Parallel Skyline Computation on Multicore Architectures

Sungwoo Park, Taekyung Kim, Jonghyun Park, Jinha Kim, Hyeonseung Im

Department of Computer Science and Engineering, Pohang University of Science and Technology
Gyeongbuk, Republic of Korea, 790-784
{gla,strikerz,parjong,goldbar,genilhs}@postech.ac.kr

***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. We compare two parallel skyline algorithms: a parallel version of the branch-and-bound algorithm (BBS) and a new parallel algorithm based on skeletal parallel programming. Experimental results show despite its simple design, the new parallel algorithm is comparable to parallel BBS in speed. For sequential skyline computation, the new algorithm far outperforms sequential BBS when the density of skyline tuples is low.**

I. Introduction

Multicore processors are going mainstream [1]. As a response to the problem of excessive power consumption and the lack of new optimization techniques, 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 [2]. 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* 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 [3], [4], [5], [6], [7], [8].

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 only on distributed architectures [9], [10], [11].

We investigate two complementary approaches to parallelizing skyline computation. First we parallelize a state-of-the-art sequential skyline algorithm to see if the design principle of sequential skyline computation also extends to parallel skyline computation. For our purpose, we choose the branch-and-bound algorithm (BBS) [7] which uses special index structures to eliminate from dominance tests a block of tuples at once. Second we develop a new parallel skyline algorithm, called $pskyline$, based on the divide-and-conquer strategy. Our parallel skyline algorithm 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).

For BBS, we parallelize the implementation provided by the authors of [12]. For $pskyline$, we use *skeletal parallel programming* [13] which provides a few primitive operations, such as map and reduce, to facilitate the principled development of parallel programs. We use the OpenMP programming environment [14] both to parallelize BBS and to instantiate $pskyline$ into an implementation tailored to multicore architectures. Experimental results show despite its simple design, $pskyline$ is comparable to parallel BBS in speed. $pskyline$ is also competitive for sequential skyline computation when only a single core is available. In particular, $pskyline$ far outperforms sequential BBS when the density of skyline tuples is low, or when the dataset is meaningful from the viewpoint of skyline computation.

Although the main topic of this paper is parallel skyline

1084-4627/09 $25.00 © 2009 IEEE
DOI 10.1109/ICDE.2009.42

computation on multicore architectures, we believe that its main contribution is to provide 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 reimplement database operations accordingly. 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 II introduces skyline computation and skeletal parallel programming, and discusses related work. Section III explains how we parallelize BBS. Section IV presents the design and implementation of our parallel skyline algorithm $pskyline$. Section V gives experimental results of parallel BBS and $pskyline$ on an eight-core machine. Section VI discusses the experimental results and Section VII concludes.

II. Preliminaries

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

A. 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 $\mathcal{S}(\mathcal{S}(D)) = \mathcal{S}(D)$ and $\mathcal{S}(S) \subset S$ 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. Let us write $D_1 \prec\!\!\succ D_2$ to mean that $p \prec\!\!\succ q$ 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 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 our algorithm may divide a given dataset in an arbitrary way.

B. Skeletal parallel programming

Skeletal parallel programming is a programming model for parallel computing in which an independent operation is applied to each element of a homogeneous collection of data. It requires two components: a parallel data structure and a set of *parallel skeletons.* A parallel data structure is a collection of homogeneous elements such that an independent operation can be applied to each individual element in parallel. Parallel skeletons are primitive constructs for performing elementary parallel computations and serve as building blocks for more complex parallel computations. By sequencing these parallel skeletons in the right order, we can implement parallel computing over the parallel data structure in a safe way.

Our parallel skyline algorithm uses lists as its parallel data structure. A list is a finite sequence of elements of the same type. We write $[x_1, \cdots, x_n]$ for a list consisting of elements x_1, $\cdots$, x_n. As special cases, [] denotes an empty list and $[x]$ denotes a singleton list consisting of element x. We write $l_1 +\!\!+ l_2$ for the concatenation of lists l_1 and l_2. $+\!\!+$ is an associative operator, so $l_1 +\!\!+ l_2 +\!\!+ l_3$, $(l_1 +\!\!+ l_2) +\!\!+ l_3$, and $l_1 +\!\!+ (l_2 +\!\!+ l_3)$ are all equivalent. We write $l - [x]$ for list l without element x.

