Parallel Skyline Computation on Multicore Architectures[☆]

Hyeonseung Im, Jonghyun Park, Sungwoo Park[*]

Pohang University of Science and Technology (POSTECH)

Republic of Korea

Abstract

With the advent of multicore processors, it has become imperative to write parallel programs if one wishes to exploit the next generation of processors. This paper deals with skyline computation as a case study of parallelizing database operations on multicore architectures. First we parallelize three sequential skyline algorithms, BBS, SFS, and SSkyline, to see if the design principles of sequential skyline computation also extend to parallel skyline computation. Then we develop a new parallel skyline algorithm PSkyline based on the divide-and-conquer strategy. Experimental results show that all the algorithms successfully utilize multiple cores to achieve a reasonable speedup. In particular, PSkyline achieves a speedup approximately proportional to the number of cores when it needs a parallel computation the most.

Keywords: Skyline computation, Multicore architecture, Parallel computation

1. Introduction

Multicore processors are going mainstream [26]. As a response to the problem of excessive power consumption and the lack of new optimization techniques,

[☆]This is a significantly extended version of the paper that appeared in the 22nd IEEE International Conference on Data Engineering, 2009 [20]. This work was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology (MEST) / National Research Foundation of Korea (NRF) (Grant 2010-0001726), and the Korea Research Foundation Grant funded by the Korean Government (KRF-2008-313-D00969).

[*]Corresponding author

Email address: {genilhs, parjong, gla}@postech.ac.kr (Hyeonseung Im, Jonghyun Park, Sungwoo Park)

the industry has adopted a new strategy for boosting processor performance by integrating multiple cores into a single processor instead of increasing clock frequency. In upcoming years, we will see processors with eight, sixteen, or more cores, but not with much higher clock frequency.

The advent of multicore processors is making a profound impact on software development [27]. As there is little performance gain when running sequential programs on multicore processors, it is imperative to write parallel programs in order to exploit the next generation of processors. Due to simpler design and lower clock frequency in individual cores, sequential programs may even experience performance loss on tomorrow's multicore processors.

This radical change in processor architectures begs an important question for the database community: *how can we exploit multicore architectures in implementing database operations?* Since multicore architectures combine multiple independent cores sharing common input/output (I/O) devices, this question is particularly relevant if database operations under consideration are computationally intensive, but not I/O intensive. In such cases, multicore architectures offer an added advantage of negligible or low overhead for communications between parallel threads, which we can implement as reads and writes to the main memory or disk.

This paper deals with *skyline computation* [2] as a case study of parallelizing database operations on multicore architectures. Given a multi-dimensional dataset of tuples, a skyline computation returns a subset of tuples, called *skyline tuples*, that are no worse than, or not dominated by, any other tuples when all dimensions are considered together. Because of its potential applications in decision making, skyline computation has drawn a lot of attention in the database community [14, 3, 9, 19, 16, 1, 35].

The computationally intensive nature of skyline computation makes it a good candidate for parallelization especially on multicore architectures. Typically the cost of skyline computation depends heavily on the number of comparisons between tuples, called *dominance tests*, which involve only integer or floating-point number comparisons and no I/O. Since a large number of dominance tests can often be performed independently, skyline computation has a good potential to exploit multicore architectures. So far, however, its parallelization has been considered mainly on distributed architectures [33, 4, 32, 31]; only Selke *et al.* recently consider its parallelization on multicore architectures [24].

We parallelize three sequential skyline algorithms of different kinds to see if the design principle of sequential skyline computation also extends to parallel skyline computation. From index-based skyline algorithms, we choose the

branch-and-bound skyline (BBS) algorithm [19] which uses R-trees [10] to eliminate from dominance tests a block of tuples at once. From sorting-based skyline algorithms, we choose the sort-filter-skyline (SFS) algorithm [3] which presorts a dataset according to a monotone preference function so that no tuple is dominated by succeeding tuples in the sorted dataset. We also parallelize a new nested-loop skyline algorithm, called SSkyline (Simple Skyline), which neither uses index structures nor presorts a dataset. In addition to parallelizing three sequential skyline algorithms, we also develop a new skyline algorithm, called PSkyline (Parallel Skyline), which is based on the divide-and-conquer strategy and designed specifically for parallel skyline computation. PSkyline is remarkably simple because it uses no index structures and divides a dataset linearly into smaller blocks of the same size (unlike existing divide-and-conquer skyline algorithms which exploit geometric properties of datasets).

We test the four parallel skyline algorithms on a sixteen-core machine (with four quad-core CPUs). Experimental results show that all the algorithms successfully utilize multiple cores to achieve reasonable speedups except on low-dimensional datasets and datasets with a low density of skyline tuples on which the sequential algorithms already run fast enough. In particular, the comparison between parallel BBS and PSkyline suggests that for the efficiency of a parallel skyline algorithm, a simple organization of candidate skyline tuples may be a better choice than a clever organization that eliminates from dominance tests a block of tuples at once but favors only sequential skyline computation.

Although the main topic of this paper is parallel skyline computation on multicore architectures, its main contribution also lies in providing evidence that the time is ripe for a marriage between database operations and multicore architectures. In order to exploit multicore architectures to their fullest, we may have to devise new index structures or re-implement database operations accordingly. In fact, researches along this line are already producing highly promising (and even surprising) results in the database community [17, 11, 12, 29, 13, 6]. Certainly we do not want to find ourselves struggling to squeeze performance out of just a single core while all other 31 cores remain idle!

This paper is organized as follows. Section 2 introduces skyline computation and the parallel programming environment OpenMP [5] for implementing the parallel skyline algorithms, and discusses related work. Sections 3 and 4 explain how we parallelize BBS and SFS, respectively. Section 5 presents SSkyline and its parallelization. Section 6 presents the design and implementation of our parallel skyline algorithm PSkyline. Section 7 gives experimental results and Section 8 concludes.

2. Preliminaries

This section reviews basic properties of skyline computation and gives a brief introduction to OpenMP. Then it discusses related work.

2.1. Skyline computation

Given a dataset, a skyline query retrieves a subset of tuples, called a skyline set, that are not dominated by any other tuples. Under the assumption that smaller values are better, a tuple p dominates another tuple q if all elements of p are smaller than or equal to their corresponding elements of q and there exists at least one element of p that is strictly smaller than its corresponding element of q. Thus the skyline set consists of those tuples that are no worse than any other tuples when all dimensions are considered together.

Let us formally define the skyline set of a d-dimensional dataset D. We write $p[i]$ for the i-th element of tuple p where $1 \leq i \leq d$. We write $p \prec q$ to mean that tuple p dominates tuple q, *i.e.*, $p[i] \leq q[i]$ holds for $1 \leq i \leq d$ and there exists a dimension k such that $p[k] < q[k]$. We also write $p \not\prec q$ to mean that p does not dominate q, and $p \prec\succ q$ to mean that p and q are incomparable ($p \not\prec q$ and $q \not\prec p$). Then the skyline set $\mathcal{S}(D)$ of D is defined as

$$\mathcal{S}(D) = \{p \in D \mid q \not\prec p \text{ if } q \in D\}.$$

Note that $\mathcal{S}(D) \subset D$ and $\mathcal{S}(\mathcal{S}(D)) = \mathcal{S}(D)$ hold. We refer to those tuples in the skyline set as skyline tuples.

The computational cost of a skyline query mainly depends on the number of dominance tests performed to identify skyline tuples. A dominance test between two tuples p and q determines whether p dominates q ($p \prec q$), q dominates p ($q \prec p$), or p and q are incomparable ($p \prec\succ q$). The computational cost of a single dominance test increases with the dimensionality of the dataset.

Usually a skyline algorithm reduces the number of dominance tests by exploiting specific properties of skyline tuples. For example, transitivity of $\prec$ allows us to eliminate from further consideration any tuple as soon as we find that it is dominated by another tuple:

Proposition 2.1 (Transitivity of $\prec$)**.**
If $p \prec q$ and $q \prec r$, then $p \prec r$.

Another useful property is that we may consider incomparable datasets independently of each other. We say that two datasets D_1 and D_2 are incomparable, written $D_1 \prec\succ D_2$, if $p \prec\succ q$ holds for every pair of tuples $p \in D_1$ and $q \in D_2$.

Proposition 2.2 (Incomparability)**.**
If $D_1 \prec\succ D_2$*, then* $\mathcal{S}(D_1 \cup D_2) = \mathcal{S}(D_1) \cup \mathcal{S}(D_2)$.

This property is the basis for existing divide-and-conquer skyline algorithms which preprocess a given dataset into incomparable datasets in order to avoid unnecessary dominance tests. In the worst case, however, a dominance test between every pair of tuples is necessary because every tuple may be a skyline tuple.

Our parallel skyline algorithm PSkyline is also a divide-and-conquer algorithm, but uses distributivity of $\mathcal{S}$ as its basis:

Proposition 2.3 (Distributivity of $\mathcal{S}$)**.**
$\mathcal{S}(D_1 \cup D_2) = \mathcal{S}(\mathcal{S}(D_1) \cup \mathcal{S}(D_2))$.

That is, it computes $\mathcal{S}(D_1 \cup D_2)$ by first computing $\mathcal{S}(D_1)$ and $\mathcal{S}(D_2)$ separately and then merging $\mathcal{S}(D_1)$ and $\mathcal{S}(D_2)$. Here D_1 and D_2 do not need to be incomparable datasets, which means that PSkyline may divide a given dataset in an arbitrary way.

2.2. OpenMP

We use the OpenMP programming environment [5] both to parallelize BBS, SFS, and SSkyline and to instantiate PSkyline into an implementation tailored to multicore architectures. OpenMP consists of a set of compiler directives and runtime library routines to support shared-memory parallel programming in C, C++, and Fortran.

OpenMP allows incremental parallelization of an existing sequential program by annotating with OpenMP directives. The following C code illustrates how to parallelize a **for** loop using OpenMP:

```
#pragma omp parallel for default(shared) private(i)
for (i = 0; i < size; i++)
  output[i] = f(input[i]);
```

The OpenMP directive in the first line specifies that iterations with loop variable `i` be executed in parallel. Thus the **for** loop applies function `f` to each element of array `input` and stores the result in array `output` in parallel. Our pseudocode uses the **parallel for** construct to denote such parallel **for** loops.

2.3. Related work

The problem of skyline computation is known as the maximal vector problem in the computational geometry community. Kung *et al.* [15] study the time complexity of the problem with a theoretical divide-and-conquer algorithm and a concrete algorithm for three-dimensional datasets. Stojmenović and Miyakawa [25] present a divide-and-conquer algorithm for two-dimensional datasets. Matousek [18] uses matrix multiplication to develop an algorithm for datasets whose dimensionality and size are equal. Dehne *et al.* [8] present a divide-and-conquer algorithm for three-dimensional datasets. All these algorithms are parallelizable, but are not suitable for skyline computation in the database context because of the constraints on the dimensionality of datasets.

