trees [2, 5].

In this paper we study a parallelization of the *eager* algorithm [5]. This algorithm performs the pruning of obsolete points: only the points currently in the skyline and those points candidate to enter the skyline in the future are stored in $\mathcal{DB}$. This property comes at the expense of a larger computational effort w.r.t similar algorithms [6, 10, 7] such as the *lazy* variant which does not perform pruning. The eager algorithm is based on the following concept:

Definition 1. *The Skyline Influence Time of a point p ($\mathtt{SIT}_p$) is the expiring time of the youngest point $r \in \mathcal{S}$ dominating p (r is the critical dominator of p).*

The algorithm maintains an *event list* $\mathcal{EL}$. 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 correspondently. Then, 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 is 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,* t_r^{exp}*)* into $\mathcal{EL}$ where t_r^{exp} is the expiring time of r, i.e. $\texttt{SIT}_p = t_r^{exp}$. Otherwise, if r does not exist, p becomes a skyline point immediately and we add the event *expire(p,* t_p^{exp}*)* into the event list.

The algorithm processes the events by using an internal timer. When an event is triggered, 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,*t_p^{exp}*)* is added to $\mathcal{EL}$;
– in the case of an *expire* event the associated point (which is part of the skyline) is discarded from $\mathcal{DB}$ and its removal from the skyline is notified through a `DEL` update.

This behavior allows us to define an important property:

Lemma 1. *If a point p reaches its expiration time* t_p^{exp} *at that time the point is part of the skyline.*

Proof. By contradiction let suppose that a point r, which dominates p, exists and its expiration time is after the one of p. Since expiration times are defined as $t_r^{exp} = t_r^{arr} + T_w$ and $t_p^{exp} = t_p^{arr} + T_w$, this means that r has been received by the system after p ($t_r^{arr} > t_p^{arr}$). This is impossible because, in that case, being r younger than p and dominating it, p would have been pruned when r arrived. □

Therefore, for each non-obsolete point p in $\mathcal{DB}$ there are two possible situations. If a new point r dominating p is received before the timer reaches $\texttt{SIT}_p$, p is pruned and its skytime event cleared from $\mathcal{EL}$. Otherwise, when the internal timer reaches $\texttt{SIT}_p$ 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 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.

4 Parallelization Design

Our parallelization is based on the *data-parallel* paradigm. It is a composition of a *MAP* pattern with an asynchronous *reduce* phase. The data structures $\mathcal{DB}$

and $\mathcal{EL}$ are partitioned among a set of *Workers* interfaced with the input stream and the output stream through an *Emitter* and a *Collector* functionality.

The implementation targets shared-memory architectures such as modern multi/manycores. Emitter, Collector and Workers are implemented by standard `POSIX` threads. Threads cooperate by exchanging pointers to shared data structures through `push` and `pop` operations on shared queues. In our implementation we use the lock-free queues provided by the `FastFlow` library [11], which exhibit great performance on cache-coherent multi-core chips.

4.1 Implementation

In the following we describe in detail the functionalities of our implementation and their cooperation pattern. The parallelization is sketched in Fig. 1.

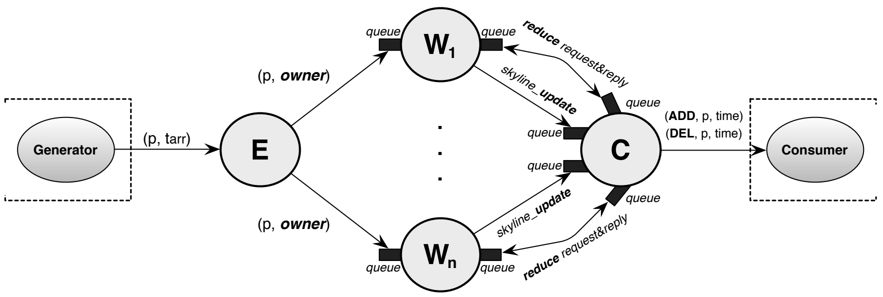


Fig. 1: Scheme of the parallel implementation: *Emitter* (`E`), *Worker* (`W`) and *Collector* (`C`) threads interacting through `push` and `pop` operations on `FastFlow` queues.

