High Parallel Skyline Computation over Low-Cardinality Domains

Markus Endres and Werner Kießling

Department of Computer Science, University of Augsburg,
86135 Augsburg, Germany
{endres,kiessling}@informatik.uni-augsburg.de
http://www.informatik.uni-augsburg.de/dbis

Abstract. A Skyline query retrieves all objects in a dataset that are not dominated by other objects according to some given criteria. Although there are a few parallel Skyline algorithms on multicore processors, it is still a challenging task to fully exploit the advantages of such modern hardware architectures for efficient Skyline computation. In this paper we present high-performance parallel Skyline algorithms based on the lattice structure generated by a Skyline query. We compare our methods with the state-of-the-art algorithms for multicore Skyline processing. Experimental results on synthetic and real datasets show that our new algorithms outperform state-of-the-art multicore Skyline techniques for low-cardinality domains. Our algorithms have linear runtime complexity and fully play on modern hardware architectures.

Keywords: Skyline, Parallelization, Multicore.

1 Introduction

The Skyline operator [1] has emerged as an important and very popular summarization technique for multi-dimensional datasets. A Skyline query selects those objects from a dataset D that are not dominated by any others. An object p having d attributes (dimensions) dominates an object q, if p is better than q in at least one dimension and not worse than q in all other dimensions, for a defined comparison function. This dominance criteria defines a partial order and therefore transitivity holds. The *Skyline* is the set of points which are not dominated by any other point of D. Without loss of generality, we consider the Skyline with the *min* function for all attributes.

Most of the previous work on Skyline computation has focused on the development of efficient sequential algorithms [2]. However, the datasets to be processed in real-world applications are of considerable size, i.e., there is the need for improved query performance, and parallel computing is a natural choice to achieve this performance improvement, since multicore processors are going mainstream [3]. This is due to the fact that Moore's law of doubling the density of transistors on a CPU every two years – and hence also doubling algorithm's performance – may come to an end in the next decade due to thermal problems. Thus, the chip

Y. Manolopoulos et al. (Eds.): ADBIS 2014, LNCS 8716, pp. 97–111, 2014.

manufactures tend to integrate multiple cores into a single processor instead of increasing the clock frequency. In upcoming years, we will see processors with more than 100 cores, but not with much higher clock rates. However, since most applications are build on using sequential algorithms, software developers must rethink their algorithms to take full advantage of modern multicore CPUs [3]. The potential of parallel computing is best described by Amdahl's law [4]: the speedup of any algorithm using multiple processors is strictly limited by the time needed to run its sequential fraction. Thus, only high parallel algorithms can benefit from modern multicore processors.

Typically an efficient Skyline computation depends heavily on the number of comparisons between tuples, called *dominance tests*. Since a large number of dominance tests can often be performed independently, Skyline computation has a good potential to exploit multicore architectures as described in [5–7]. In this paper we present algorithms for high-performance parallel Skyline computation which do *not* depend on tuple comparisons, but on the *lattice structure* constructed by a Skyline query over *low-cardinality domains*. Following [8, 2] many Skyline applications involve domains with small cardinalities – these cardinalities are either inherently small (such as star ratings for hotels), or can naturally be mapped to low-cardinality domains (such as price ranges on hotels).

The remainder of this paper is organized as follows: In Section 2 we discuss some related work. In Section 3 we revisit the Hexagon algorithm [9], since it is the basic idea behind our parallel algorithms. Based on this background we will present our parallel Skyline algorithms in Section 4. We conduct an extensive performance evaluation on synthetic and real datasets in Section 5. Section 6 contains our concluding remarks.

2 Related Work

Algorithms of the *block-nested-loop* class (BNL) [1] are the most prominent algorithms for computing Skylines. In fact the basic operation of collecting maxima during a single scan of the input data can be found at the core of several Skyline algorithms, cp. [10, 2]. Another class of Skyline algorithms is based on a straightforward *divide-and-conquer* (D&C) strategy. D&C uses a recursive split-and-merge scheme, which is definitely applicable in parallel scenarios [11].

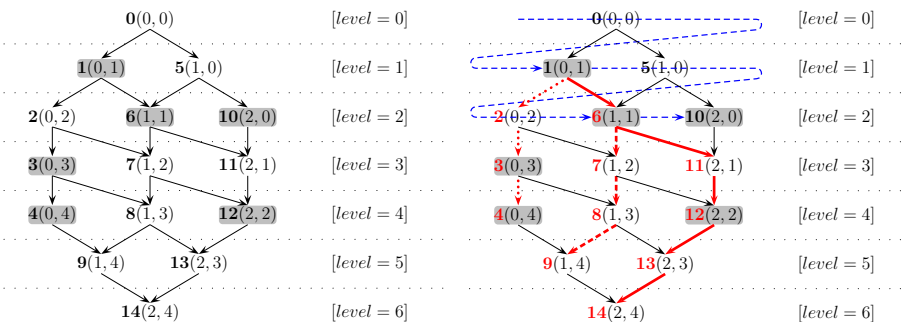
There is also a growing interest in distributed Skyline computation, e.g., [12–16], where data is partitioned and distributed over net databases. Also there are several approaches based on the MapReduce framework, e.g., [17]. All approaches have in common that they share the idea of partitioning the input data for parallel shared-nothing architectures communicating only by exchanging messages. The nodes locally process the partitions in parallel, and finally merge the local Skylines. The main difference of such a parallel Skyline computation resides in the partitioning schemes of the data. The most used partitioning scheme is grid-based partitioning [14]. Recent work [18] focus on an angle-based space partitioning scheme using hyperspherical coordinates of the data points. In [19], the authors partition the space using hyperplane projections to obtain useful partitions of the dataset for parallel processing.