In the design of our parallel skyline algorithm, we make extensive use of functions. For example, parallel skeletons are functions which 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 [15].

We write $f\ x$ for an application of function f to argument x. Note that we do not use parentheses to enclose x as in $f(x)$. A function may take multiple arguments and we write $f\ x_1\ \cdots\ x_n$ for an application of function f to arguments

$x_1, \cdots, x_n$ in that order. 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:

$$f\ x_1\ \cdots\ x_n \quad = \quad e$$

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 our parallel skyline algorithm, we consider two parallel skeletons: map and reduce. The map skeleton $pmap$ (*p*arallel map) takes a unary function f and a list l as its arguments, and applies f to each element of l in parallel:

$$pmap\ f\ [x_1, x_2, \cdots, x_n] \quad = \quad [f\ x_1, f\ x_2, \cdots, f\ x_n]$$

If f takes $O(1)$ sequential time, $pmap\ f\ l$ takes $O(1)$ parallel time. The reduce skeleton $preduce$ (*p*arallel $reduce$) takes an associative binary function f and a list l, and collapses l into a single element by repeatedly applying f to its elements:

$$\begin{aligned} preduce\ f\ [x] &= x \\ preduce\ f\ [x_1, x_2, \cdots, x_n] &= x_1\ \langle f \rangle\ x_2\ \langle f \rangle\ \cdots\ \langle f \rangle\ x_n \end{aligned}$$

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

We also consider $sreduce$ (*s*equential $reduce$), a sequential version of the reduce skeleton, defined as follows:

$$\begin{aligned} &sreduce\ f\ [x_1, \cdots, x_{n-1}, x_n] \ = \\ &\qquad f\ (sreduce\ f\ [x_1, \cdots, x_{n-1}])\ x_n \end{aligned}$$

$sreduce\ f\ l$ takes $O(n)$ sequential time if f takes $O(1)$ sequential time and n is the length of l.

C. Related work

The problem of skyline computation is known as the maximal vector problem in the computational geometry community. Kung *et al.* [16] 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 [17] 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.* [19] 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 a few algorithms that do not require special index structures or preprocessing. The block-nested-loops (BNL) algorithm [3] performs a dominance test between every pair of tuples while maintaining a window of candidate skyline tuples. The sort-first-skyline (SFS) algorithm [5] presorts a dataset according to a monotone preference function before starting a procedure similar to BNL. LESS (linear elimination sort for skyline) [6] incorporates two optimization techniques into SFS and is currently regarded as the best skyline algorithm that does not require index structures or preprocessing. SFS and LESS are amenable to parallelization because both algorithms involve sorting a dataset, but their parallelization has not been reported.

There are also a few skyline algorithms that exploit special index structures. The nearest-neighbor (NN) algorithm [4] and the branch-and-bound (BBS) algorithm [7] use R-trees as their index structures. The ZSearch algorithm [8] uses a new variant of B$^+$-tree, called *ZBtree*, for maintaining the set of candidate skyline tuples in Z-order [20]. Experimental results show that ZSearch runs consistently faster than BBS on typical datasets.

Parallel skyline computation so far has been considered only on distributed architectures in which participating nodes in a network share nothing and communicate only by exchanging messages [9], [10], [11]. While similar in spirit in that it attempts to utilize multiple computational units, our parallel skyline algorithm 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.

The idea of skeletal parallel programming is already in use by such database programming models as MapReduce [21], [22] and Map-Reduce-Merge [23]. Parallel constructs in these programming models are actually different from parallel skeletons in skeletal parallel programming. For example, the map construct in MapReduce takes as input a single key/value pair instead of a list of key/value pairs.