Emitter: the role of the Emitter is to interface the input stream (implemented by a TCP/IP socket) with the parallel computation. For each received point p the Emitter performs the following sequence of actions:

1. it assigns a timestamp t_p^{arr} according to the current system time;
2. it assigns the ownership of p to a specific Worker W_k, which will store p in its $\mathcal{DB}_k$ until the internal time reaches p's expiration time $t_p^{exp} = t_p^{arr} + T_w$;
3. p is multicasted to *all* the Workers. The Emitter performs a `push` operation on every input queue of the Workers. The message is the pair (p, k), where p is the tuple data structure and k is the index of the owner.

The role of the Emitter is critical for *load balancing*, i.e. to keep the size of the partitions of $\mathcal{DB}$ as similar as possible. To do that, the Emitter should implement clever *owner selection policies*. This aspect will be discussed in Sect. 4.2.

Workers: each Worker receives from the Emitter a stream of pairs (p, k). Any Worker W_i performs the following actions:

1. the pruning from its local partition $\mathcal{DB}_i$ all the points dominated by p; the private event list $\mathcal{EL}_i$ is cleared from the events related to the pruned points;

2. if a pruned point was in the skyline, a `DEL` update is generated;
3. W_i calculates the local SIT_p^i corresponding to the expiration time of the younger dominator of p in $\mathcal{DB}_i$. If p has no dominator in $\mathcal{DB}_i$ then SIT_p^i is zero. W_i sends the value of SIT_p^i to the Collector for the *reduce* phase;

After these actions all the Workers except the owner discard p, thus the owner has a very limited additional overhead. The reduce phase is necessary to determine the global value of $\texttt{SIT}_p$, which is the maximum between the local $\texttt{SIT}_p^i$ for all the partitions, i.e. $\texttt{SIT}_p = \max_{i=1}^{n}\{\texttt{SIT}_p^i\}$. Because of its fine-grained nature, the reduce is centralized in the Collector: `C` receives the values of $\texttt{SIT}_p^i$ from the Workers, calculates the global $\texttt{SIT}_p$ and sends it to the owner W_k that:

4. if $\texttt{SIT}_p = 0$ p must be added to the skyline immediately: an $expire(p, t_p^{exp})$ event is added to $\mathcal{EL}_k$ and an `ADD` update is transmitted to the Collector. Otherwise, a $skytime(p, \texttt{SIT}_p)$ event is added to $\mathcal{EL}_k$.

Workers are responsible for processing events in the correct order by producing skyline updates to the Collector. Each Worker has an internal notion of time that moves forward at each reception of a new point from the Emitter, i.e. when a point p is received, the current time is set equal to t_p^{arr}. Before starting the computation related to the received point, each Worker executes (unrolls) all the events in its $\mathcal{EL}_i$ with timestamp smaller than t_p^{arr}. `ADD` and `DEL` updates are transmitted to the Collector when *skytime* and *expire* events arise.

Collector: this thread receives two types of messages from the Workers:

– *reduce messages* with the local $\texttt{SIT}_p^i$ of a point p from Worker W_i. Once all the $\texttt{SIT}_p^i$ for $i = 1, \ldots, n$ have been received, the Collector computes $\max_{i=1}^{n}\{\texttt{SIT}_p^i\}$ and sends this value *only* to the owner of point p;
– *skyline updates* need to be buffered by the Collector in order to transmit them onto the output stream by respecting the chronological order. To do that, the Collector buffers the updates and keeps them ordered by timestamp using a priority queue. The Collector maintains the timestamp of the last received update from each Worker (denoted by $lst\text{-}t_i$). All the buffered updates with timestamp smaller or equal than $\min_{i=1}^{n}\{lst\text{-}t_i\}$ can be safely transmitted onto the output stream of the computation.

4.2 Optimizations

In this section we discuss two optimizations: *i)* we design an *asynchronous reduce* for the computation of the global `SIT` of each new point; *ii)* we study proper *owner selection policies* to balance the workload among Workers.

Asynchronous reduce: the value of the `SIT` of the last received point is the result of a reduce involving the Workers and the Collector threads. In the basic implementation the reduce is executed *synchronously.* The owner of the current point p cannot start the computation on the next point r until the reduce result is made available by the Collector.

According to the semantics of the eager algorithm the reduce can be performed *asynchronously.* Let the owner of the current point p be the Worker W_k. Instead of waiting the value SIT_p explicitly from the Collector, W_k can process subsequent points received from the Emitter while $\texttt{SIT}_p$ is not available yet. For each successive point r the Worker W_k:

- searches the youngest dominator of r in its $\mathcal{DB}_k$: this operation uses the expire time of the stored points and their spatial coordinates, thus it is independent from $\texttt{SIT}_p$;
- all the points $v \in \mathcal{DB}_k$ such that $r \prec v$ can be pruned. If p is one of the pruned points, the value of $\texttt{SIT}_p$ is no longer necessary.