Im et al. [6] focuses on exploiting properties specific to multicore architectures in which participating cores inside a processor share everything and communicate simply by updating the main memory. They propose a parallel Skyline algorithm called *pSkyline*. pSkyline divides the dataset linearly into N equal sized partitions. The local Skyline is then computed for each partition in parallel using sSkyline [6]. Afterwards the local Skyline results have to be merged. Liknes et al. [7] present the *APSkyline* algorithm for efficient multicore computation of Skyline sets. They focus on the partitioning of the data and use the angle-based partitioning from [18] to reduce the number of candidate points that need to be checked in the final merging phase. The authors of [5] modified the well-known BNL algorithm to develop parallel variants based on a shared linked list for the Skyline window. In their evaluation, the lazy locking scheme [20] is shown to be most efficient in comparison to continuous locking or lock-free synchronization. There is also recent work on computing Skylines using specialized parallel hardware, e.g., GPU [21] and FPGA [22]. In contrast to previous works, our approach is based on the parallel traversal of the lattice structure of a Skyline query.

3 Skyline Computation Using the Lattice Revisited

Our parallel algorithms are based on the algorithms *Hexagon* [9] and *LS-B* [8], which follow the same idea: the partial order imposed by a Skyline query over a low-cardinality domain constitutes a *lattice*. This means if $a, b \in D$, the set $\{a, b\}$ has a least upper bound and a greatest lower bound in D. Visualization of such lattices is often done using *Better-Than-Graphs* (*BTG*) (Hasse diagrams), graphs in which edges state dominance. The nodes in the BTG represent *equivalence classes*. Each equivalence class contains the objects mapped to the same feature vector. All values in the same class are considered substitutable.

An example of a BTG over a 2-dimensional space is shown in Figure 1a. We write $[2, 4]$ to describe a two-dimensional domain where the first attribute A_1 is an element of {0,1,2} and attribute A_2 an element of {0,1,2,3,4}. The arrows show the dominance relationship between elements of the lattice.



(a) 2d Skyline over [2,4] (b) BFT and DFT in Hexagon

Fig. 1. The Hexagon algorithm revisited [9]

The node $(0,0)$ presents the *best node*, i.e., the least upper bound for two arbitrary nodes a and b in the lattice. The node $(2,4)$ is the *worst node* and serves as the greatest lower bound. The bold numbers next to each node are *unique identifiers* (ID) for each node in the lattice, cp. [9]. Nodes having the same level are incomparable. That means for example, that neither the objects in the node $(0,4)$ are better than the objects in $(2,2)$ nor vice versa. They have the same overall *level 4*. A dataset D does not necessarily contain representatives for each lattice node. In Figure 1a the gray nodes are occupied (*non-empty*) with real elements from the dataset whereas the white nodes have no element (*empty*).

The method to obtain the Skyline can be visualized using the BTG. The elements of the dataset D that compose the Skyline are those in the BTG that have *no path leading to them from another non-empty node in D*. In Figure 1a these are the nodes $(0,1)$ and $(2,0)$. All other nodes have direct or transitive edges from these both nodes, and therefore are *dominated*. The algorithms in [9, 8] exploit these observations and in general consist of three phases:

1) **Phase 1:** The *Construction Phase* initializes the data structures. The lattice is represented by an *array* in main memory with the size of the lattice, i.e., the number of nodes. Each position in the array stands for one node ID in the lattice. Initially, all nodes of the lattice are marked as *empty*.
2) **Phase 2:** In the *Adding Phase* the algorithm iterates through each element t of the dataset D. For each element t the unique ID and the node of the lattice that corresponds to t is determined. This node is marked *non-empty*.
3) **Phase 3:** After all tuples have been processed, the nodes of the lattice that are marked as *non-empty* and which are not reachable by the transitive dominance relationship from any other *non-empty* node of the lattice represent the Skyline values. Nodes that are *non-empty* but are reachable by the dominance relationship, and hence are not Skyline values, are marked *dominated* to distinguish them from present Skyline values.

   From an algorithmic point of view this is done by a combination of *breadth-first traversal* (BFT) and *depth-first traversal* (DFT). The nodes of the lattice are visited level-by-level in a breadth-first order (the blue dashed line in Figure 1b). When an empty node is reached, it is removed from the BFT relation. Each time a *non-empty* and *not dominated* node is found, a DFT is done marking all dominated nodes as *dominated*. For example, the node $(0,1)$ in Figure 1b is not empty. The DFT walks down to the nodes $(1,1)$ and $(0,2)$. Which one will be visited first is controlled by a so called *edge weight*, cp. [9]. Here, $(1,1)$ will be marked as *dominated* and the DFT will continue with $(2,1)$, etc. (the red solid arrows in Figure 1b). If the DFT reaches the bottom node $(2,4)$ (or an already dominated node) it will recursively follow the other edge weights, i.e. the red dashed arrows, and afterwards the red dotted arrows. Afterwards the BFT will continue with node $(1,0)$, which will be removed because it is empty. The next non-empty node is $(1,1)$, which is already dominated and therefore we will continue with $(2,0)$. Since all other nodes are marked as dominated, the algorithm will stop and the remaining nodes $(0,1)$ and $(2,0)$ present the Skyline.