For generic skyline computation as a database operation, there are several algorithms that do not require special index structures or preprocessing. The block-nested-loops (BNL) algorithm [2] performs a dominance test between every pair of tuples while maintaining a window of candidate skyline tuples. The sort-filter-skyline (SFS) algorithm [3] is similar to BNL, but presorts a dataset according to a monotone preference function so that no tuple is dominated by succeeding tuples in the sorted dataset. Presorting a dataset allows SFS to compare a candidate skyline tuple only with those tuples in the window (which contains only skyline tuples) and not with succeeding tuples in the sorted dataset during its filtering phase. LESS (linear elimination sort for skyline) [9] incorporates two optimization techniques into the external sorting phase of SFS. It uses an elimination-filter window to eliminate tuples during the initial sorting phase, and combines the final sorting phase with the initial filtering phase of SFS. SaLSa (sort and limit skyline algorithm) [1] is another skyline algorithm that presorts a dataset, but unlike SFS and LESS, does not necessarily inspect every tuple in the sorted dataset. It maintains a special tuple called a stop point in order to check if all remaining tuples in the sorted dataset can be eliminated. The recent skyline algorithm OSPSPF (object-based space partitioning skyline with partitioning first) [35] does not use index structures, but constantly updates an in-memory tree structure called LCRS (left-child/right-sibling) tree to store current skyline tuples. SFS, LESS, and SaLSa are all amenable to parallelization because they involve sorting a dataset, but their full parallelization has not been reported. (OSPSPF seems difficult to parallelize because of its use of a dynamic tree structure.) We choose to parallelize SFS as the representative algorithm that does not use special index structures or preprocessing.

There are also several skyline algorithms that exploit special index structures. The nearest-neighbor (NN) algorithm [14] and the branch-and-bound skyline (BBS)

algorithm [19] use R-trees [10] as their index structures in order to eliminate from dominance tests a block of tuples at once. BBS has an important property that it is I/O optimal: it makes a single visit to only those nodes in the R-tree that may contain skyline tuples. The ZSearch algorithm [16] uses a new variant of B^+-tree, called *ZBtree*, for maintaining the set of candidate skyline tuples in Z-order [22]. We choose to parallelize BBS as the representative algorithm that uses special index structures.

Parallel skyline computation so far has been considered mainly on distributed architectures in which participating nodes in a network share nothing and communicate only by exchanging messages [33, 4, 32, 31]. While similar in spirit in that we attempt to utilize multiple computational units, we focus on exploiting properties specific to multicore architectures in which participating cores inside a processor share everything and communicate simply by updating the main memory. Recently Selke *et al.* [24] consider parallel skyline computation on multicore architectures. They parallelize BNL using three synchronization and locking techniques: continuous locking, lazy locking, and lock free synchronization using a special hardware operation called compare-and-swap. They experimentally show that their parallel BNL algorithms using lazy locking and lock free synchronization techniques achieve a speedup approximately proportional to the number of cores. Our work differs from their work in that we ensure the correctness of our algorithms without using any synchronization techniques. (Locking and synchronization often lead to performance loss.)

3. Parallel BBS

This section develops a parallel version of the branch-and-bound skyline (BBS) algorithm. We give a brief introduction to BBS and explain how to parallelize it.

3.1. Branch-and-bound skyline algorithm

BBS is a state-of-the-art skyline algorithm which uses R-trees as its index structures. Algorithm 1 describes the pseudocode of BBS. It takes as input an R-tree R built from a dataset and returns as output the skyline set. BBS maintains in the main memory a heap H holding R-tree nodes and an array S holding skyline tuples. In lines 5 and 8, BBS performs dominance tests between a candidate skyline tuple (or an intermediate node of R) and all skyline tuples in S. If e is an intermediate node in line 6, BBS retrieves children of e before executing the loop in lines 7–10.

Algorithm 1 BBS (R: R-tree)

```
1:  S ← φ
2:  insert all children of the root R into heap H
3:  while H is not empty do
4:      e ← remove the top entry of H
5:      if ∀e' ∈ S.e' ⊀ e then
6:          if e is an intermediate node of R then
7:              for each child c_i of e do
8:                  if ∀e' ∈ S.e' ⊀ c_i then
9:                      insert c_i into H
10:             endfor
11:         else insert e into S
12:     end if
13: end while
14: return S
```

3.2. Parallelizing BBS

A naive approach to parallelizing BBS is to perform dominance tests in lines 5 and 8 of Algorithm 1 in parallel. These two lines are the most time-consuming part of BBS and also do not require I/O operations. For example, the following table shows profiling results for two 10-dimensional datasets with 102400 tuples from Section 7:

	lines 5 and 8	others
dataset 1	97.6%	2.4%
dataset 2	98.6%	1.4%

Moreover dominance tests in lines 5 and 8 are all independent of each other. Therefore lines 5 and 8 are the most appropriate part to rewrite when parallelizing BBS.

The naive approach, however, is not so effective for two reasons. First, as soon as a certain thread finds a skyline tuple e' in S dominating a common R-tree node (e in line 5 and c_i in line 8 of Algorithm 1), we may terminate all other threads immediately because the $\forall$ condition is already violated, but we still have to wait until all other threads terminate by themselves (partially because of the limitation of OpenMP). If we explicitly initiate communications between threads in such cases, the communication cost far outweighs the benefit of parallel dominance tests. Second spawning a thread itself is relatively expensive in comparison with

the actual computation assigned to each thread, especially when the number of skyline tuples is small.

Our approach is to execute lines 5 and 8 of Algorithm 1 for multiple candidate skyline tuples in parallel. That is, we first accumulate candidate skyline tuples that are incomparable with each other, and then inspect these candidate skyline tuples in parallel. This approach resolves the two problems with the naive approach. First all threads independently perform dominance tests on different candidate skyline tuples. That is, each thread checks the whole condition $\forall e' \in S'.e' \not\prec e$ for a specific candidate skyline tuple e rather than performing a single dominance test $e' \not\prec e$ as in the naive approach. Second, by accumulating many candidate skyline tuples before spawning multiple threads, we can assign to each thread a relatively heavy computation whose cost far exceeds the cost of spawning the thread itself.

Algorithm 2 shows the pseudocode of parallel BBS. It first accumulates incomparable R-tree nodes in an additional array S' (lines 4–7). It stops accumulating R-tree nodes either when the number of candidate skyline tuples (or equivalently leaf nodes) in S' exceeds a threshold β or when an intermediate node is added into S'. Then it inspects in parallel if each accumulated node e_i in S' is dominated by any skyline tuple in S (lines 9–12). For each node e_i that survives dominance tests, parallel BBS sets a corresponding flag $flag_i$ to **true**. If the surviving node e_i is an intermediate node, parallel BBS inserts into heap H all children of e_i that are not dominated by any skyline tuple in S (lines 15–23); it performs dominance tests for the children of e_i in parallel and uses flags $cflag_j$ to indicate whether the children of e_i survive dominance tests (lines 16–19). If the surviving node e_i is a leaf node, or equivalently a skyline tuple, parallel BBS inserts it into S (line 24).

Note that we stop accumulating R-tree nodes in S' as soon as an intermediate node is added to S' (line 4). This is necessary to ensure the correctness of parallel BBS: even if an intermediate node dominates a candidate skyline tuple, the candidate tuple may still be a (final) skyline tuple, since dominance tests use the lower-left corner of the minimum bounding rectangle of an intermediate node. If we do not stop accumulating R-tree nodes in S' in such cases, we may discard a skyline tuple that happens to be dominated by an intermediate node.

4. Parallel SFS

This section develops a parallel version of the sort-filter-skyline (SFS) algorithm. We give a brief introduction to SFS and explain how to parallelize it.

Algorithm 2 Parallel BBS (R: R-tree)

```
 1: S ← φ, S' ← φ
 2: insert all children of the root R into heap H
 3: while H or S' is not empty do
 4:    if there is no intermediate node in S' and |S'| < β and H is not empty then
 5:       e ← remove the top entry of H
 6:       if ∀e' ∈ S'.e' ⊀ e then
 7:          insert e into S'
 8:    else
 9:       parallel for each e_i ∈ S' do
10:          if ∀e' ∈ S.e' ⊀ e_i then
11:             flag_i ← true
12:       end parallel for
13:       for each e_i ∈ S' do
14:          if flag_i = true then
15:             if e_i is an intermediate node of R then
16:                parallel for each child c_j of e_i do
17:                   if ∀e' ∈ S.e' ⊀ c_j then
18:                      cflag_j ← true
19:                end parallel for
20:                for each child c_j of e_i do
21:                   if cflag_j = true then
22:                      insert c_j into H
23:                end for
24:             else insert e_i into S
25:          end if
26:       end for
27:       S' ← φ
28:    end if
29: end while
30: return S
```

4.1. Sort-filter-skyline algorithm

SFS is a skyline algorithm based on presorting and uses no index structures. Algorithm 3 describes the pseudocode of SFS. It takes as input an array $D[1 \ldots n]$ of tuples which is assumed to fit in the main memory. It returns as output the skyline set $\mathcal{S}(D[1 \ldots n])$. SFS maintains an array S holding skyline tuples and visits all tuples in sorted order (line 3). For each tuple $D[i]$, SFS performs dominance tests with all skyline tuples in S (line 4). If no skyline tuples in S dominate $D[i]$,

Algorithm 3 SFS ($D[1 \ldots n]$)

```
1: S ← φ
2: sort D[1...n] topologically with respect to ≺
3: for i = 1 to n do
4:     if ∀ e ∈ S. e ⊀ D[i] then
5:         insert D[i] into S
6: end for
7: return S
```

SFS inserts $D[i]$ into S (line 5).

4.2. Parallelizing SFS

The main idea in parallelizing SFS is the same as in parallelizing BBS: instead of performing dominance tests in line 4 of Algorithm 3 in parallel, we first accumulate multiple candidate skyline tuples (up to a threshold number γ) that are incomparable with each other, and then inspect each candidate skyline tuple independently. In this way, we assign to each thread a relatively heavy computation which attempts to compare one or more candidate skyline tuples with all skyline tuples in S.

Algorithm 4 describes the pseudocode of parallel SFS. While visiting each tuple in sorted order (line 3), parallel SFS accumulates incomparable tuples in an additional array S' (lines 4–5). After accumulating γ incomparable tuples (line 6), parallel SFS inspects all accumulated tuples in S' in parallel to identify and store new skyline tuples in a temporary array S'' (lines 8–11). In lines 15–19, parallel SFS inspects in parallel remaining candidate skyline tuples in S' (whose size is less than γ).

5. Parallel SSkyline

This section develops a nested-loop skyline algorithm which neither uses index structures nor presorts a dataset. The key idea is already found in another skyline algorithm called Best [30], but we have independently discovered a highly efficient cache-conscious implementation of the same idea, which we call SSkyline (Simple Skyline). We first introduce SSkyline and prove its correctness. Then we explain how to parallelize it.