In conclusion the asynchronous reduce works as follows:

1. when a new point is received by a Worker, whether it is the owner or not it participates in the reduce phase without waiting for the result;
2. each Worker waits for messages either from the Emitter (a new point) or from the Collector (reduce result);
3. when the reduce result is received from the Collector: *i)* if the point has been pruned the `SIT` is ignored; *ii)* otherwise, a new event (*skytime* or *expire*) is inserted into the event list of the owner according to the value of `SIT` (if it is equal or greater to zero, see Sect. 3).

This optimization leads to a significant improvement in the performance achieved by our implementation, as it will be shown in Sect. 5.

Owner selection policies: the ownership must be assigned in order to keep the partitions $\mathcal{DB}_1, \ldots, \mathcal{DB}_n$ evenly sized. This problem is particularly critical in continuous skyline queries, since the cardinality of the partitions can change significantly due to the variability of the arrival rate and the effect of the pruning.

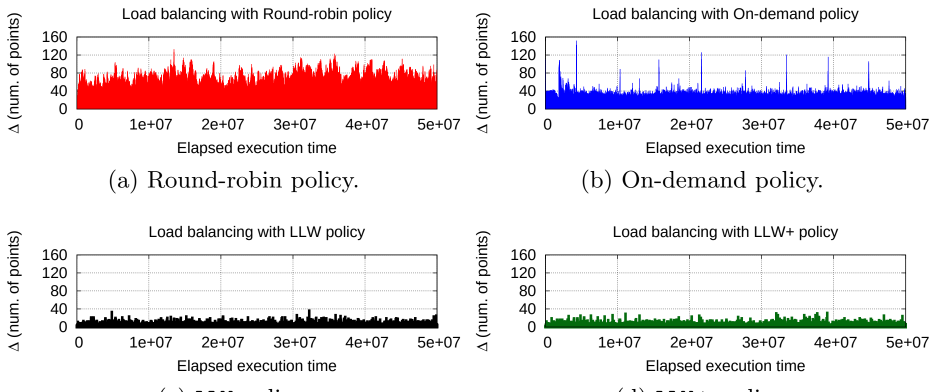
In the literature a similar problem has been studied for skyline queries over static datasets. Local skylines are computed for each partition and then merged to define the global skyline. The partitions are usually determined using the spatial coordinates of points as in the *grid-based* and *angle-based* schemes proposed in [4]. In our case such approaches are not sufficiently effective: *i)* in the case of points not uniformly distributed the partitions can have very different cardinalities; *ii)* many skyline points can fall in few partitions, thus in our parallelization some Workers might provide a very marginal contribution to the skyline definition. In this paper we apply owner selection policies independent from the spatial coordinates of points. We consider four heuristics:

- *Round robin* (`RR`): the ownership is interleaved among Workers, i.e. point x_j is assigned to Worker W_i such that $i = (j \mod n) + 1$;
- *On demand* (`OD`): the ownership of a new point is assigned to the first Worker able to accommodate it in its input queue;

– *Least Loaded Worker* (`LLW`): each new point is assigned to the Worker with the smaller partition of $\mathcal{DB}$;
– *Least Loaded Worker* with *Ownership* (`LLW+`): this policy is an extension of `LLW` in which, in addition to the size of the partitions, the Emitter takes into account for each Worker the number of enqueued points for which it has been designated as the owner.

For the last two policies the Emitter must know the size of the partitions and the number of enqueued points owned by each Worker. We use shared counters between the Emitter/Workers threads, implemented as `std::atomic<int>` of the standard `C++` library with atomic increment/decrement operations.

Fig. 2 shows a comparison on an Intel multicore composed of two Xeon Sandy Bridge E5-2650 CPUs for a total of 16 cores operating at 2 GHz with 32GB or RAM. Each core has private L1d (32KB) and L2 (256KB) caches. Each CPU has a shared L3 cache of 20MB. We use a configuration with 4.5K non-obsolete points distributed in 12 partitions (one per Worker). We measure the difference between the biggest and the smallest partition, i.e. $\Delta = |\mathcal{DB}^{max}| - |\mathcal{DB}^{min}|$. We use five double precision floating-point numbers per point ($d = 5$). Higher dimensionalities have minor effects on the results.