4 Parallel Skyline Algorithms

In this section we describe our parallel algorithms, the used data structures, discuss some implementation issues, and have a look at the complexity and memory requirements of our algorithms.

4.1 Parallel Skyline Computation

For the development of our parallel Skyline algorithms we combine a *split* approach of the input dataset with a *shared data structure* supporting fine grained locking and apply them to the Hexagon algorithm described in Section 3.

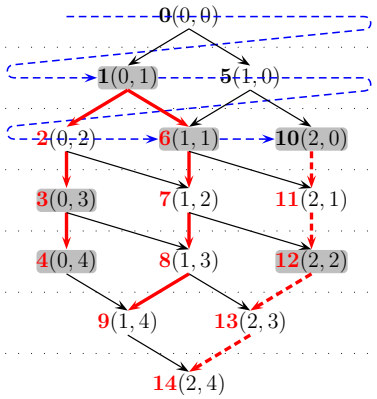
The general idea of parallelizing the Hexagon algorithm is to parallelize the adding phase (Phase 2) and the removal phase (Phase 3). Phase 1 is not worth to parallelize because of its simple structure and minor time and effort for the initialization. Parallelizing Phase 2 can be done using a simple partitioning approach of the input dataset, whereas for Phase 3 two different approaches can be used: In the first variant the parallel Phase 3 starts *after* all elements were added to the BTG. We call this algorithm **ARL-Skyline** (Adding-Removal-Lattice-Skyline). The second approach runs the adding and removal simultaneously. This algorithm is called **HPL-Skyline** (High-Parallel-Lattice-Skyline).

The ARL-Skyline Algorithm (ARL-S) is designed as follows:

- **Phase 1:** Initialize all data structures.
- **Phase 2:** Split the input dataset into c partitions, where c is the number of used threads. For each partition a worker thread iterates through the partition, determines the IDs for the elements and marks the corresponding entries in the BTG as *non-empty*.
- **Phase 3:** After adding *all* elements to the BTG a breadth-first walk beginning at the top starts (blue line in Figure 2a). For each *non-empty* and *not dominated* node run *tasks*[1] for the depth-first walk with the dominance test. In parallel continue with the breadth-first walk.

  For example, if the node $(0, 1)$ is reached in Figure 2a, two further tasks can be started in parallel to run a DFT down to $(0, 2)$ and $(1, 1)$ (red solid arrows). Continuing with the BFT we reach the already dominated node $(1, 1)$ and afterwards $(2, 0)$. A new DFT task follows the red dashed arrows to mark nodes as dominated. Note that the BFT task might be slower or faster than the DFT from node $(0, 1)$ and therefore the DFT could follow different paths in the depth-first dominance search. The pseudocode for ARL-S reduced to its essence is decpicted in Figure 2b; the fork/join task for the DFT can be found in Figure 3b.

[1] We use the `ForkJoinPool` from Java 7 to manage the recursive DFT tasks.



(a) BTG for ARL-S

```
1:  ARL-S A =
2:    // Phase 1, initialize data structures
3:    BTG := array with nodes based on lazy locking
4:    // Phase 2, Parallel Adding Phase
5:    split D into c disjoint sets D_i
6:    parallel for each D_i do
7:      for each t in D_i do
8:            ID := compute ID for t
9:            BTG[ID].setNonEmpty()
10:     end for each
11:   end parallel for
12:   // Phase 3, Parallel Removal Phase
13:   do a BFT beginning at the top
14:   for each nodeID in BFT do
15:     if ¬BTG[nodeID].isDominated()
16:           && ¬BTG[nodeID].isEmpty()
17:       fork/join task for DFT(nodeID)
18:     end if
19:   end for
```

(b) Pseudocode

Fig. 2. The ARL-Skyline algorithm

The HPL-S algorithm combines Phase 2 and 3 of ARL-S to *one* phase.

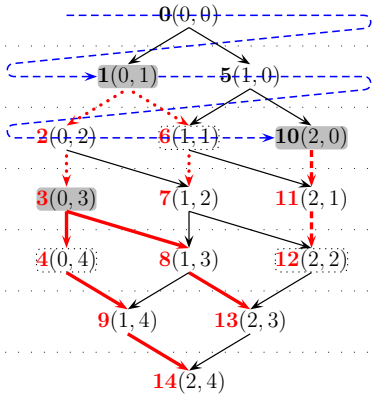
The HPL-Skyline Algorithm (HPL-S) is designed as:

- **Phase 1:** Initialize all data structures.
- **Phase 2+3:** Similar to Phase 2 in ARL-S we split the dataset into c partitions, for each partition a worker thread c_i. If one of the worker threads marks a node in the BTG as *non-empty*, it immediately starts a task for the DFT dominance test (if not done yet) and continues with adding elements, cp. Figure 3a. The simplified pseudocode is shown in Figure 3b.