5.1. Simple skyline algorithm

Algorithm 4 Parallel SFS ($D[1 \ldots n]$)

```
1: S ← φ, S' ← φ
2: sort D[1...n] topologically with respect to ≺
3: for i = 1 to n do
4:    if ∀ e ∈ S'. e ⊀ D[i] do
5:       insert D[i] into S'
6:    if |S'| = γ then
7:       S'' ← φ
8:       parallel for each e ∈ S' do
9:          if ∀ e' ∈ S. e' ⊀ e then
10:            insert e into S''
11:      end parallel for
12:      S ← S ∪ S'', S' ← φ
13:   end if
14: end for
15: S'' ← φ
16: parallel for each e ∈ S' do
17:    if ∀ e' ∈ S. e' ⊀ e then
18:       insert e into S''
19: end parallel for
20: return S ∪ S''
```

Algorithm 5 describes the pseudocode of SSkyline. As input, it takes an array $D[1 \ldots n]$ of tuples which is assumed to fit in the main memory. As output, it returns the skyline set $\mathcal{S}(D[1 \ldots n])$. We design SSkyline as an in-place algorithm which requires no extra memory, but overwrites the input array. For notational convenience, we write $D[i]$ for the i-th element of array D and $D[i \ldots j]$ for the subarray of D from index i to index j.

SSkyline uses two nested loops. The outer loop uses $head$ as its loop variable which is initialized to 1 and always increases. The inner loop uses i as its loop variable which is initialized to $head + 1$ and always increases. The two loops share another variable $tail$ which is initialized to n and always decreases. When i increases past $tail$, the inner loop terminates; when $head$ and $tail$ meet in the middle, the outer loop terminates.

In order to show the correctness of SSkyline, we first analyze the inner loop and then prove that $D[1 \ldots head]$ holds the skyline set of the input array when the outer loop terminates. Roughly speaking, the inner loop searches for a skyline tuple in $D[head \ldots tail]$ and stores it in $D[head]$; the outer loop repeats the inner

Algorithm 5 SSkyline ($D[1 \ldots n]$)

```
 1: head ← 1
 2: tail ← n
 3: while head < tail do
 4:     i ← head + 1
 5:     while i ≤ tail do
 6:         if D[head] ≺ D[i] then
 7:             D[i] ← D[tail]
 8:             tail ← tail − 1
 9:         else if D[i] ≺ D[head] then
10:             D[head] ← D[i]
11:             D[i] ← D[tail]
12:             tail ← tail − 1
13:             i ← head + 1
14:         else
15:             i ← i + 1
16:         end if
17:     end while
18:     if head < tail then
19:         head ← head + 1
20: end while
21: return D[1 . . . head]
```

loop until it identifies all skyline tuples.

Let D_{inner} be the value of $D[head \ldots tail]$ before the inner loop starts (after line 4). SSkyline maintains the following invariants at the beginning of the inner loop (in line 5):

1. $D[head \ldots tail] \subset D_{inner}$.
2. $D[head] \prec\!\succ D[(head + 1) \ldots (i - 1)]$.
3. $\mathcal{S}(D_{inner}) = \mathcal{S}(D[head \ldots tail])$.

Invariant 1 means that $D[head \ldots tail]$ is a subset of D_{inner}. Invariant 2 implies that $D[head]$ is a skyline tuple of $D[head \ldots (i - 1)]$. Invariant 3 implies that in order to compute $\mathcal{S}(D_{inner})$, we only have to consider a subarray $D[head \ldots tail]$.

Proposition 5.1. *The inner loop maintains Invariants 1, 2, and 3.*

Proof. See Appendix. □

The inner loop terminates when $i = tail + 1$. By Invariant 2, $D[head]$ is a skyline tuple of $D[head \ldots tail]$, and by Invariant 3, $D[head]$ is a skyline tuple of $\mathcal{S}(D_{inner})$. Thus the inner loop terminates when it identifies a skyline tuple of D_{inner}.

Let D_{outer} be the value of $D[1 \ldots n]$ before the outer loop begins (after line 2). SSkyline maintains the following invariants at the beginning of the outer loop (in line 3):

4. $D[1 \ldots (head - 1)] \prec\!\succ D[head \ldots tail]$.
5. $\mathcal{S}(D_{outer}) = \mathcal{S}(D[1 \ldots tail])$.
6. $\mathcal{S}(D[1 \ldots (head - 1)]) = D[1 \ldots (head - 1)]$.

Invariant 4 implies that we may consider $D[head \ldots tail]$ independently of $D[1 \ldots (head - 1)]$. Invariant 5 implies that in order to compute $\mathcal{S}(D_{outer})$, we only have to consider a subarray $D[1 \ldots tail]$. Invariant 6 means that $D[1 \ldots (head - 1)]$ is a skyline set.

Proposition 5.2. *The outer loop maintains Invariants 4, 5, and 6.*

Proof. See Appendix. □

The outer loop terminates when $head = tail$, and SSkyline returns $D[1 \ldots head] = \mathcal{S}(D_{outer})$:

$$
\begin{array}{lll}
 & D[1 \ldots head] & \\
= & D[1 \ldots head - 1] \cup D[head] & \\
= & \mathcal{S}(D[1 \ldots head - 1]) \cup D[head] & \text{by Invariant 6} \\
= & \mathcal{S}(D[1 \ldots head - 1]) \cup \mathcal{S}(D[head]) & \\
= & \mathcal{S}(D[1 \ldots head - 1] \cup D[head]) & \text{by Invariant 4and Proposition 2.2} \\
= & \mathcal{S}(D[1 \ldots head]) & \\
= & \mathcal{S}(D_{outer}) & \text{by Invariant 5}
\end{array}
$$

Thus we state the correctness of SSkyline as follows:

Theorem 5.3. *Let D_{outer} be the input to SSkyline. When the outer loop terminates, we have $D[1 \ldots head] = \mathcal{S}(D_{outer})$.*

As it maintains a window of tuples in nested loops, SSkyline is similar to the block-nested-loops (BNL) algorithm, but with an important difference. BNL maintains a window of incomparable tuples that may later turn out to be non-skyline tuples. Hence, each time it reads a tuple, the window either grows, shrinks, or remains unchanged. In contrast, SSkyline maintains a window of skyline tuples, namely $D[1 \ldots (head - 1)]$, which grows only with new skyline tuples. In essence, BNL is designed to minimize disk reads, whereas SSkyline is designed to exploit the assumption that the input array fits in the main memory.

5.2. Parallelizing SSkyline

SSkyline is difficult to parallelize for two reasons. First SSkyline frequently updates elements of an input array. Thus, in order to parallelize SSkyline, we need to ensure that while a thread is accessing a certain element of the array, other threads do not update the same element. Locking and synchronization techniques can prevent such situations, but they often result in performance loss. Second the invariants of SSkyline fail to hold if we simply parallelize either the outer loop (lines 3–20) or the inner loop (lines 5–17) of SSkyline. For example, assume that we parallelize the inner loop. When a thread executes the i_k-th iteration, $D[head]$ is not necessarily incomparable with all the tuples in $D[(head+1)\dots(i_k-1)]$ because $D[head]$ may not have been compared yet with all the tuples in $D[(head+1)\dots(i_k-1)]$. Hence Invariant 2 does not hold. Similarly, if we parallelize the outer loop, Invariant 6 does not hold: when a thread executes the $head_k$-th iteration, all the tuples in $D[1\dots(head_k-1)]$ are not necessarily skyline tuples because dominance tests for such tuples may not have been completed yet. In summary, SSkyline is difficult to parallelize because it keeps reorganizing the input array so that the front subarray $D[1\dots(head-1)]$ holds skyline tuples while the next subarray $D[head\dots tail]$ holds candidate skyline tuples.

We address this problem by associating a flag with each tuple. More precisely, instead of reorganizing an input array itself, we update flags in an atomic way: we first mark all candidate skyline tuples as $unknown$; then, whenever finding a skyline tuple, we mark it as $skyline$; if a tuple is found to be dominated by another tuple, we mark it as $dominated$. Note that there is only one way to update each flag. That is, a $skyline$ flag cannot be updated to $dominated$ and vice versa. Hence we do not need to employ a synchronization technique in parallelizing SSkyline. Furthermore, since we do not maintain two continuous subarrays for skyline tuples and candidate skyline tuples, the invariant of parallel SSkyline becomes much simpler: only skyline tuples can be marked as $skyline$ and no skyline tuples are ever marked as $dominated$.

Algorithm 6 describes the pseudocode of parallel SSkyline. It first marks every tuple in D as $unknown$ (line 1), and divides D into b smaller blocks (line 2). Then it inspects all blocks in parallel to find skyline tuples in each block (lines 3–23). To simplify the notation, for each block B whose first element is $D[l]$, we write k to mean index $(k+l-2)\%n+1$ (line 4). In other words, we interpret $D[k]$ in lines 4–22 as $D[(k+l-2)\%n+1]$. Thus, in fact, the innermost loop visits D in the order of $D[l+1],\cdots,D[n],D[1],\cdots,D[l-1]$ (line 7). For each block B, parallel SSkyline select a candidate skyline tuple $D[head]$ in B which is marked

Algorithm 6 Parallel SSkyline ($D[1 \ldots n]$)

```
 1: mark every tuple in D[1...n] as unknown
 2: divide D[1...n] into b blocks
 3: parallel for each block B do
 4:    // use array index k to mean (k + l − 2)%n + 1 where B[1] = D[l]
 5:    for each i such that D[i] is marked as unknown and D[i] ∈ B do
 6:       head ← i
 7:       for each j such that D[j] is marked as unknown and j ∈ {(i + 1), ..., n} do
 8:          if D[head] is not marked as unknown
 9:             break
10:          if D[j] is not marked as unknown
11:             continue
12:          if D[head] ≺ D[j] then
13:             mark D[j] as dominated
14:          else if D[j] ≺ D[head] then
15:             mark D[head] as dominated
16:             head ← j
17:             restart
18:          end if
19:       end for
20:       if D[head] is marked as unknown
21:          mark D[head] as skyline
22:    end for
23: end parallel for
24: return all the tuples in D[1...n] marked as skyline
```

as $unknown$ (lines 5–6), and compares it with all other $unknown$ tuples $D[j]$ in $D[(i+1) \ldots n]$ (lines 7–19). Since other threads may mark $D[head]$ and $D[j]$ as $dominated$ or $skyline$, parallel SSkyline checks if they are marked as $unknown$ before performing a dominance test (lines 8–11). Parallel SSkyline marks $D[j]$ as $dominated$ if $D[head]$ dominates $D[j]$ (lines 12–13). In the case that $D[j]$ dominates $D[head]$, parallel SSkyline not only marks $D[head]$ as $dominated$ but also restarts the innermost loop with $D[j]$ as a tuple to be compared with other tuples (lines 14–17). We restart the innermost loop with $D[j]$ because $D[j]$ is more likely to dominate many other tuples than an arbitrary tuple. In lines 20–21, if $D[head]$ is not dominated by any other tuple, parallel SSkyline marks it as $skyline$.