III. Parallel BBS

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

A. Branch-and-bound algorithm

BBS is a state-of-the-art skyline algorithm which uses R-trees as its index structures. Figure 1 shows the pseudocode of BBS. It takes as input an R-tree built from a dataset and returns the skyline set. BBS maintains in the main memory a heap holding candidate skyline tuples and an array S holding skyline tuples. Line 7 and 10 consist of a sequence of dominance tests between a candidate skyline tuple and all skyline tuples in S. Line 9 retrieves R-tree nodes and is the only part that requires I/O operations.

We parallelize BBS and compare it with our parallel skyline algorithm $pskyline$. Note that the comparison between $pskyline$ and BBS cannot be conclusive, since $pskyline$ has an advantage of requiring no index structures while BBS has an advantage of producing skyline tuples in a progressive manner. In this regard, such algorithms as BNL, SFS, and LESS are better choices which do not require special index structures. BBS, however, is usually faster and less costly than those algorithms requiring no index structures (see [8] for a comparison), which implies that our comparison with BBS does not compromise the evaluation of the relative performance of $pskyline$.

We use OpenMP [14] to parallelize BBS. OpenMP is a programming environment for parallel computing on shared

```
1.  BBS (R : R-tree) =
2.    begin
3.      S ← φ
4.      insert the root node of R into heap
5.      while heap is not empty do
6.        e ← remove top entry of heap
7.        if ∀e′ ∈ S.e ≺≻ e′
8.          if e is an intermediate node of R then
9.            for each child e_i of e do
10.             if ∀e′ ∈ S.e_i ≺≻ e′
11.               insert e_i into heap
12.           end for
13.         else insert e into S
14.         end if
15.     end while
16.     return S
17.   end
```

Fig. 1. Pseudocode of BBS

memory architectures such as multicore architectures. Usually parallel programming with OpenMP begins with a working sequential program. Then the programmer annotates parallelizable loops in it with OpenMP compiler directives to obtain an equivalent parallel program. Thus OpenMP provides an incremental way of writing parallel programs, which is one of its main strengths.

B. Parallelizing BBS

A naive approach to parallelizing BBS is to perform dominance tests in lines 7 and 10 of Figure 1 in parallel. These two lines are the most time-consuming part of BBS and do not require I/O operations. As an example, the following table shows the profiling result for two typical 10-dimensional datasets with 102400 tuples:

	Lines 7 and 10	Others
Dataset 1	97.6%	2.4%
Dataset 2	98.6%	1.4%

Thus lines 7 and 10 are the most appropriate part to rewrite when parallelizing BBS. We use OpenMP to implement this approach.

Unfortunately the speedup of the naive approach is not as high as we expect. The reason is that even if a certain thread finds a candidate skyline tuple to be dominated by a skyline tuple in S, we cannot terminate all other threads immediately (partially due to the limitation of OpenMP). If we initiate explicitly communications between threads in such cases, the communication cost far outweighs the benefit of parallel dominance tests.

Our approach is to execute lines 7 and 10 of Figure 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. Figure 2 shows the pseudocode of parallel BBS. The parallel sections of this algorithms are lines 12–15 and lines 19–22 which are marked with the **parallel for** construct. In addition to array S holding skyline tuples as in

```
1.  Parallel BBS (R : R-tree) =
2.    begin
3.      S ← φ, S′ ← φ
4.      insert the root node of R into heap
5.      while heap or S′ is not empty do
6.        if there is no intermediate node in S′
7.          and heap is not empty then
8.          e ← remove top entry of heap
9.          if ∀e′ ∈ S′.e ≺≻ e′
10.           S′ ← S′ ∪ {e}
11.       else
12.         parallel for each e′_i of S′ do
13.           if ∀e′ ∈ S.e′_i ≺≻ e′
14.             flag_i ← true
15.         end for
16.         for each e′_i of S′ do
17.           if flag_i is true
18.             if e′_i is an intermediate node of R then
19.               parallel for each child c_j of e′_i do
20.                 if ∀e′ ∈ S.c_j ≺≻ e′
21.                   cflag_j ← true
22.               end for
23.               for each child c_j of e′_i do
24.                 if cflag_j is true
25.                   insert c_j into heap
26.               end for
27.             else insert e′_i into S
28.             end if
29.         end for
30.         S′ ← φ
31.       end if
32.     end while
33.     return S
34.   end
```