  For example, thread c_1 adds an element to $(0, 3)$ and immediately starts additional tasks for the DFT (red arrows). Simultaneously another thread c_2 adds an element to the node $(0, 1)$ and starts tasks for the DFT and dominance tests (red dotted arrows). After thread c_1 has finished, it wants to add an element to $(1, 1)$. However, since it is already marked as dominated, thread c_1 can continue with adding elements to other nodes in the BTG without performing a DFT dominance test.

  After all threads have finished, a breadth-first traversal is done on the remaining nodes (blue line in Figure 3a). Again, the non-empty and not dominated nodes present the Skyline.

The advantage of the HPL-S in comparison to ARL-S is that the DFT search will mark dominated nodes as *dominated* and other parallel running threads do not have to add possible elements to these already dominated nodes. This saves memory and runtime.



(a) BTG for HPL-S

```
1:  HPL-S A =
2:    // Phase 1, initialize data structures
3:    BTG := array with nodes based on lazy locking
4:    // Phase 2+3, Parallel Adding and Removal
5:    split D into c disjoint sets D_i
6:    parallel for each D_i do
7:      for each t in D_i do
8:          ID := compute ID for t
9:          atomic if ¬BTG[ID].isDominated()
10:           BTG[ID].setNonEmpty()
11:           fork/join task for DFT(ID)
12:         end atomic if
13:     end for
14:   end parallel for
```

```
1:  DFT(ID) =
2:    parallel for each successor sID of ID do
3:      if BTG[sID].isDominated()
4:        return
5:      end if
6:      BTG[sID].setDominated()
7:      fork/join task for DFT(sID)
8:    end parallel for
```

(b) Pseudocode

Fig. 3. The HPL-Skyline algorithm

4.2 Data Partitioning and Choosing the Right Data Structure

Data Partitioning. The performance of known parallel and distributed BNL and D&C style algorithms (and many variants) are heavily influenced by the underlying partitioning of the input dataset. [13] suggests a grid-based partitioning, [18, 7] uses an angle-based partitioning, and [19] uses hyperplane projections to divide the dataset into disjoint sets. The *lattice algorithms* are independent from the partitioning, because the dominance tests are done on the lattice structure instead of relying on a tuple-to-tuple comparison. This is also the reason why the underlying data distribution (i.e., whether the dataset attributes are correlated, independent, or anti-correlated) does not influence performance.

Choosing the Right Data Structure. In general concurrency on a shared data structure requires a fine grained locking amongst all running threads to avoid unnecessary locks. In addition, one has to ensure that no data is read or written which has just been accessed by another thread (dirty reads or writes) in order to avoid data inconsistency. When considering for example the parallel removal phase in HPL-S (Figure 3b), a critical situation may occur if two threads try to append (line 10 in HPL-S) or delete an element (line 6 in DFT) on the same node simultaneously. This problem can be tackled by synchronization and locking protocols, cp. [5]. The *lazy locking* approach uses as few locks as possible. Locks are only acquired when they are really needed, i.e., when modifying nodes. Reading can be done in parallel without inconsistency problems. From a performance point of view lazy locking is definitely superior to all other locking protocols like continuous locking, full, or lock-free synchronization.

For the lattice implementation we used three different data structures: **Arrays**, **HashMaps**, and **SkipLists** [23]. Using an array means that each index in the array represents an ID in the lattice. The entries of the array are *nodes* holding the different states *empty*, *non-empty*, and *dominated*. Each *node* follows the lazy locking synchronization[2]. For the HashMap and SkipList implementation[3] we used the approach of a level-based storage, cp. Figure 4. An array models the *levels* of the BTG. Then the *nodes* are stored in a HashMap or SkipList. Adding an element to the BTG means computing the ID and the level it belongs to and marking the node at the right position as *non-empty* or *dominated*. The advantage of the level-based storage using SkipLists in contrast to HashMaps lies in the reduced memory requirements, because we do not have to initialize the whole data structure in main memory. A node is initialized on-the-fly if it is marked as *non-empty* or *dominated*. Additionally, if each node in a level is dominated, we can remove all nodes from the corresponding SkipList, mark the level-entry in the array as *dominated* and free memory.

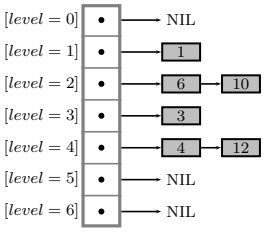


Fig. 4. Level-based storage of the BTG using SkipLists

In [20, 5] a LazyList with some advantages against the concurrent SkipList implementation was proposed to use for concurrent programming. Nevertheless, we decided to use SkipLists instead of LazyLists, because the traversal of a SkipList is faster than that of a LazyList due to the additional pointers which skip some irrelevant elements. Since not all nodes in the lattice are present and we have to find some nodes in the lattice during the DFT search quickly, the concurrent SkipList is the better choice.

4.3 Complexity Analysis and Memory Requirements

Complexity Analysis. The original lattice based algorithms [9, 8] have linear runtime complexity. More precisely, the complexity is $\mathcal{O}(dV + dn)$, where d is the dimensionality, n is the number of input tuples, and V is the product of the cardinalities of the d low-cardinality domains from which the attributes are drawn. Since there are V total entries in the lattice, each compared with at most d entries, this step is $\mathcal{O}(dV)$, cp. [8]. In the original version of Hexagon all entries in the lattice are positioned in an array. Since array accesses are $\mathcal{O}(1)$, the pass through the data to mark an entry as *non-empty* is $\mathcal{O}(dn)$.