(a) Round-robin policy. (b) On-demand policy.

(c) `LLW` policy. (d) `LLW+` policy.

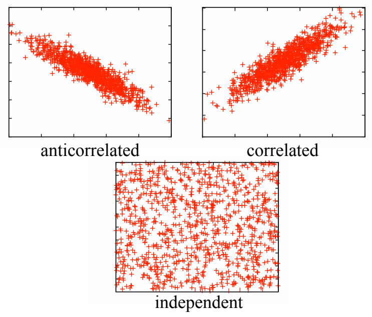
Fig. 2: Load balancing results: independent distribution. Average window size of 4.4K points. The same qualitative behavior is observed for the other point distributions.

The results show that the last two policies are able to produce partitions with very similar cardinalities over the execution. This is an expected result because the first two policies are independent from the actual load of the Workers. The best policy is `LLW+`. Numerically we have the following values ($\Delta^{avg} + \sigma^2$): `RR`:64.02 + 229, `OD`:34.13 + 1791, `LLW`:3.15 + 4.28, `LLW+`:2.55 + 4.05. As we can observe: *i)* load-aware policies are able to obtain smaller Δ^{avg} with a significantly lower variance; *ii)* by taking into account the number of owned enqueued points, the `LLW+` policy is able to achieve a 20% improvement than `LLW`.

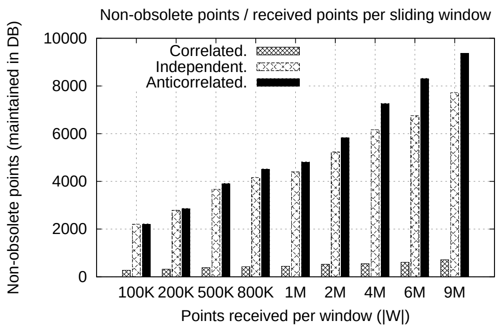
5 Experiments

In this section we study the performance of our parallelization on the Intel multicore. We use the `gcc` compiler version `4.8.1` with the −`O3` optimization flag. We set the affinity of each thread on a different core. The maximum number of Workers is 12 in our machine, since we have four threads for the Generator, Emitter, Collector and Consumer. Mapping two threads onto the same core (hyperthreading) is not beneficial due to the aggressive busy-waiting synchronization performed by `pop` and `push` operations on the `FastFlow` queues [11].

Data distribution and memory usage: the effect of the pruning depends on the spatial distribution of data. Analogously to existing works [5], we consider three point distributions: the *anticorrelated*, *correlated* and *independent* ones as shown in Fig. 3a for 2D points. In the correlated case a small set of points dominate the others and the pruning phase is very intensive. The anticorrelated case in on the opposite, with a large number of points that are part of the skyline set. The third one is an intermediate case with points uniformly distributed in the space. Fig. 3b shows the number of points maintained in $\mathcal{DB}$ with respect to



(a) Point distributions.



(b) Effect of pruning.

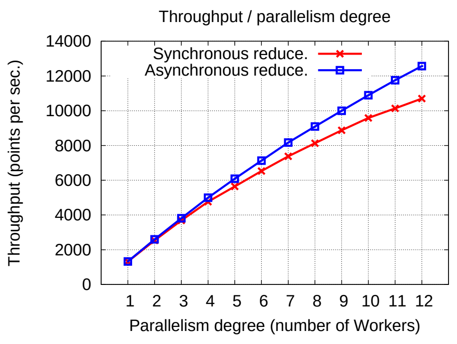
Fig. 3: Space distribution of points (anticorrelated, correlated and independent) and effect of the pruning on the number of stored points.

the total number of points received per sliding window (denoted by $|\mathcal{W}|$). This number is given by the product between the arrival rate of the input stream and the window length, i.e. $|\mathcal{W}| = \lambda \times T_w$. The number of non-obsolete points is at least three orders of magnitude smaller than the number of received points. The pruning phase increases the processing time per tuple (all the dominated points must be identified and removed) but it greatly saves memory occupancy. In the worst case of the anticorrelated distribution, with 9M points received per window we need to maintain only $\sim$ 9K non-obsolete points in $\mathcal{DB}$ (with four doubles per point we need $\sim$ 280KB instead of $\sim$ 275MB).