Fig. 2. Pseudocode of parallel BBS

sequential BBS, parallel BBS uses another array S' holding R-tree nodes to be inspected in parallel. Lines 6–10 accumulate R-tree nodes in S' until an intermediate R-tree node is added to S', at which point all R-tree nodes in S' are inspected in parallel (lines 12–29). Note that all R-tree nodes in S' are incomparable. With this approach, we achieve a speedup approximately proportional to the number of cores. The details are described in Section V.

IV. Parallel skyline algorithm $pskyline$

This section develops our parallel skyline algorithm $pskyline$ and proves its correctness. It also describes an implementation of $pskyline$ and analyzes the I/O cost and the memory usage.

A. 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 $pskyline$ ($p$arallel $skyline$) such that

$$pskyline\ D \quad = \quad \mathcal{S}(D).$$

We represent all datasets as lists and use list concatenation $++$ in place of set union $\cup$. For example, distributivity of $\mathcal{S}$ now states $\mathcal{S}(D_1 ++ D_2) = \mathcal{S}(\mathcal{S}(D_1) ++ \mathcal{S}(D_2))$.

Basically our parallel skyline algorithm 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, our algorithm does not use a particular strategy. Rather it simply divides D linearly into smaller blocks so that their concatenation rebuilds D:

$$D \quad = \quad D_1 \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} D_b$$

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 the map and reduce skeletons, we introduce two auxiliary functions: $sskyline$ (*s*equential $skyline$) and $smerge$ (*s*equential $merge$). $sskyline$ performs a sequential computation to obtain the skyline set of a given block; $smerge$ performs a sequential computation to merge two skyline sets:

$$\begin{array}{rcl} sskyline\ D_i & = & \mathcal{S}(D_i) \\ smerge\ S_1\ S_2 & = & \mathcal{S}(S_1 \mathbin{+\!\!+} S_2) \\ & & \text{where } \mathcal{S}(S_1) = S_1 \text{ and } \mathcal{S}(S_2) = S_2 \end{array}$$

Using the fact that $smerge$ is associative, we obtain a definition of $pskyline$ that uses $pmap$ and $preduce$:

$$\begin{array}{rcl} pskyline_1\ D & = & preduce\ smerge\ (pmap\ sskyline\ L) \\ & & \text{where } D = D_1 \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} D_b \\ & & \quad L = [D_1, \cdots, D_b] \end{array}$$

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 $pskyline_1\ D$ 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 $D = D' \mathbin{+\!\!+} D''$. 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 $pmerge$ (*p*arallel $merge$) that performs a parallel computation to merge two skyline sets, we obtain another definition of $pskyline$ that uses $sreduce$ and $pmerge$:

$$pskyline_2\ D \quad = \quad sreduce\ pmerge\ (pmap\ sskyline\ L)$$

where L is given as in the definition of $pskyline_1$. Now we have a definition of $pskyline$ that takes full advantage of parallel computing. Our parallel skyline algorithm uses this second definition $pskyline_2$.

Below we describe our implementation of $pmerge$, $pmap$, and $sskyline$ in turn. For the sake of efficiency, our implementation stores all datasets in arrays. 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.

```
1.  pmerge S1 S2 =
2.    begin
3.      T1 ← S1
4.      T2 ← []
5.      f y = begin
6.                for each x ∈ T1 do
7.                  if y ≺ x then
8.                    T1 ← T1 − [x]
9.                  else if x ≺ y then
10.                   return
11.                 end if
12.               end for
13.               T2 ← T2 ++ [y]
14.             end
15.     pmap f S2
16.     return T1 ++ T2
17.   end
```

Fig. 3. $pmerge$ for parallel skyline merging

B. Parallel skyline merging

$pmerge$ is a function taking two skyline sets S_1 and S_2 and returning the skyline set of their union $S_1 \mathbin{+\!\!+} S_2$:

$$\begin{array}{rcl} pmerge\ S_1\ S_2 & = & \mathcal{S}(S_1 \mathbin{+\!\!+} S_2) \\ & & \text{where } \mathcal{S}(S_1) = S_1 \text{ and } \mathcal{S}(S_2) = S_2 \end{array}$$

First we describe an implementation of $pmerge$ that uses lists, and prove its correctness. Then we describe an equivalent implementation that uses arrays.