[2] Implemented with `ReentrantReadWriteLock` in Java 7.
[3] We use `ConcurrentHashMap` and `ConcurrentSkipListMap` from Java 7.

The ARL-S and HPL-S algorithms with an *array* as BTG representation follow the original implementation of [9, 8] and therefore have a complexity of $\mathcal{O}(dV + dn)$. Using a level-based representation of the BTG with a HashMap for each level, we have a constant access for each level and $\mathcal{O}(1)$ for the look-up in the HashMap, since we can use a perfect hash function due the known width of the BTG in each level, cp. [24]. In summary this leads to $\mathcal{O}(dV + dn)$, too. For the SkipList based BTG implementation we have $\mathcal{O}(dV + dn \log w)$, since operations on SkipLists are $\mathcal{O}(\log w)$ [23], where w is the number of elements in the SkipList, i.e., the width of the BTG in the worst case.

Memory Requirements. Given a discrete low-cardinality domain $dom(A_1) \times \ldots \times \mathrm{dom}(A_m)$ on attributes A_i, the number of nodes in the BTG is given by $\prod_{i=1}^{m}(\max(A_i) + 1)$ [9]. Each node of the BTG has one of three different states: *empty*, *non-empty*, and *dominated*. The easiest way to encode these three states is by using two bits with 0x00 standing for *empty*, 0x01 for *non-empty* and 0x10 for *dominated*. This enables us to use the extremely fast bit functions to check and change node states. Since one byte can hold four nodes using two bits each, we have in summary that the BTG for a Skyline query may require the following maximal amount of memory, i.e, it is linear w.r.t. the size of the BTG.

$$mem(BTG) := \left\lceil \frac{1}{4} \prod_{i=1}^{m} (\max(A_i) + 1) \right\rceil$$

4.4 Remarks

Concurrent programming usually increases performance when the number of used threads is equal or less than the number of available processor cores and idling of threads can be prevented. Otherwise it can decrease performance due to waiting or mutual locking program codes. Our algorithms use high parallelism to complete the running tasks. This might be a performance problem for very small BTGs, if many threads work on a lattice where the size is much smaller than the number of threads. In this case there could be a lot of synchronization necessary. However, in practical Skyline problems this should not occur.

Another question concerns the speedup of concurrently programmed algorithms with larger input data. A good description of potentially benefits is given by Gustafson's Law [4], which says that computations involving arbitrarily large datasets can be efficiently parallelized. Our algorithms depend on the lattice size and the dominance tests on the lattice nodes, but not on a tuple-to-tuple comparison. Therefore, for larger datasets only the adding phase influences the performance, but not the removal phase, because the BTG size is independent from the input size. In addition, the HPL-S algorithms have the advantage of an "premature domination" of nodes, i.e., we filter out unnecessary elements early.

Due to Amdahl's law the sequential part of concurrent programs must be reduced to a minimum. In our algorithms only the initialization of the data structure (Phase 1), and the last tuple scanning is sequential, because initializing an array, a SkipList or HashMap is just instantiating these objects.

The reader will notice that the lattice based algorithms require two scans of the dataset to output the Skyline, the first to mark positions in the lattice structure and a second to output Skyline elements from values derived from the lattice. Another approach used in [9] is to mark the nodes in the lattice as *non-empty* and additionally hold pointers to the elements in the dataset. Obviously, this requires more memory, but avoids the second linear scan of the dataset.

In summary that means that we have high parallel algorithms with a minimal sequential part and therefore expect an enormous speed-up in Skyline evaluation.

5 Experiments

This section provides our comprehensive benchmarks on synthetic and real data to reveal the performance of the outlined algorithms. Due to the restricted space of the paper we only present some selected characteristic tests. However, all results show the same trends as those presented here.