In order to avoid sequentially scanning the input array D in lines 5 and 7, each

tuple maintains an index to the next candidate $unknown$ tuple. Specifically, when the index maintained by a tuple $D[i]$ is j, all tuples from $D[i]$ to $D[j-1]$ are guaranteed to be marked as either $dominated$ or $skyline$, but not as $unknown$. Parallel SSkyline updates these indexes whenever it finds $unknown$ tuples in lines 5 and 7. With this optimization technique, parallel SSkyline achieves a speedup approximately proportional to the number of cores.

6. Parallel skyline algorithm PSkyline

This section develops our parallel skyline algorithm PSkyline (Parallel Skyline) and proves its correctness. It also describes an implementation of PSkyline.

6.1. Notational conventions

In the design of PSkyline, we make extensive use of functions (*e.g.*, functions that take other functions as their arguments). Hence we first describe our notation for functions and function applications. The notation is similar to the syntax of the functional programming language Haskell [21].

We write $f(x)$ for an application of function f to argument x. A function may take multiple arguments and we write $f\ (x_1,\ \cdots,\ x_n)$ for an application of function f to arguments $x_1, \cdots, x_n$. If an application of a function f to arguments $x_1, \cdots, x_n$ returns e as the result, we write the specification for f as follows:

$$f\ (x_1,\ \cdots,\ x_n) \stackrel{\text{def}}{=} e$$

For a binary function f, we write $\langle f \rangle$ for an equivalent infix operator such that $f\ (x,\ y) = x\ \langle f \rangle\ y$.

In developing PSkyline, we consider two parallel functions: PMap and PReduce. PMap (Parallel Map) takes a unary function f and a set D as its arguments, and applies f to each element of D in parallel:

$$\text{PMap}\ (f,\ \{x_1, x_2, \cdots, x_n\}) \stackrel{\text{def}}{=} \{f(x_1), f(x_2), \cdots, f(x_n)\}$$

If f takes $O(1)$ sequential time, PMap $(f,\ D)$ takes $O(1)$ parallel time. PReduce (Parallel Reduce) takes an associative binary function f and a set D, and collapses D into a single element by repeatedly applying f to its elements:

$$\begin{aligned} \text{PReduce}\ (f,\ \{x\}) &\stackrel{\text{def}}{=} x \\ \text{PReduce}\ (f,\ \{x_1, x_2, \cdots, x_n\}) &\stackrel{\text{def}}{=} x_1\ \langle f \rangle\ x_2\ \langle f \rangle\ \cdots\ \langle f \rangle\ x_n \end{aligned}$$

The associativity of f ensures that PReduce $(f,\ D)$ may apply f to elements of D in any order. If f takes $O(1)$ time, PReduce $(f,\ D)$ takes $O(\log n)$ parallel time where n is the length of D.

We also consider SReduce (Sequential Reduce), a sequential version of PReduce, defined as follows:

$$\text{SReduce } (f,\ \{x_1, \cdots, x_{n-1}, x_n\}) \stackrel{\text{def}}{=} (\text{SReduce } (f,\ \{x_1, \cdots, x_{n-1}\}))\ \langle f \rangle\ x_n$$

SReduce $(f,\ D)$ takes $O(n)$ sequential time if f takes $O(1)$ sequential time and n is the length of D.

6.2. Overall design

We wish to design a parallel skyline algorithm that computes the skyline set $\mathcal{S}(D)$ of a given dataset D. Our goal is to define a function PSkyline such that

$$\text{PSkyline } (D) \stackrel{\text{def}}{=} \mathcal{S}(D).$$

Basically PSkyline is a simple divide-and-conquer algorithm:

1. It divides D into b smaller blocks $D_1, \cdots, D_b$.
2. For each block D_i ($1 \leq i \leq b$), it computes $\mathcal{S}(D_i)$ separately.
3. It merges b skyline sets $\mathcal{S}(D_1), \cdots, \mathcal{S}(D_b)$.

When dividing D, PSkyline does not use a particular strategy. Rather it simply divides D linearly into smaller blocks so that their concatenation rebuilds D:

$$D \ = \ D_1 \cup \cdots \cup D_b$$

In contrast, most of the existing divide-and-conquer algorithms for skyline computation divide the dataset geometrically (*e.g.*, by repeatedly splitting tuples along the median element in each dimension) in order to eliminate from dominance tests a block of tuples at once.

In order to define PSkyline in terms of PMap and PReduce functions, we use SSkyline introduced in Section 5.1 and introduce an auxiliary function SMerge (Sequential Merge). SSkyline performs a sequential computation to obtain the skyline set of a given block. (Although we may use any sequential algorithm, we choose SSkyline because it particularly runs fast on small datasets.) SMerge performs a sequential computation to merge two skyline sets:

$$\begin{aligned} \text{SSkyline } (D_i) &\stackrel{\text{def}}{=} \mathcal{S}(D_i) \\ \text{SMerge } (S_1,\ S_2) &\stackrel{\text{def}}{=} \mathcal{S}(S_1 \cup S_2) \quad \text{where } \mathcal{S}(S_1) = S_1 \text{ and } \mathcal{S}(S_2) = S_2 \end{aligned}$$

Algorithm 7 PSkyline ($D[1 \ldots n]$)

1: divide $D[1 \ldots n]$ into $D_1 \cdots D_b$
2: **parallel for** each D_i **do**
3: $S_i \leftarrow$ SSkyline (D_i)
4: **end parallel for**
5: $S \leftarrow \phi$
6: **for** $i = 1$ **to** n **do**
7: $S \leftarrow$ PMerge $(S,\ S_i)$
8: **end for**
9: **return** S

Using the fact that SMerge is associative, we obtain a definition of PSkyline that uses PMap and PReduce:

$$\text{PSkyline}_1\ (D) \stackrel{\text{def}}{=} \text{PReduce}\ (\text{SMerge},\ (\text{PMap}\ (\text{SSkyline},\ \{D_1, \cdots, D_b\}))) \quad \text{where } D = D_1 \cup \cdots \cup D_b$$

A drawback of PSkyline_1 is that the whole computation reverts to a sequential computation precisely when it needs a parallel computation the most. To see why, observe that $\text{PSkyline}_1\ (D)$ eventually ends up with an invocation of SMerge with two skyline sets S' and S'' such that $S' = \mathcal{S}(D')$ and $S'' = \mathcal{S}(D'')$ where $D = D' \cup D''$. If the size of a skyline set grows with the size of its dataset, this last invocation of SMerge is likely to be the most costly among all invocations of SMerge, yet it cannot take advantage of parallel computing.

This observation leads to another definition of PSkyline that uses a sequential version of PReduce but a parallel version of SMerge. Assuming an auxiliary function PMerge (Parallel Merge) that performs a parallel computation to merge two skyline sets, we obtain another definition of PSkyline that uses SReduce and PMerge:

$$\text{PSkyline}_2\ (D) \stackrel{\text{def}}{=} \text{SReduce}\ (\text{PMerge},\ (\text{PMap}\ (\text{SSkyline},\ \{D_1, \cdots, D_b\}))) \quad \text{where } D = D_1 \cup \cdots \cup D_b$$

Now we have a definition of PSkyline that takes full advantage of parallel computing and use this second definition PSkyline_2 as our parallel skyline algorithm.

Algorithm 7 describes the pseudocode of PSkyline. PSkyline first divides D linearly into $D_1, \cdots, D_b$ of the same size (line 1). Then it invokes PMap which is implemented using the **parallel for** construct; lines 2–4 implement PMap (SSkyline, $\{D_1, \cdots, D_b\}$) which computes the local skyline set S_i of each block D_i.

Algorithm 8 PMerge $(S_1,\ S_2)$

```
 1: T1 ← S1
 2: T2 ← φ
 3: f (y) ≝ begin
 4:             for each x ∈ T1 do
 5:               if y ≺ x then
 6:                 T1 ← T1 − {x}
 7:               else if x ≺ y then
 8:                 return
 9:               end if
10:             end for
11:             T2 ← T2 ∪ {y}
12:           end
13: parallel for each z ∈ S2 do
14:   f (z)
15: end parallel for
16: return T1 ∪ T2
```

Then lines 6–8 merge local skyline sets sequentially using PMerge. Below we describe our implementation of PMerge.

6.3. Parallel skyline merging

PMerge is a function taking two skyline sets S_1 and S_2 and returning the skyline set of their union $S_1 \cup S_2$:

$$\text{PMerge}\ (S_1,\ S_2) \stackrel{\text{def}}{=} \mathcal{S}(S_1 \cup S_2) \quad \text{where } \mathcal{S}(S_1) = S_1 \text{ and } \mathcal{S}(S_2) = S_2$$

First we describe an implementation of PMerge that uses sets. Then we describe an equivalent implementation that uses arrays.

Algorithm 8 describes an implementation of PMerge that uses sets. Given two local skyline sets S_1 and S_2, it creates another two local sets T_1 and T_2 (lines 1–2). Then it defines a local function f (lines 3–12) and applies f to each element z of S_2 in parallel (lines 13–15). Finally it returns $T_1 \cup T_2$ as its result (line 16).

T_1 is initialized with S_1 (line 1) and decreases its size by eliminating non-skyline tuples in S_1 (line 6). T_2 is initialized with an empty set (line 2) and increases its size by admitting skyline tuples from S_2 (line 11). Eventually T_1 eliminates all non-skyline tuples in S_1 to become $S_1 \cap \mathcal{S}(S_1 +\!+ S_2)$, and T_2 admits all skyline tuples from S_2 to become $S_2 \cap \mathcal{S}(S_1 +\!+ S_2)$.

Given a tuple y as input, the local function f updates T_1 and T_2 by performing dominance tests between y and tuples in T_1. It keeps eliminating from T_1 those tuples dominated by y (lines 5–6), but as soon as it locates a tuple in T_1 that dominates y, it terminates (lines 7–8). If no tuple in T_1 dominates y, it appends y to T_2 (line 11). Note that f only updates T_1 and T_2 and returns no interesting result.

PMerge applies f to each tuple in S_2 in parallel (line 13). Since multiple invocations of f attempt to update T_1 and T_2 simultaneously, we assume that assignments to T_1 and T_2 in lines 6 and 11 are atomic operations. Note that an invocation of f may observe changes in T_1 made by other parallel invocations of f. These changes are safe because T_1 never grows and the loop in f never considers the same tuple more than once. In fact, it is because parallel updates to T_1 are allowed that all invocations of f cooperate with each other.

The following theorem states the correctness of PMerge:

Theorem 6.1. *If $\mathcal{S}(S_1) = S_1$ and $\mathcal{S}(S_2) = S_2$, PMerge $(S_1,\ S_2)$ returns $\mathcal{S}(S_1 \cup S_2)$.*

Proof. See Appendix. □

It is important that because of frequent updates to T_1 and T_2 by parallel invocations of f, we intend to use PMerge only on multicore architectures. On distributed architectures, for example, an update to T_1 in line 6 or T_2 in line 11 may be accompanied by communications to other nodes in the network and its communication cost is likely to outweigh the benefit of parallel computing. On multicore architectures, such an update incurs only a single write to the main memory and thus can be implemented at a relatively low (or almost negligible) cost.