Figure 3 shows an implementation of $pmerge$ that uses lists. Given two lists S_1 and S_2 holding skyline sets, it creates two local lists T_1 and T_2 (lines 3 – 4). Then it defines a local function f (lines 5 – 14) and invokes $pmap$ (line 15). Finally it returns $T_1 \mathbin{+\!\!+} T_2$ as its result (line 16).

T_1 is initialized with S_1 (line 3) and decreases its size by eliminating non-skyline tuples in S_1 (line 8). T_2 is initialized with an empty list (line 4) and increases its size by admitting skyline tuples from S_2 (line 13). Eventually T_1 eliminates all non-skyline tuples in S_1 to become $S_1 \cap \mathcal{S}(S_1 \mathbin{+\!\!+} S_2)$, and T_2 admits all skyline tuples from S_2 to become $S_2 \cap \mathcal{S}(S_1 \mathbin{+\!\!+} S_2)$.

Given a tuple y in S_2, 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 7 – 8), but as soon as it locates a tuple in T_1 that dominates y, it terminates (lines 9 – 10). If no tuple in T_1 dominates y, it appends y to T_2 (line 13). 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 15). 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 8 and 13 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 4.1: If both $\mathcal{S}(S_1) = S_1$ and $\mathcal{S}(S_2) = S_2$ hold, $pmerge\ S_1\ S_2$ returns $\mathcal{S}(S_1 +\!+ S_2)$.

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 8 or T_2 in line 13 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 Figure 3 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 8 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 13 as a write of $\mathbf{true}$ to an element of F_2.

In order to store the array for $T_1 +\!+ T_2$, 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.

C. Parallel map

We use OpenMP [14] to achieve a lightweight implementation of the map skeleton $pmap$. The following C code illustrates how to implement the map skeleton with OpenMP:

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

It applies function `f` to each element of array `input` and stores the result in array `output`. The OpenMP directive in the first line specifies that `f` be applied in parallel. By encapsulating the loop within a function taking `f` as an argument, we obtain an implementation of the map skeleton.

D. Sequential skyline computation

Figure 4 shows our implementation of $sskyline$ for sequential skyline computation. 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 $\mathcal{S}(D[1 \ldots n])$. It uses an in-place algorithm and requires no extra memory, but overwrites the input array.

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

Fig. 4. $sskyline$ for sequential skyline computation

$sskyline$ 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 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 6). $sskyline$ maintains the following invariants at the beginning of the inner loop (in line 7):

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 $D[head \ldots tail]$.

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 $D[1 \dots n]$ before the outer loop begins (after line 4). $sskyline$ maintains the following invariants at the beginning of the outer loop (in line 5):

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

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

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

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

Thus we state the correctness of $sskyline$ as follows:

Theorem 4.2:

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

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 $D[1 \dots (head-1)]$, 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.

E. 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 our algorithm computes the skyline set of a dataset D of dimensionality d. We report the I/O cost in terms of the number of page access and the memory usage in terms of the number of tuples. We use a sequential file structure (with no indexes) to store D, and assume that a total of c cores are available and that the disk page size v is given in terms of the number of tuples.