5.1 Benchmark Framework

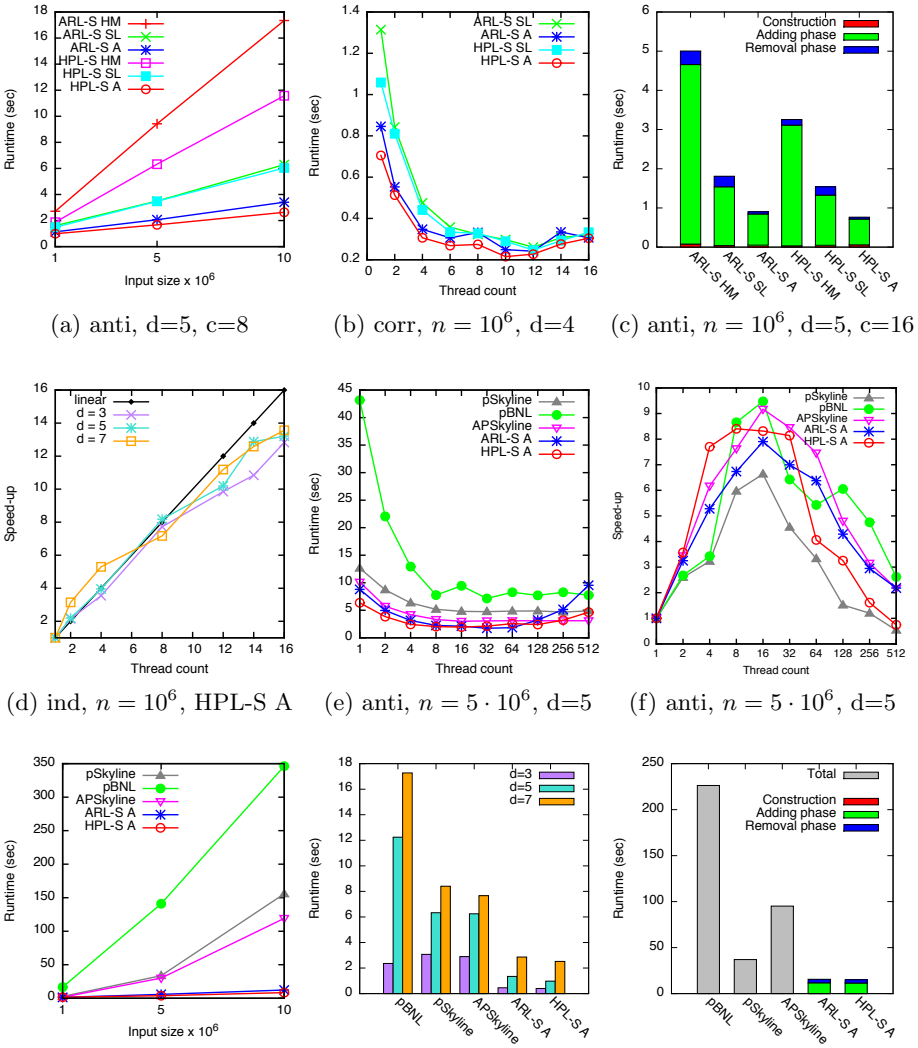
For our synthetic datasets we used the data generator commonly used in Skyline research [1]. We generated *anti-correlated* (anti), *correlated* (corr), and *independent* (ind) distributions and varied three parameters: (1) the data cardinality n, (2) the data dimensionality d, and (3) the number of distinct values for each attribute domain. For real data we used the entries from *www.zillow.com*. This dataset contains more than 2M entries about real estate in the United States. Each entry includes number of *bedrooms* and *bathrooms*, *living area* in sqm, and *age* of the building. The Zillow dataset also serves as a real-world application which requires finding the Skyline on data with a low-cardinality domain.

Our algorithms have been implemented using Java 7 using only built-in techniques for locking, compare-and-swap operations, and thread management. All experiments are performed on a single node running Debian Linux 7.1. The machine is equipped with two Intel Xeon 2.53 GHz quad-core processors using Hyper-Threading, that means a total of 16 cores.

5.2 Experimental Results

Comparison of ARL-S and HPL-S. For our algorithms **ARL-S** and **HPL-S** we used different data structures, i.e. **A** (Array), **HM** (HashMap), **SL** (SkipList) as described in Section 4. For comparison we used synthetic datasets, because they allow us to carefully explore the effect of various data characteristics.

Figure 5a presents the runtime performance of our algorithms on different data cardinality n. We used $n = 1 \cdot 10^6$ to $10 \cdot 10^6$ tuples and 5 dimensions, since this is realistic in practical cases. We fixed the number of threads to $c = 8$ and used a domain derived from $[1, 2, 5, 100, 100]$. In this case the lattice has 367236 nodes. The array based implementations ARL-S A and HPL-S A perform best, whereas the level-based versions with non-linear time complexity are worser, cp. Section



(a) anti, d=5, c=8 (b) corr, $n = 10^6$, d=4 (c) anti, $n = 10^6$, d=5, c=16

(d) ind, $n = 10^6$, HPL-S A (e) anti, $n = 5 \cdot 10^6$, d=5 (f) anti, $n = 5 \cdot 10^6$, d=5

(g) anti, d=5, c=16 (h) anti, $n = 10^6$, c=16 (i) Zillow dataset

Fig. 5. Experimental results

4.3. Interestingly, the HashMap implementation of the BTG is not as good as the SkipList based version. Maybe this is due to the additional computation of the hash function. The Skyline size ranges from 392 ($n = 10^6$) to 3791 objects ($n = 10 \cdot 10^6$).

In Figure 5b we compared the SkipList and Array variants of ARL-S and HPL-S. We used correlated data on 4 dimensions and varied the number of threads up to 16 ('one thread per core'). As expected, ARL-S A and HPL-S A

are the best algorithms exploiting a high parallelism on a data structure having constant access time.

Figure 5c shows the segmented runtime for our algorithms, i.e., the time for the construction phase, the adding phase, and the removal phase. In all algorithms the time for the construction phase is negligible. The adding phase is the most time consuming part. The HashMap based implementations are worser than the SkipList implementations, since the adding phase takes much longer Thereby, the removal phase is nearly the same. The both array based implementations are significantly faster than the competitors. HPL-S A is slightly better than ARL-S A, in particular in the removal phase. Note that we separated the time for adding and removal in the HPL-S algorithms, since this two phases are combined to one phase in HPL-S.