Now we rewrite the implementation in Algorithm 8 by storing all datasets in arrays. For S_1 and S_2, we use two arrays of tuples which are assumed to fit in the main memory. Since no assignment to S_1 occurs and T_1 always holds a subset of S_1, we represent T_1 as an array F_1 of boolean flags such that $F_1[i] = \mathbf{true}$ if and only if $S_1[i] \in T_1$. Then every element of F_1 is initialized with $\mathbf{true}$ and the assignment to T_1 in line 6 changes to a single write of $\mathbf{false}$ to a certain element of F_1. Since every update to F_1 writes the same boolean value $\mathbf{false}$, the order of writes to the same element of F_1 does not matter, which implies that all updates to F_1 are effectively atomic operations. Similarly we represent T_2 as an array F_2 whose elements are initialized with $\mathbf{false}$ and implement the assignment to T_2 in line 11 as a write of $\mathbf{true}$ to an element of F_2.

In order to store the array for $T_1 \cup T_2$, we reuse the two arrays allocated for S_1 and S_2. Thus the new implementation of PMerge uses an in-place algorithm

except that it allocates two fresh arrays F_1 and F_2.

6.4. Overall implementation

Our parallel skyline algorithm PSkyline proceeds in two steps: first a parallel application of the sequential skyline algorithm SSkyline and second a sequential application of the parallel skyline merging algorithm PMerge. Below we describe how PSkyline computes the skyline set of a dataset D of dimensionality d. We assume that a total of c cores are available.

In the first step, PSkyline loads the dataset D in the main memory at once and divides it into b blocks $D_1, \cdots, D_b$. Then it applies SSkyline to each block in parallel; the b resultant skyline sets $S_1, \cdots, S_b$ are kept in the main memory. In the second step, PSkyline applies PMerge to a pair of skyline sets a total of $b-1$ times. It begins by reading S_1 from the main memory. We let $S'_1 = \mathcal{S}(S_1) = S_1$. At the i-th invocation where $1 \leq i \leq b-1$, PMerge reads S_{i+1} and computes $S'_{i+1} = \mathcal{S}(S'_i \cup S_{i+1})$. The memory usage (in terms of the number of tuples) is no higher than $|D|(1+\frac{1}{d})$ which is reached when all tuples are skyline tuples. Here $|D|\frac{1}{d}$ accounts for two arrays of boolean flags whose combined length is $|D|$. The memory usage is also no lower than $|\mathcal{S}(D)|(1+\frac{1}{d})$ because PMerge eventually stores the entire skyline set in the main memory. As a special case, if we set b to c, the memory usage always lies between $|D|$ and $|D|(1+\frac{1}{d})$.

6.5. Comparisons with MapReduce

The idea of basic constructs such as PMap, SReduce, and PMerge in PSkyline is already in use by such database programming models as MapReduce [7, 23] and Map-Reduce-Merge [34]. Parallel constructs in these programming models, however, are actually different from PMap, SReduce, and PMerge. For example, the map construct in MapReduce takes as input a single key/value pair and returns as output a list of key/value pairs, whereas PMap takes as input a unary function and a set and returns as output another set. Although PSkyline can be implemented using MapReduce, our implementation using OpenMP is much simpler since it specifies only high-level compiler directives to parallelize loops (see Section 2.2).

7. Experimental results

This section presents experimental results of running parallel BBS, parallel SFS, parallel SSkyline, and PSkyline on a sixteen-core machine. We are primarily interested in the speedup of each individual algorithm when it exploits multiple cores. We do not consider the relative performance of all the algorithms because it

is hard to ensure fair comparisons. For example, parallel BBS is a disk-based algorithm which dynamically loads R-tree nodes as necessary, whereas all the other algorithms load the entire dataset into the main memory at once. The memory usage of PSkyline also depends on both the number of blocks and cores. Thus we focus on analyzing the speed of each parallel algorithm to see how good it is at utilizing multi-core architectures.

We implement parallel SFS, parallel SSkyline, and PSkyline in C. For BBS, we parallelize the implementation, written in C++, provided by the authors of [28]. We reimplement SFS as a main-memory algorithm loading the entire dataset into the main memory, and then parallelize it. (SFS is originally proposed as a disk-based algorithm [3].) Our implementation of parallel SFS presorts a dataset according to a monotone preference function based on entropy [3].

7.1. Experiment setup

The experiments use both synthetic datasets and real datasets. We generate synthetic datasets according to either independent distributions or anti-correlated distributions [2]. For datasets based on independent distributions, we independently generate all tuple elements using a uniform distribution. For datasets based on anti-correlated distributions, we generate tuples in such a way that a good tuple element in one dimension is likely to indicate the existence of bad elements in other dimensions. We use two real datasets, Household and NBA, which follow independent and anti-correlated distributions, respectively. All tuple elements are restricted to the unit interval $[0.0, 1.0)$ and represented as floating-point numbers of 4 bytes.

A dataset D determines two parameters d and n:

- d denotes the dimensionality of the dataset.
- n denotes the number of tuples in the dataset, or its size. When reporting the number of tuples, we use K for 1024 (*e.g.*, $100K = 102400$).

We use two groups of synthetic datasets based on independent and anti-correlated distributions. The first group varies the dimensionality of the dataset: we use $d = 4$, 6, 8, 10, 12, 14, 16 and $n = 100K$ to obtain $2 * 7 = 14$ datasets, all of the same size. The second group varies the number of tuples in the dataset: we use $d = 10$ and $n = 10K$, $50K$, $100K$, $500K$, $1000K$, $5000K$ to obtain $2 * 6 = 12$ datasets, all with the same dimensionality. (Thus we use a total of $14+12-2 = 24$ synthetic datasets.)

Given a dataset, we specify each experiment with a single parameter c:

- c denotes the number of cores participating in parallel skyline computation.

For each dataset, we try $c = 1, 4, 8, 16$. Note that even when c is set to 1, parallel BBS, parallel SFS, and parallel SSkyline do not degenerate to their sequential counterparts because of the use of accumulators or flag arrays.

We run all experiments on a Dell PowerEdge server with four quad-core Intel Xeon 1.6GHz CPUs (a total of sixteen cores) and 8 gigabytes of main memory. The disk page size is 4 kilobytes. We compile all the algorithms using gcc 4.2.4 with -O3 option. In all experiments, the main memory is large enough to hold all data (both source data and intermediate results) manipulated by each algorithm. As the main performance metric, we use the elapsed time T measured in wall clock seconds. We write T_c for the elapsed time when c cores are used. For parallel BBS, we do not include the time for building R-trees from datasets; for parallel SFS, we include the time for presorting datasets. All measurements are averaged over 10 sample runs.

For PSkyline, we need to choose the number b of blocks into which the dataset is divided in the first step. Too low a value for b diminishes the benefit of parallel computation during the first step (a parallel application of SSkyline). For example, setting b to 1 results in a sequential skyline computation regardless of the number of cores available because PMerge is never invoked. Too high a value for b, on the other hand, incurs an unnecessarily high cost of spawning threads. In the extreme case of $b = n$, a total of n threads are spawned only to find singleton skyline sets. After experimentally determining the relation between b and the performance of PSkyline, we choose to set b to c, in which case the speedup is usually maximized.

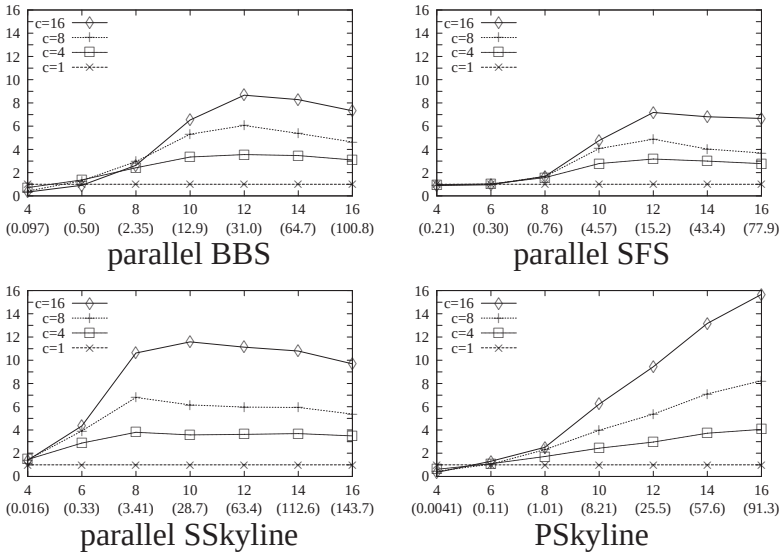
7.2. Effect of different numbers of cores on the speed

The first set of experiments test the effect of using multiple cores on the speed of parallel BBS, parallel SFS, parallel SSkyline, and PSkyline.

7.2.1. Effect of dimensionality and dataset size

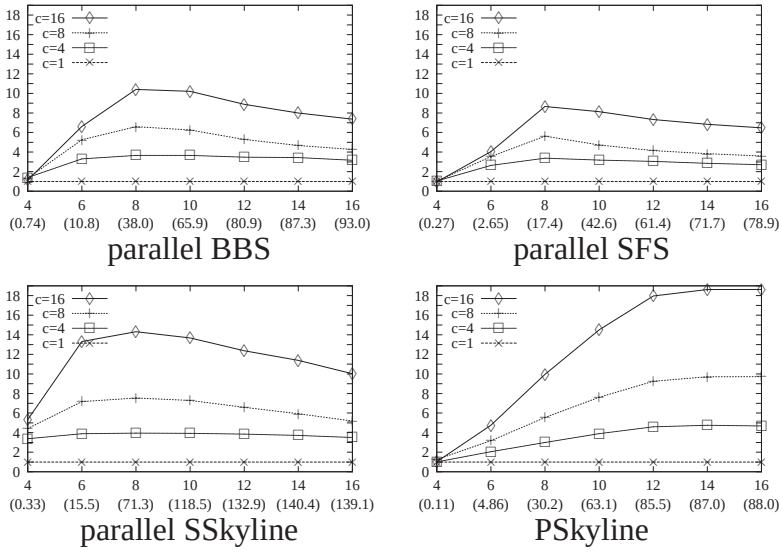
Figure 1 shows the speedup relative to the case of $c = 1$ when d, the dimensionality of the dataset, varies from 4 to 16, while n is set to $100K$. For each dataset, it shows the elapsed time T_1 for the case of $c = 1$ in parentheses; the tables show the density of skyline tuples for each dataset. Parallel BBS, parallel SFS, and parallel SSkyline, all of which are originally proposed as sequential skyline algorithms, exhibit a similar pattern of increase in speedup: the speedup increases to a certain point, far less than the number of cores, and then remains stable or gradually decreases as d increases. For example, both parallel BBS and parallel

dimensionality	4	6	8	10	12	14	16
skyline tuples (%)	0.35	2.60	8.95	26.18	44.32	66.08	82.09