In the first step, our algorithm divides D into b blocks $D_1, \cdots, D_b$. For the sake of simplicity, we assume that the size of every block is a multiple of v. Then it applies $sskyline$ to each block in parallel, and writes b resultant skyline sets $S_1, \cdots, S_b$ to temporary files in the disk. Since each invocation of $sskyline$ reads a block and writes a new skyline set, the I/O cost is

$$\sum_{i=1,\cdots,b} (\frac{|D_i|}{v} + \lceil \frac{|S_i|}{v} \rceil) \;=\; \frac{|D|}{v} + \sum_{i=1,\cdots,b} \lceil \frac{|S_i|}{v} \rceil$$

The first step stores $\min(b,c)\frac{|D|}{b}$ tuples in memory because only up to $\min(b,c)$ invocations of $sskyline$ can be simultaneously active.

In the second step, our algorithm applies $pmerge$ to a pair of skyline sets a total of $b-1$ times. It begins by reading S_1 from the disk. We let $S'_1 = \mathcal{S}(S_1) = S_1$. At the i-th invocation where $1 \le i \le b-1$, $pmerge$ reads S_{i+1} from the disk and computes $S'_{i+1} = \mathcal{S}(S'_i \cup S_{i+1})$. Thus the I/O cost is

$$\sum_{i=1,\cdots,b} \lceil \frac{|S_i|}{v} \rceil.$$

The second step requires

$$\max_{i=1,\cdots,b-1} (|S'_i| + |S_{i+1}|)(1 + \frac{1}{d})$$

tuples of memory because only one invocation of $pmerge$ is active at any moment. Here $(|S'_i| + |S_{i+1}|)\frac{1}{d}$ accounts for two arrays of boolean flags whose combined length is $|S_{i'}| + |S_{i+1}|$.

In total, $pskyline$ incurs an I/O cost of

$$\frac{|D|}{v} + 2 \sum_{i=1,\cdots,b} \lceil \frac{|S_i|}{v} \rceil$$

and requires

$$\max(\min(b,c)\frac{|D|}{b}, \max_{i=1,\cdots,b-1} (|S'_i| + |S_{i+1}|)(1 + \frac{1}{d}))$$

tuples of memory. Since skyline set S_i ($1 \le i \le b$) is no larger than block D_i, the I/O cost is no higher than $3\frac{|D|}{v}$. The memory usage is no higher than $|D|(1 + \frac{1}{d})$ which is reached when all tuples are skyline tuples, and no lower than $|\mathcal{S}(D)|(1 + \frac{1}{d})$ because $pmerge$ eventually loads 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 $|D|(1 + \frac{1}{d})$.

V. Experiments

This section presents experimental results of parallel BBS and $pskyline$ on an eight-core machine.

A. Experiment setup

The evaluation of parallel BBS and $pskyline$ uses both synthetic datasets and real datasets. We generate synthetic datasets according to either independent distributions or anti-correlated distributions [3]. 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. We represent each tuple element as a floating-point number of 4 bytes.

A dataset D determines three parameters d, n, and s:

- 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.*, $100K = 102400$).
- s denotes the number of skyline tuples in the dataset.

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 $n = 10K$, $100K$, $1000K$ to obtain $2 * 3 = 6$ datasets, all with the same dimensionality.

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 $c = 1$, 2, 4, 8. When c is set to 1, both parallel BBS and $pskyline$ degenerate to sequential skyline algorithms.

Each experiment reports three measurements T, O, and M as the performance metrics:

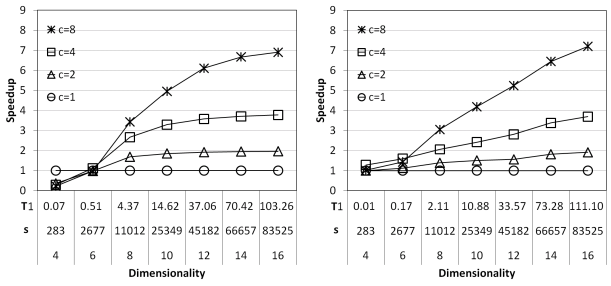
- T denotes the computation time measured in wall clock seconds. We write T_c for the computation time when c cores are used.
- O denotes the I/O cost calculated in terms of the number of page access. The I/O cost is independent of the number of cores.
- M denotes the memory usage in terms of the number of tuples.

All measurements are averaged over 10 sample runs.