We also considered the memory usage of our algorithms in this experiment. We measured the most memory consuming part, i.e., the adding phase, because in that phase all objects must fit into memory (we associated the BTG nodes with the input objects). ARL-S HM uses more than 70 MB of memory, whereas ARL-S SL using a SkipList can reduce the memory usage to 45 MB due to the fact of dynamic adding and removal of single nodes of the BTG. ARL-S A using an array needs about 50 MB of memory. In contrast, all HPL-S algorithms need much less memory due to the combined adding and removal phase. HPL-S HM using 35 MB still needs the most memory, whereas HPL-S SL uses a total memory of 5 MB. HPL-S A takes 25 MB. In summary, the HPL-S algorithm using a SkipList is the most memory saving algorithm.

For speed-up experiments we used independent data, fixed $n = 10^6$ and used $d = 3, 5$ and 7 dimensions. We executed our HPL-S A algorithm using up to 16 threads. The results are shown in Figure 5d. Our HPL-S A algorithm achieves superlinear speed-up until 6 threads, which we believe to be the result of a relative small BTG size (about 3000 nodes for $d = 7$). In the case of 3 and 5 dimensions the BTG is much smaller, but in these cases much more synchronization is necessary, because threads may try to lock the same node.

Comparison of Multi-core Skyline Algorithms. We compared our algorithms against the state-of-the-art multicore algorithms **APSkyline** [7] using equi-volume angle-based partitioning, **pSkyline** [11], and the lazy locking parallel BNL (**pBNL**) [5]. For a better overview we skipped the Parallel Divide-and-Conquer approach, because it is outperformed by pBNL [5]. Note that all our results are in line with the results presented in [5] and [7].

In Figure 5e we show the results in the case of an anti-correlated dataset as we increase the number of threads from 1 to 512. We expected to reach peak performance at 16 threads, which is the maximum number of hardware threads (two quad-core processors using Hyper-Threading). As the number of threads increase beyond 16, the performance gain moderately ceases for all algorithms due to the increased synchronization costs without additional parallel computing power. For the used low-cardinality domain $[2, 2, 2, 2, 100]$ we observed that our algorithms ARL-S A and HPL-S A outperform the competitors until 128 threads. Beyond that, the parallel computing power decreases and ARL-S A and HPL-S

A become worser. This is due to the fact of the high number of locks on the BTG nodes, in particular when using 512 threads. The Skyline has 37971 objects.

In Figure 5f we measure the speed-up of each algorithm on anti-correlated data using $n = 5 \cdot 10^6$ and 5 dimensions ($[2, 3, 5, 10, 100]$). We observed that all algorithms have nearly linear speed-up up to 8 threads. From the ninth thread on, the performance only marginally increases and beyond 16 thread it gradually decreases. This can be explained with decreasing cache locality and increasing communication costs as our test systems uses two quad-core processors with Hyper-Threading (8 cores per CPU). Starting with the ninth core, the second processor must constantly communicate with the first.

Figure 5g presents the behavior of the algorithms for increased data size. APSkyline is better than pBNL and pSkyline as mentioned in [7]. However, the domain derived from $[2, 3, 5, 10, 100]$, which is typical for Skyline computation (a few small attributes together with a large attribute) [8], is best suited for our algorithms, which significantly outperform all others.

Figure 5h shows the obtained results when increasing the number of dimensions: $d = 3$ ([2,2,100]) to $d = 7$ ([2,2,2,2,2,2,100]). The number of input tuples (anti) was fixed to $n = 10^6$ and $c = 16$. pSkyline and APSkyline are quiet similar for all dimensions, whereas pBNL is better for $d = 3$. It should be mentioned that the size of the Skyline set normally increases on anti-correlated data with the dimensionality of the dataset [25] (138 Skyline objects for $d = 3$, 4125 objects for $d = 7$). This makes Skyline processing for algorithms relying on tuple-to-tuple comparison more demanding. This experiments verifies the advantage of our algorithms based on the lattice structure and not on a tuple comparison, in particular for higher values of dimensionality.

Real Dataset. In Figure 5i we show the obtained results for the real-world Zillow dataset. The parallel BNL algorithm is outperformed in an order of magnitude by all other algorithms. APSkyline is outperformed by pSkyline because of an unfair data partitioning as mentioned in [7]. Our lattice based algorithms do not rely on any partitioning scheme and are independent from data distribution. Therefore, the best performing algorithms are ARL-S A and HPL-S A. Thereby the latter one slightly performs better. For both algorithms the adding phase is the most time consuming part. Note that we skipped the HashMap and SkipList implementations for an better overview. They are also outperformed by ARL-S A and HPL-S A. There are 95 objects in the Skyline.

6 Conclusion

In this paper we presented two algorithms for high-performance parallel Skyline computation on shared-memory multi-processor systems. Both algorithms are based on the lattice structure constructed by a Skyline query over low-cardinality domains, and do not rely on any data partitioning. Our algorithms have linear runtime complexity and a memory requirement which is linear w.r.t. the size of the lattice. In our extensive experiments on synthetic and real data, we showed the superior characteristics of these algorithms in different settings. Exploiting

the parallelization on the lattice structure we are able to outperform state-of-the-art approaches for Skyline computation on modern hardware architectures. As future work we want to extend our algorithms to handle high-cardinality domains, which could be a challenging task.

Acknowledgement. We want to thank Selke et al. [5] for providing us with the source code of the parallel BNL and pSkyline. The implementation of APSkyline is based on the source code made available by Liknes et al. [7].