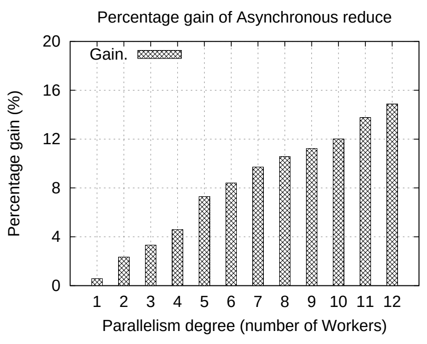
These results have an important implication on the implementation design. Only a little portion of the received points needs to be stored by the algorithm.

Furthermore, since our parallelization is a data-parallel solution, the set of non-obsolete points is partitioned among n Workers further decreasing the size of each $\mathcal{DB}_i$. According to [5,7], many existing applications of continuous skyline queries (e.g. analysis of social media such as Twitter, Facebook and so on) are executed with window lengths of few tens of seconds and arrival rates of several thousands of points/sec, leading to a total number of points per sliding window in the order of few millions of tuples. With these sizes, the $\{\mathcal{DB}_i\}_{i=1}^{n}$ data structures are implemented by *dynamic arrays* (usually one per dimension, to increase data locality in the cache hierarchy of multicores) without relying on additional index structures (e.g.R-tree and R*-trees) that are beneficial with larger datasets.

Effect of asynchronous reduce: we measure the benefit of the asynchronous reduce on throughput. Throughput is the average number of points processed per second. With different configurations in terms of arrival rate and window length, we achieve an average gain between $(10 \div 20)\%$. Fig. 4 shows a scenario with an arrival rate of $100K$ points/sec generated according to the independent distribution with a window length $T_w = 20$ seconds. The average gain is of $\sim$ 10%



(a) Throughput.

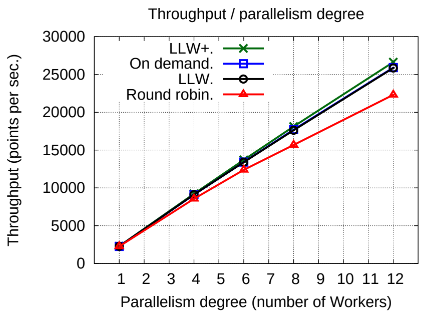


(b) Percentage gain.

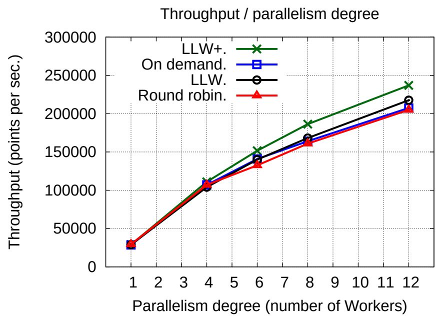
Fig. 4: Comparison between synchronous and asynchronous reduce. Scenario: $\lambda = 100$K points/sec, $T_w = 20$ sec, independent distribution and `LLW+` owner selection policy.

with a peak of $\sim$ 15% with 12 Workers. As we can observe from Fig. 4b, the gain increases with the parallelism degree. The reason is that even the `LLW+` policy is not able to produce perfectly equal partitions. With high parallelism degrees the partitions of $\mathcal{DB}$ are smaller and a slight difference in their cardinality has a negative effect (higher in percentage) on throughput. The asynchronous reduce is able to mitigate this slight load unbalance among Workers, by achieving better throughput compared with the synchronous reduce implementation.

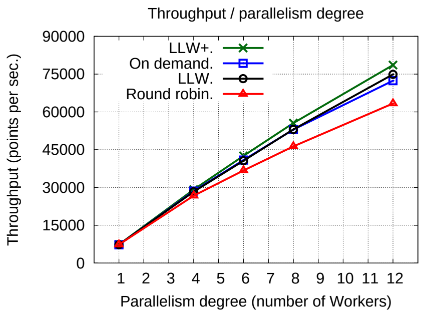
Throughput and scalability: we show the throughput and the best scalability achieved by our parallelization. For each distribution we use a different scenario in terms of arrival rate and window length. The results are shown in Fig. 5. For all the scenarios the maximum number of cores (12) is not sufficient to achieve the maximum throughput (equal to the arrival rate of the stream), hence we are able to study how the throughput increases up to the maximum number of physical cores of our machine. Tab. 5d shows: the best throughput achieved with the



(a) **`Anticorrelated`**: $T_w = 10$ sec, $\lambda =$ 80K points/sec.