(1) Independent datasets

dimensionality	4	6	8	10	12	14	16
skyline tuples (%)	2.01	19.47	51.75	74.88	82.81	83.86	83.36



(2) Anti-correlated datasets

Figure 1: Speedup $\frac{T_1}{T}$ $(n = 100K)$

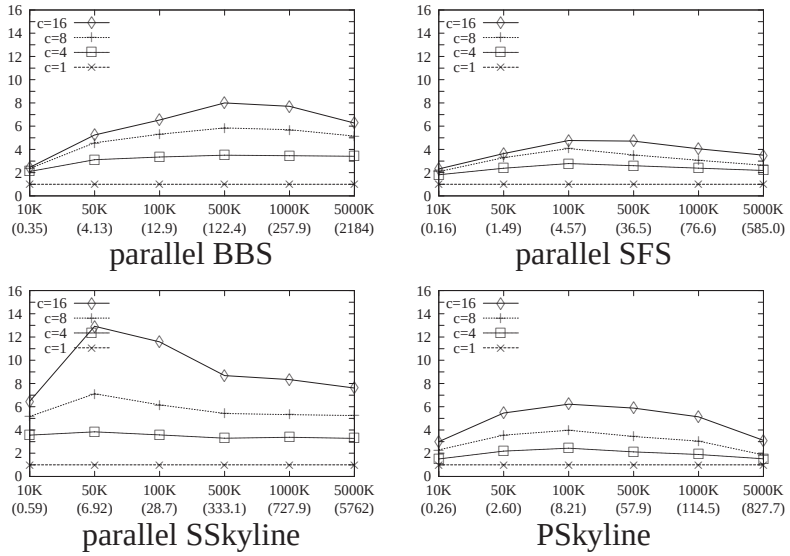
SFS achieve the maximum speedup when $d = 12$ for independent datasets and when $d = 8$ for anti-correlated datasets; parallel SSkyline achieves the maximum speedup when $d = 10$ for independent datasets and when $d = 8$ for anti-correlated datasets. In contrast, the speedup of PSkyline increases towards c as d increases. Thus PSkyline better utilizes multiple cores for high-dimensional datasets. In particular, for independent datasets, PSkyline achieves a speedup close to c as d increases toward 16. For anti-correlated datasets with $d \geq 12$, PSkyline achieves a superlinear speedup of more than c, which we explain in detail in Section 7.2.2.

Note that for a few datasets with low dimensionalities (independent datasets with $d = 4, 6$ and anti-correlated dataset with $d = 4$), an increase in c sometimes results in a decrease in the speedup. For such datasets with relatively few skyline tuples, the overhead of spawning multiple threads and performing synchronization can outweigh the gain from an increase in the number of cores. In particular, an increase of c from 4 to 8 and from 8 to 16 means that we use not just multiple cores but multiple CPUs (two and four CPUs, respectively), in which case the overhead is much higher.

Among parallel BBS, parallel SFS, and parallel SSkyline, it is parallel SSkyline that achieves the best speedup for all datasets. This is because both parallel BBS and parallel SFS accumulate multiple candidate skyline tuples that are incomparable with each other before performing parallel sections, which is inherently a sequential execution. The portion of such a sequential execution is much smaller in parallel SSkyline (only lines 1, 2, and 24 in Algorithm 6); thus parallel SSkyline is more suitable for parallelization than parallel BBS and parallel SFS. We remark, however, that although parallel SSkyline achieves a better speedup than parallel BBS and parallel SFS, it is the slowest for most high-dimensional datasets. For example, for the anti-correlated dataset with $d = 8$, when $c = 16$, the speedup of parallel SSkyline is 14.3 while it takes 4.98 seconds to compute the whole skyline tuples. The speedup of parallel BBS and parallel SFS is approximately 10 and 8, respectively, while their elapsed time is 3.65 and 2.01 seconds, respectively.

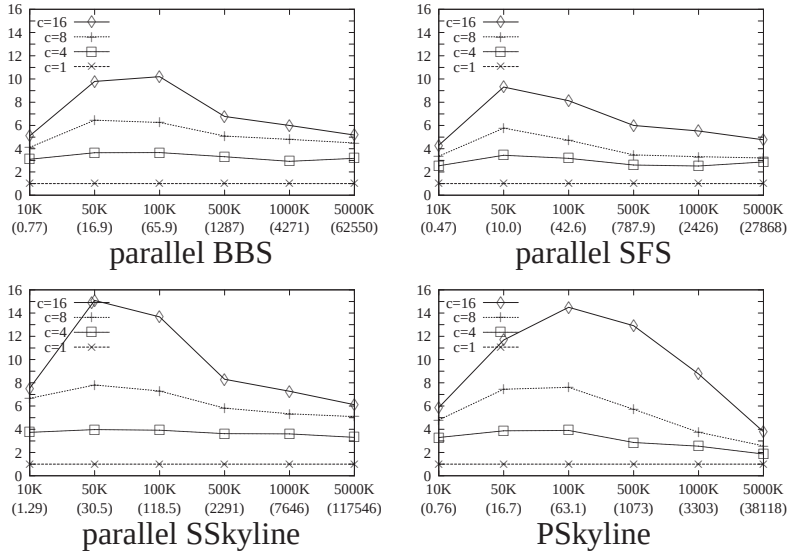
Figure 2 shows the speedup relative to the case of $c = 1$ when n, the number of tuples in the dataset, varies from $10K$ to $1000K$, while d is set to 10. As in Figure 1, for each dataset, it shows the elapsed time T_1 for the case of $c = 1$ in parentheses; the tables show the density of skyline tuples for each dataset. For this second group of datasets, all the algorithms exhibit a similar pattern of increase in speedup: the speedup increases to a certain point, less than the number of cores, and then remains stable or gradually decreases as n increases. Overall an increase of c from 4 to 8 and 8 to 16 does not necessarily result in a perfor-

dataset size	10K	50K	100K	500K	1000K	5000K
skyline tuples (%)	46.31	29.76	26.18	13.73	9.57	5.08



(1) Independent datasets

dataset size	10K	50K	100K	500K	1000K	5000K
skyline tuples (%)	85.91	78.97	74.88	59.97	52.36	35.44



(2) Anti-correlated datasets

Figure 2: Speedup $\frac{T_1}{T}$ ($d = 10$)

mance improvement commensurate with the increase in c, as is the case for most independent datasets. Note that there is no particular relation between the dataset size and the speedup resulting from using multiple cores. In fact, as will be seen in Section 7.2.2, the speedup depends more on the density of skyline tuples than on the type of dataset (either independent or anti-correlated) or its size.

From these experiments, we draw the following conclusion:

- In general, parallel BBS, parallel SFS, parallel SSkyline, and PSkyline all successfully utilize multiple cores and achieve a reasonable speedup as the number of cores increases.

7.2.2. Effect of the density of skyline tuples

We expect that a higher density of skyline tuples gives a better utilization of multiple cores by parallel BBS, parallel SFS, parallel SSkyline and PSkyline. Here we experimentally test the relation between the speedup and the density of skyline tuples using all the datasets from the previous experiments. For each dataset, we calculate $\frac{T_1}{T_{16}}$, the speedup from using sixteen cores relative to the case of $c = 1$, and the density of skyline tuples.

Figure 3 shows the speedup versus the density of skyline tuples. We observe a correlation between the speedup and the density of skyline tuples: there is little performance gain when most tuples are non-skyline tuples, whereas a relatively high speedup results when most tuples are skyline tuples. Moreover there is no particular relation between the speedup and the type of the dataset (either independent or anti-correlated). For example, PSkyline achieves a speedup of more than 15 for some independent dataset while it fails to achieve a speedup of 2 for some anti-correlated dataset.

Note that PSkyline is the most sensitive to the density of skyline tuples. When the density of skyline tuples is high, most of the elapsed time is spent on dominance tests between skyline tuples. Assume that all tuples are skyline tuples, and a total of s skyline tuples are equally distributed across c data blocks, where c is the number of cores. During the map phase of PSkyline, each block contains $\frac{s}{c}$ skyline tuples and each thread performs $\frac{\frac{s}{c}(\frac{s}{c}+1)}{2} = \frac{s^2+sc}{2c^2}$ dominance tests. During the reduce phase, PSkyline uses PMerge to combine c skyline sets where each set contains $\frac{s}{c}$ tuples. Since PMerge fully exploits multiple cores, PSkyline completes the reduce phase in the following number of parallel steps:

$$\frac{\frac{s}{c}\cdot\frac{s}{c}}{c} + \frac{\frac{2s}{c}\cdot\frac{s}{c}}{c} + \cdots + \frac{\frac{(c-1)s}{c}\cdot\frac{s}{c}}{c} = \frac{s^2+2s^2+\cdots+(c-1)s^2}{c^3} = \frac{s^2\cdot\frac{c(c-1)}{2}}{c^3} = \frac{s^2c-s^2}{2c^2}$$

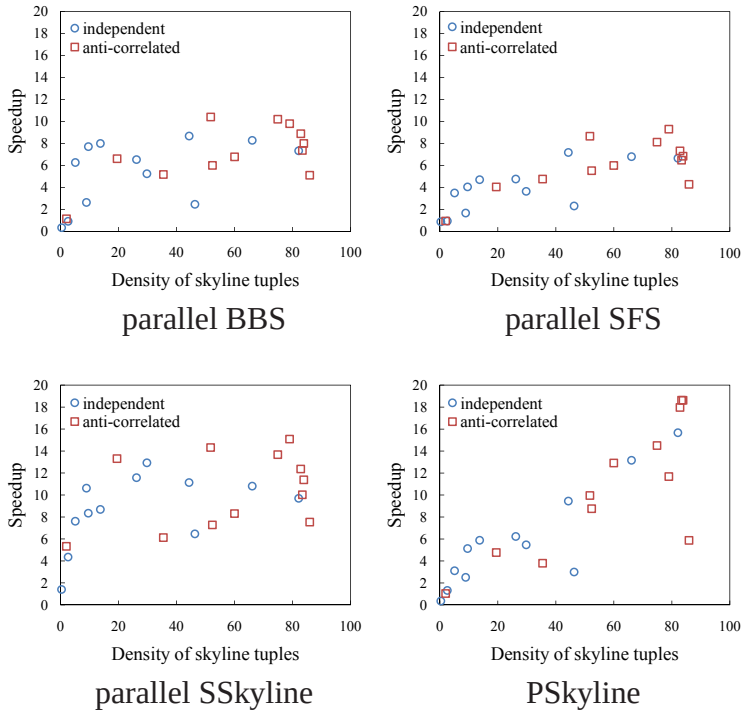


Figure 3: Speedup $\frac{T_1}{T_{16}}$

Hence PSkyline computes all skyline tuples in $\frac{s^2+sc}{2c^2} + \frac{s^2c-s^2}{2c^2} = \frac{s(s+1)}{2c}$ parallel steps. Since PSkyline computes all skyline tuples in $\frac{s(s+1)}{2}$ sequential steps when $c = 1$, we conclude that the speedup of PSkyline increases towards c as the density of skyline tuples increases. In fact, for some anti-correlated datasets with more than 80% of skyline tuples, PSkyline sometimes achieves a superlinear speedup of more than $c = 16$ because of the effect of using cache during the map phase: a thread considers the whole dataset when $c = 1$, whereas it considers only $\frac{1}{16}$ of the dataset when $c = 16$, thus resulting in more cache hits on average. (In our experiments, each core has its own 2 megabyte cache.)