References

1. Börzsönyi, S., Kossmann, D., Stocker, K.: The Skyline Operator. In: Proc. of ICDE 2001, pp. 421–430. IEEE, Washington, DC (2001)
2. Chomicki, J., Ciaccia, P., Meneghetti, N.: Skyline Queries, Front and Back. SIGMOD Rec. 42(3), 6–18 (2013)
3. Mattson, T., Wrinn, M.: Parallel Programming: Can we PLEASE get it right this time? In: Fix, L. (ed.) DAC, pp. 7–11. ACM (2008)
4. Gustafson, J.L.: Reevaluating Amdahl's law. Commun. ACM 31(5), 532–533 (1988)
5. Selke, J., Lofi, C., Balke, W.-T.: Highly Scalable Multiprocessing Algorithms for Preference-Based Database Retrieval. In: Kitagawa, H., Ishikawa, Y., Li, Q., Watanabe, C. (eds.) DASFAA 2010. LNCS, vol. 5982, pp. 246–260. Springer, Heidelberg (2010)
6. Im, H., Park, J., Park, S.: Parallel Skyline Computation on Multicore Architectures. Inf. Syst. 36(4), 808–823 (2011)
7. Liknes, S., Vlachou, A., Doulkeridis, C., Nørvåg, K.: APSkyline: Improved Skyline Computation for Multicore Architectures. In: Bhowmick, S.S., Dyreson, C.E., Jensen, C.S., Lee, M.L., Muliantara, A., Thalheim, B. (eds.) DASFAA 2014, Part I. LNCS, vol. 8421, pp. 312–326. Springer, Heidelberg (2014)
8. Morse, M., Patel, J.M., Jagadish, H.V.: Efficient Skyline Computation over Low-Cardinality Domains. In: Proc. of VLDB 2007, pp. 267–278. (2007)
9. Preisinger, T., Kießling, W.: The Hexagon Algorithm for Evaluating Pareto Preference Queries. In: Proc. of MPref 2007 (2007)
10. Godfrey, P., Shipley, R., Gryz, J.: Algorithms and Analyses for Maximal Vector Computation. The VLDB Journal 16(1), 5–28 (2007)
11. Park, S., Kim, T., Park, J., Kim, J., Im, H.: Parallel Skyline Computation on Multicore Architectures. In: Proc. of ICDE 2009, pp. 760–771 (2009)
12. Lo, E., Yip, K.Y., Lin, K.-I., Cheung, D.W.: Progressive Skylining over Web-accessible Databases. IEEE TKDE 57(2), 122–147 (2006)
13. Wu, P., Zhang, C., Feng, Y., Zhao, B.Y., Agrawal, D.P., El Abbadi, A.: Parallelizing Skyline Queries for Scalable Distribution. In: Ioannidis, Y., et al. (eds.) EDBT 2006. LNCS, vol. 3896, pp. 112–130. Springer, Heidelberg (2006)
14. Hose, K., Vlachou, A.: A Survey of Skyline Processing in Highly Distributed Environments. The VLDB Journal 21(3), 359–384 (2012)
15. Afrati, F.N., Koutris, P., Suciu, D., Ullman, J.D.: Parallel Skyline Queries. In: Proc. of ICDT 2012, pp. 274–284. ACM, New York (2012)
16. Cosgaya-Lozano, A., Rau-Chaplin, A., Zeh, N.: Parallel Computation of Skyline Queries. In: Proc. of HPCS 2007, p. 12 (2007)
17. Park, Y., Min, J.-K., Shim, K.: Parallel Computation of Skyline and Reverse Skyline Queries Using MapReduce. PVLDB 6(14), 2002–(2013)

18. Vlachou, A., Doulkeridis, C., Kotidis, Y.: Angle-based Space Partitioning for Efficient Parallel Skyline Computation. In: Proc. of SIGMOD 2008, pp. 227–238 (2008)
19. Köhler, H., Yang, J., Zhou, X.: Efficient Parallel Skyline Processing using Hyperplane Projections. In: Proc. of SIGMOD 2011, pp. 85–96. ACM (2011)
20. Heller, S., Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N., Shavit, N.: A Lazy Concurrent List-Based Set Algorithm. In: Anderson, J.H., Prencipe, G., Wattenhofer, R. (eds.) OPODIS 2005. LNCS, vol. 3974, pp. 3–16. Springer, Heidelberg (2006)
21. Bøgh, K.S., Assent, I., Magnani, M.: Efficient GPU-based Skyline computation. In: Proc. of DaMoN, pp. 5:1–5:6. ACM, New York (2013)
22. Woods, L., Alonso, G., Teubner, J.: Parallel Computation of Skyline Queries. In: Proc. of the FCCM, pp. 1–8. IEEE, Washington, DC (2013)
23. Pugh, W.: Skip Lists: A Probabilistic Alternative to Balanced Trees. Commun. ACM 33(6), 668–676 (1990)
24. Glück, R., Köppl, D., Wirsching, G.: Computational Aspects of Ordered Integer Partition with Upper Bounds. In: Bonifaci, V., Demetrescu, C., Marchetti-Spaccamela, A. (eds.) SEA 2013. LNCS, vol. 7933, pp. 79–90. Springer, Heidelberg (2013)
25. Shang, H., Kitsuregawa, M.: Skyline Operator on Anti-correlated Distributions. In: Proc. of VLDB 2013 (2013)