(b) **`Correlated`**: $T_w = 60$ sec, $\lambda = 250$K points/sec.



(c) **`Independent`**: $T_w = 10$ sec, $\lambda =$ 100K points/sec.

	`Anticorr.`	`Corr.`	`Indep.`
$\mathcal{B}^{(12)}$	28K p/s	237K p/s	78K p/s
$\mathcal{S}^{(12)}$	11.65	8.16	10.7
$\lvert\mathcal{DB}\rvert$	4,598	1,192	4,226
$\lvert\mathcal{W}\rvert$	800K	15M	1M
$\mathcal{P}$	0.994	0.999	0.996

(d) Summary of the numerical results (p/s = points /sec).

Fig. 5: Throughput achieved with different point distributions.

highest parallelism degree ($\mathcal{B}^{(12)}$), the best scalability ($\mathcal{S}^{(12)}$) measured as the ratio between the throughput with 12 Workers and the one with just one single Worker thread, the number of non-obsolete points ($|\mathcal{DB}|$), the total number of points received per sliding window (($|\mathcal{W}|$), and the pruning probability $\mathcal{P}$. The best throughput and scalability results are reported only for the `LLW+` policy. With the correlated distribution we achieve the lowest scalability. Although this scenario is characterized by the highest value of the arrival rate and window length, the number of non-obsolete points is small due to a very high pruning probability. In this case the computation is very fine grained and a slight difference in the cardinalities of the partitions (also of few units) prevents to achieve a near-optimal scalability also with the `LLW+` policy and the asynchronous reduce. Near optimal results are achieved with the other two distributions.

6 Conclusions

This paper presented a *map-reduce* parallelization of the skyline operator on data streams, optimized with an asynchronous reduce phase and smart load balancing strategies. The experiments confirmed that the `LLW+` policy is the best one, and near-optimal scalability can be achieved with the anticorrelated and independent distributions. The correlated case is the most challenging due to the very fine-grained nature of the computation, and deserves to be further investigated in the future with possible run-time mechanisms enabling dynamic adaptiveness to sustain highly variable input rates [12].

References

1. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems. PODS '02, New York, NY, USA, ACM (2002) 1–16
2. Börzsönyi, S., Kossmann, D., Stocker, K.: The skyline operator. In: Proceedings of the 17th International Conference on Data Engineering, Washington, DC, USA, IEEE Computer Society (2001) 421–430
3. Im, H., Park, J., Park, S.: Parallel skyline computation on multicore architectures. Inf. Syst. **36** (2011) 808–823
4. Vlachou, A., Doulkeridis, C., Kotidis, Y.: Angle-based space partitioning for efficient parallel skyline computation. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data. SIGMOD '08, New York, NY, USA, ACM (2008) 227–238
5. Tao, Y., Papadias, D.: Maintaining sliding window skylines on data streams. Knowledge and Data Eng., IEEE Transactions on **18** (2006) 377–391
6. Lu, H., Zhou, Y., Haustad, J.: Efficient and scalable continuous skyline monitoring in two-tier streaming settings. Inf. Syst. **38** (2013) 68–81
7. 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. **41** (2014) 277–309
8. Lin, X., Yuan, Y., Wang, W., Lu, H.: Stabbing the sky: efficient skyline computation over sliding windows. In: Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on. (2005) 502–513
9. Morse, M., Patel, J., Grosky, W.I.: Efficient continuous skyline computation. In: Data Engineering, 2006. ICDE '06. Proceedings of the 22nd International Conference on. (2006) 108–108
10. Xin, J., Wang, G., Chen, L., Zhang, X., Wang, Z.: Continuously maintaining sliding window skylines in a sensor network. DASFAA'07, Berlin, Heidelberg, Springer-Verlag (2007) 509–521
11. Aldinucci, M., Danelutto, M., Kilpatrick, P., Meneghin, M., Torquati, M.: An efficient unbounded lock-free queue for multi-core systems. In: Proceedings of the 18th International Conference on Parallel Processing. Euro-Par'12, Berlin, Heidelberg, Springer-Verlag (2012) 662–673
12. Mencagli, G., Vanneschi, M.: Qos-control of structured parallel computations: A predictive control approach. 2013 IEEE 5th International Conference on Cloud Computing Technology and Science **0** (2011) 296–303