From these experiments, we draw the following conclusion:

- The speedup for parallel BBS, parallel SFS, parallel SSkyline, and PSkyline is mainly driven by the density of skyline tuples. In particular, all the algorithms are the most effective when most tuples are skyline tuples, or when it needs a parallel computation the most.

Dataset	parallel BBS		parallel SFS		parallel SSkyline		PSkyline	
	$c = 1$	$c = 16$	$c = 1$	$c = 16$	$c = 1$	$c = 16$	$c = 1$	$c = 16$
Household	0.3293	0.2662	0.5823	0.4377	1.5784	0.1939	0.4698	0.1325
NBA	0.0294	0.0418	0.0674	0.0568	0.1076	0.0341	0.0414	0.0427

Figure 4: Results on real datasets

7.3. Real datasets

The second set of experiments test parallel BBS, parallel SFS, parallel SSkyline, and PSkyline on two real datasets Household and NBA. Household is a six-dimensional dataset following an independent distribution ($n = 127931$ with 5774 skyline tuples) where each tuple records the percentage of an American family's annual income spent on six types of expenditures (`http://www.ipums.org`). NBA is an eight-dimensional dataset following an anti-correlated distribution ($n = 17264$ with 1796 skyline tuples) where each tuple records the statistics of an NBA player's performance in eight categories (`http://www.nba.com`). For each algorithm, we try $c = 1, 16$.

Figure 4 shows the results on the real datasets. For Household, when using sixteen cores, parallel BBS, parallel SFS, and PSkyline achieve only a speedup of 1.23, 1.33, and 3.55, respectively, while parallel SSkyline achieves a moderate speedup of 8.14. Note that for Household the density of skyline tuples is low, *i.e.*, 4.36%, which implies a relatively small speedup as explained in Section 7.2.2. The moderate speedup of parallel SSkyline partly results from the fact that it is much slower than other algorithms when $c = 1$. For NBA, only parallel SFS and parallel SSkyline achieve a small speedup (1.19 and 3.16, respectively), while parallel BBS and PSkyline even deteriorate in speed. The reason is that the dataset size is small and hence the overhead of spawning multiple threads outweighs the benefit from an increase in the number of cores. To conclude, parallel skyline algorithms achieve only a small speedup when using multiple cores on small datasets and datasets with a low density of skyline tuples on which the sequential algorithms already run fast enough.

7.4. Scalability of the proposed algorithms

We expect that when more cores are available (for example 32 and 64), the speedup of each parallel algorithm still increases with the number of cores. Especially when the density of skyline tuples is high, the speedup of each algorithm increases towards the number of cores. To see why, assume that all tuples are skyline tuples. Then all the algorithms compute a total of s skyline tuples in $\frac{s(s+1)}{2}$

sequential steps when only a single core is available. (In the case of parallel BBS, we ignore dominance tests involving intermediate nodes.) When a total of c cores are available and all skyline tuples are equally distributed across c data blocks, PSkyline computes all skyline tuples in $\frac{s(s+1)}{2c}$ parallel steps as described in Section 7.2.2. As for parallel SSkyline, each core equally performs the following number of dominance tests:

$$(s-1)+\cdots+(s-\tfrac{s}{c}) = \tfrac{\frac{s}{c}(s-\frac{s}{c}+s-1)}{2} = \tfrac{s(2s-\frac{s}{c}-1)}{2c}$$

Hence parallel SSkyline computes all skyline tuples in $\frac{s(2s-\frac{s}{c}-1)}{2c}$ parallel steps. Parallel SFS sequentially accumulates γ incomparable tuples before performing dominance tests in parallel. If γ divides into s, parallel SFS performs $\frac{\gamma(\gamma-1)}{2} \cdot \frac{s}{r} = \frac{s(\gamma-1)}{2}$ sequential steps and the following number of parallel steps:

$$\tfrac{\gamma\cdot\gamma}{c} + \tfrac{2\gamma\cdot\gamma}{c} + \cdots + \tfrac{(\frac{s}{\gamma}-1)\gamma\cdot\gamma}{c} = \tfrac{\gamma^2}{c}\{1+2+\cdots+(\tfrac{s}{\gamma}-1)\} = \tfrac{\gamma^2}{c} \cdot \tfrac{(\frac{s}{\gamma}-1)\frac{s}{\gamma}}{2} = \tfrac{s(s-\gamma)}{2c}$$

Similarly, if parallel BBS sequentially accumulates β incomparable tuples, it computes all skyline tuples in $\frac{s(\beta-1)}{2}$ sequential steps and $\frac{s(s-\beta)}{2c}$ parallel steps. Note that unlike parallel SSkyline and PSkyline, the elapsed time of parallel BBS and parallel SFS is bounded by $\frac{s(\beta-1)}{2}$ and $\frac{s(\gamma-1)}{2}$ sequential steps respectively, regardless of the number of available cores.

8. Conclusion

The time is ripe for a marriage between database operations and multicore architectures. We investigate parallel skyline computation as a case study of parallelizing database operations on multicore architectures. We believe that opportunities abound for parallelizing other database operations for multicore architectures, for which our work may serve as a future reference.

References

[1] I. Bartolini, P. Ciaccia, and M. Patella. Efficient sort-based skyline evaluation. *ACM Transactions on Database Systems*, 33(4):1–49. 2008.

[2] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *Proceedings of the 17th International Conference on Data Engineering*, pages 421–430. IEEE Computer Society, 2001.

[3] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. In *Proceedings of the 19th International Conference on Data Engineering*, pages 717–816. IEEE Computer Society, 2003.

[4] A. Cosgaya-Lozano, A. Rau-Chaplin, and N. Zeh. Parallel computation of skyline queries. In *Proceedings of the 21st Annual International Symposium on High Performance Computing Systems and Applications*, pages 12. IEEE Computer Society, 2007.

[5] L. Dagum and R. Menon. OpenMP: An Industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55. 1998.

[6] S. Das, S. Antony, D. Agrawal, and A. E. Abbadi. Thread cooperation in multicore architectures for frequency counting over multiple data streams. *Proceedings of the VLDB Endowment (PVLDB)*, 2(1):217–228, 2009.

[7] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation*, pages 137–150. USENIX Association, 2004.

[8] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel geometric algorithms for coarse grained multicomputers. In *Proceedings of the 9th Annual Symposium on Computational Geometry*, pages 298–307. ACM, 1993.

[9] P. Godfrey, R. Shipley, and J. Gryz. Maximal vector computation in large data sets. In *Proceedings of the 31st International Conference on Very Large Data Bases*, pages 229–240. VLDB Endowment, 2005.

[10] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 47–57. ACM, 1984.

[11] W.-S. Han, W. Kwak, J. Lee, G. M. Lohman, and V. Markl. Parallelizing query optimization. *Proceedings of the VLDB Endowment (PVLDB)*, 1(1):188–200, 2008

[12] W.-S. Han and J. Lee. Dependency-aware reordering for parallelizing query optimization in multi-core cpus. In *Proceedings of the 35th SIGMOD International Conference on Management of Data*, pages 45–58. ACM, 2009.

[13] C. Kim, E. Sedlar, J. Chhugani, T. Kaldewey, A. D. Nguyen, A. D. Blas, V. W. Lee, N. Satish, and P. Dubey. Sort vs. Hash revisited: Fast join implementation on modern multi-core CPUs. *Proceedings of the VLDB Endowment (PVLDB)*, 2(2):1378–1389, 2009

[14] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: an online algorithm for skyline queries. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 275–286. VLDB Endowment, 2002.

[15] H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *Journal of the ACM*, 22(4):469–476, 1975.

[16] K. C. K. Lee, B. Zheng, H. Li, and W.-C. Lee. Approaching the skyline in Z order. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, pages 279–290. VLDB Endowment, 2007.

[17] L. Liu, E. Li, Y. Zhang, and Z. Tang. Optimization of frequent itemset mining on multiple-core processor. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, pages 1275–1285. VLDB Endowment, 2007.

[18] J. Matousek. Computing dominances in E^n. *Information Processing Letters*, 38(5):277–278, 1991.

[19] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *ACM Transactions on Database Systems*, 30(1):41–82, 2005.

[20] S. Park, T. Kim, J. Park, J. Kim, and H. Im. Parallel skyline computation on multicore architectures. In *Proceedings of the 22nd International Conference on Data Engineering*. IEEE Computer Society, 2009.

[21] S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[22] F. Ramsak, V. Markl, R. Fenk, M. Zirkel, K. Elhardt, and R. Bayer. Integrating the UB-tree into a database system kernel. In *Proceedings of the 26th International Conference on Very Large Data Bases*, pages 263–272. Morgan Kaufmann Publishers Inc., 2000.

[23] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24. IEEE Computer Society, 2007.

[24] J. Selke, C. Lofi, and W.-T. Balke. Highly scalable multiprocessing algorithms for preference-based database retrieval. In *Proceedings of the 15th International Conference on Database Systems for Advanced Applications (DASFAA)*, pages 246–260, 2010.

[25] I. Stojmenovic and M. Miyakawa. An optimal parallel algorithm for solving the maximal elements problem in the plane. *Parallel Computing*, 7(2):249–251, 1988.

[26] H. Sutter The free lunch is over: a fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), Mar. 2005.

[27] H. Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.

[28] Y. Tao, X. Xiao, and J. Pei. Efficient skyline and top-k retrieval in subspaces. *IEEE Transactions on Knowledge and Data Engineering*, 19(8):1072–1088, 2007.

[29] S. Tatikonda and S. Parthasarathy. Mining tree-structured data on multicore systems. *Proceedings of the VLDB Endowment (PVLDB)*, 2(1):694–705, 2009.

[30] R. Torlone and P. Ciaccia. Finding the best when it's a matter of preference. In *Proceedings of the 10th Italian Symposium on Advanced Database Systems (SEBD)*, pages 347–360, 2002.

[31] A. Vlachou, C. Doulkeridis, and Y. Kotidis. Angle-based space partitioning for efficient parallel skyline computation. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 227–238. ACM, 2008.

[32] S. Wang, B. C. Ooi, A. K. H. Tung, and L. Xu. Efficient skyline query processing on peer-to-peer networks. In *Proceedings of the 23rd International Conference on Data Engineering*, pages 1126–1135. IEEE, 2007.

[33] P. Wu, C. Zhang, Y. Feng, B. Y. Zhao, D. Agrawal, and A. E. Abbadi. Parallelizing skyline queries for scalable distribution. In *Proceedings of the 10th International Conference on Extending Database Technology*, pages 112–130, 2006.

[34] H. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-Reduce-Merge: simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1029–1040. ACM, 2007.

[35] S. Zhang, N. Mamoulis, and D. W. Cheung. Scalable skyline computation using object-based space partitioning. In *Proceedings of the 35th SIGMOD International Conference on Management of Data*, pages 483–494. ACM, 2009.

Appendix

Proof of Proposition 5.1

1. $D[head \dots tail] \subset D_{inner}$.
2. $D[head] \prec\!\succ D[(head+1) \dots (i-1)]$.
3. $\mathcal{S}(D_{inner}) = \mathcal{S}(D[head \dots tail])$.

Proof. We prove each invariant by induction on the iterations of the inner loop. For Invariants 1 and 3, the base case holds because at the beginning of the first iteration, D_{inner} is the value of $D[head \dots tail]$. For Invariant 2, the base case holds because $D[(head+1) \dots (i-1)]$ is empty ($i - 1 = head$ from line 4).

For the inductive case, we assume that each invariant holds at the $(k-1)$-st iteration, and analyze three possible subcases: $D[head] \prec D[i]$, $D[i] \prec D[head]$, and $D[head] \prec\!\succ D[i]$. Below we give the proof for the inductive case of each invariant.

Proof of Invariant 1. Let D_{k-1} be the value of $D[head \dots tail]$ at the beginning of the $(k-1)$-st iteration, where $D_{k-1} \subset D_{inner}$ by induction hypothesis. If $D[head] \prec D[i]$, D_k is the same as D_{k-1} except that it excludes the value of $D[i]$ from D_{k-1}. Likewise, if $D[i] \prec D[head]$, D_k is the same as D_{k-1} except that it excludes the value of $D[head]$ from D_{k-1}. Otherwise D_k is exactly the same as D_{k-1}. Therefore $D_k \subset D_{k-1}$ and from the assumption, $D_k \subset D_{inner}$. □

Proof of Invariant 2. Assume that the invariant holds at the $(k-1)$-st iteration: $D[head] \prec\succ D[(head+1)\ldots(i-1)]$. If $D[head] \prec D[i]$, then $D[(head+1)\ldots(i-1)]$ does not change. Therefore, from the assumption, the invariant holds. If $D[i] \prec D[head]$, then the invariant holds because $D[(head+1)\ldots(i-1)]$ is empty ($i-1 = head$ from line 13). Otherwise $D[head] \prec\succ D[i]$. After i increases by 1, $D[head] \prec\succ D[i]$ is rewritten as $D[head] \prec\succ D[i-1]$, and the assumption is rewritten as $D[head] \prec\succ D[(head+1)\ldots(i-2)]$. Hence $D[head] \prec\succ D[(head+1)\ldots(i-1)]$. □

Proof of Invariant 3. Let D_{k-1} be the value of $D[head\ldots tail]$ at the beginning of the $(k-1)$-st iteration, where $\mathcal{S}(D_{inner}) = \mathcal{S}(D_{k-1})$ by induction hypothesis. Inside the inner loop, if $D[head] \prec D[i]$ or $D[i] \prec D[head]$, then only the value of the dominated tuple is removed from D_{k-1}, which cannot be a skyline tuple. Otherwise D_k is the same as D_{k-1}. In either way, $\mathcal{S}(D_k) = \mathcal{S}(D_{k-1})$ and from the assumption, $\mathcal{S}(D_k) = \mathcal{S}(D_{inner})$. □

□

Proof of Proposition 5.2

4. $D[1\ldots(head-1)] \prec\succ D[head\ldots tail]$.
5. $\mathcal{S}(D_{outer}) = \mathcal{S}(D[1\ldots tail])$.
6. $\mathcal{S}(D[1\ldots(head-1)]) = D[1\ldots(head-1)]$.

Proof. We prove each invariant by induction on the iterations of the outer loop. For Invariants 4 and 6, the base case holds because at the beginning of the first iteration, $D[1\ldots(head-1)]$ is empty ($head = 1$ from line 1). For Invariant 5, the base case holds because at the beginning of the first iteration, $D_{outer} = D[1\ldots tail]$.

Below we give the proof for the inductive case of each invariant.

Proof of Invariant 4. Assume that the invariant holds at the $(k-1)$-st iteration: $D[1\ldots(head-1)] \prec\succ D[head\ldots tail]$. Let D_{inner} be the value of $D[head\ldots tail]$ before the inner loop starts. Then, by Invariant 1, $D[1\ldots(head-1)] \prec\succ D[head\ldots tail]$ still holds after the inner loop terminates. Together with Invariant 2, $D[1\ldots head] \prec\succ D[(head+1)\ldots tail]$. After $head$ increases by 1, $D[1\ldots(head-1)] \prec\succ D[head\ldots tail]$ at the beginning of the k-th iteration. □

Proof of Invariant 5. Let S_{k-1} be $\mathcal{S}(D[1\ldots tail])$ at the beginning of the $(k-1)$-st iteration. As an induction hypothesis, assume that the invariant holds at the

$(k-1)$-st iteration; then $\mathcal{S}(D_{outer}) = S_{k-1}$. We will show $S_k = S_{k-1}$. Note $D[1 \ldots tail] = D[1 \ldots (head-1)] \cup D[head \ldots tail]$. Then, by Invariant 4 and Proposition 2.2, $S_{k-1} = \mathcal{S}(D[1 \ldots (head-1)]) \cup \mathcal{S}(D[head \ldots tail])$. Let D_{inner} be the value of $D[head \ldots tail]$ before the inner loop starts. Then $S_{k-1} = \mathcal{S}(D[1 \ldots (head-1)]) \cup \mathcal{S}(D_{inner})$. By Invariant 3, $S_{k-1} = \mathcal{S}(D[1 \ldots (head-1)]) \cup \mathcal{S}(D[head \ldots tail])$ after the inner loop terminates. (Note that at this point, D_{inner} may differ from the value of $D[head \ldots tail]$.) By Invariants 2 and 4, and Proposition 2.2, $S_{k-1} = \mathcal{S}(D[1 \ldots (head-1)]) \cup \mathcal{S}(D[head]) \cup \mathcal{S}(D[(head+1) \ldots tail]) = \mathcal{S}(D[1 \ldots head]) \cup \mathcal{S}(D[(head+1) \ldots tail])$. After $head$ increases by 1, by Invariant 4 and Proposition 2.2, $S_{k-1} = \mathcal{S}(D[1 \ldots tail])$ which is S_k at the beginning of the k-th iteration. □

Proof of Invariant 6. As an induction hypothesis, assume that the invariant holds at the $(k-1)$-st iteration: $\mathcal{S}(D[1 \ldots (head-1)]) = D[1 \ldots (head-1)]$. Let D_{inner} be the value of $D[head \ldots tail]$ before the inner loop starts at the $(k-1)$-st iteration of the outer loop. Then, by Invariant 4, $D[1 \ldots (head-1)] \prec\succ D_{inner}$. After the inner loop terminates, by Invariant 1, $D[1 \ldots (head-1)] \prec\succ D[head \ldots tail]$ and subsequently $D[1 \ldots (head-1)] \prec\succ D[head]$. After $head$ increases by 1 (in line 19), $D[1 \ldots (head-2)] \prec\succ D[head-1]$ and the assumption is rewritten as $\mathcal{S}(D[1 \ldots (head-2)]) = D[1 \ldots (head-2)]$. By proposition 2.2 and the assumption, $\mathcal{S}(D[1 \ldots (head-1)]) = \mathcal{S}(D[1 \ldots (head-2)]) \cup \mathcal{S}(D[head-1]) = D[1 \ldots (head-2)] \cup D[head-1] = D[1 \ldots (head-1)]$, as desired. □

□

Proof of Theorem 6.1

Proof. Let T_{1f} and T_{2f} be the final values of T_1 and T_2 in line 16, respectively. Since T_1 is initialized with S_1 and never grows, $T_{1f} \subset T_1 \subset S_1$ always holds. Since T_2 is initialized with an empty set and admits only tuples in S_2, $T_2 \subset T_{2f} \subset S_2$ always holds. Our goal is to prove $T_{1f} = S_1 \cap \mathcal{S}(S_1 \cup S_2)$ and $T_{2f} = S_2 \cap \mathcal{S}(S_1 \cup S_2)$:

(1) $T_{1f} \subset S_1 \cap \mathcal{S}(S_1 \cup S_2)$;
(2) $S_1 \cap \mathcal{S}(S_1 \cup S_2) \subset T_{1f}$;
(3) $T_{2f} \subset S_2 \cap \mathcal{S}(S_1 \cup S_2)$;
(4) $S_2 \cap \mathcal{S}(S_1 \cup S_2) \subset T_{2f}$.

(1) We have already shown $T_{1f} \subset S_1$. Consider $x \in T_{1f}$. There is no $y \in S_2$ such that $y \prec x$, for the invocation of $f\ y$ would find x in T_1 in line 4 and remove x from T_1 in line 6. Since S_2 contains no tuple dominating x, we have $x \in \mathcal{S}(S_1 \cup S_2)$.

(2) Consider $x \in S_1 \cap \mathcal{S}(S_1 \cup S_2)$. If $x \notin T_{1f}$, there must be $y \in S_2$ such that the invocation of $f\ y$ removes x from T_1 in line 6. Such y dominates x (line 5), which contradicts $x \in \mathcal{S}(S_1 \cup S_2)$. Hence we have $x \in T_{1f}$.

(1) and (2) prove $T_{1f} = S_1 \cap \mathcal{S}(S_1 \cup S_2)$.

(3) We have already shown $T_{2f} \subset S_2$. Consider $y \in T_{2f}$. The invocation of $f\ y$ must reach line 11, which means that it encounters no $x \in T_1$ such that $x \prec y$. Moreover it encounters all tuples in $T_{1f} = S_1 \cap \mathcal{S}(S_1 \cup S_2)$ because T_1 never admits new tuples. Hence $S_1 \cap \mathcal{S}(S_1 \cup S_2)$ contains no tuple dominating y, and we have $y \in \mathcal{S}(S_1 \cup S_2)$.

(4) Consider $y \in S_2 \cap \mathcal{S}(S_1 \cup S_2)$. If $y \notin T_{2f}$, the invocation of $f\ y$ must encounter $x \in T_1$ such that $x \prec y$, which contradicts $y \in \mathcal{S}(S_1 \cup S_2)$. Hence we have $y \in T_{2f}$.

(3) and (4) prove $T_{2f} = S_2 \cap \mathcal{S}(S_1 \cup S_2)$. □