We run all experiments on a Dell PowerEdge server with two quad-core Intel Xeon 2.83GHz CPUs (a total of eight cores) and 8 gigabytes of main memory. The disk page size is 4 kilobytes. We compile both parallel BBS and $pskyline$ using gcc (without optimization). In all experiments, the main memory is large enough to hold all data (both source data and intermediate results) manipulated by parallel BBS and $pskyline$.

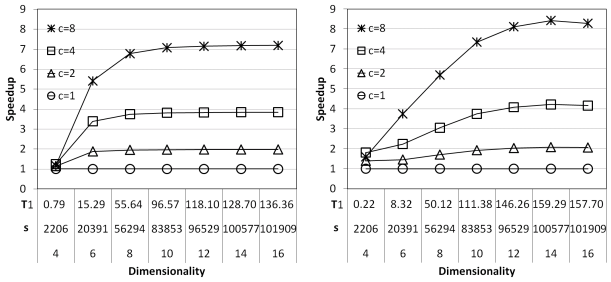
For $pskyline$, we need to choose the number b of blocks into which the dataset is divided in the first step. Too low a value for b may diminish the benefit of parallel computation during the first step (a parallel application of $sskyline$), but the I/O cost also decreases accordingly. Too high a value for b, on the other hand, may incur a high cost of I/O without fully exploiting multiple cores. In the extreme case of $b = n$, a total of n disk writes occur only to find singleton skyline sets.

We experimentally determine the relation between b and the performance of $pskyline$. We use two synthetic datasets, independent and anti-correlated, with $d = 10$ and $n = 100K$. We try $c = 1$, 2, 4, 8 and $b = 1$, 2, 4, 8, 16, 32, 64 and report T, O, and M for each combination of c and b. (If $b = 1$, all combinations are equivalent.)



Parallel BBS $pskyline$

(1) Independent datasets



Parallel BBS $pskyline$

(2) Anti-correlated datasets

Fig. 5. Speedup $\frac{T_1}{T}$ ($n = 100K$)

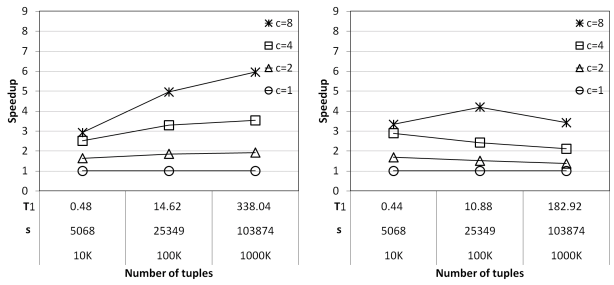
Experimental results (omitted for space reasons) show that the speedup is maximized when b is set to c. In this case, the entire dataset is loaded in the main memory and there is only a moderate I/O overhead. As every dataset fits in the main memory, we set b to c in all subsequent experiments so as to maximize the speedup.

B. 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 and $pskyline$.

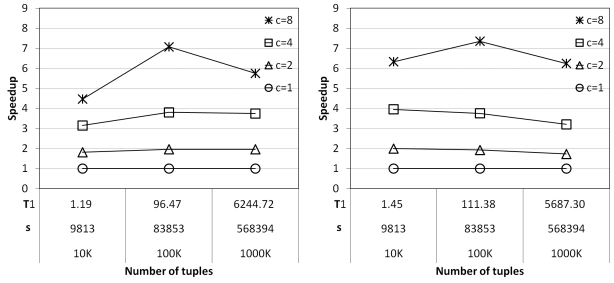
1) Effect of dimensionality and dataset size: Figure 5 shows the speedup relative to the case of $c = 1$ when d, the dimensionality of the dataset, varies from 4 to 16. For each dataset, it shows the number s of skyline tuples and the actual computation time T_1 for the case of $c = 1$. Both parallel BBS and $pskyline$ exhibit a similar pattern of increase in speedup: the speedup increases towards c as d increases. $pskyline$ sometimes achieves superlinear speedups on anti-correlated datasets because it uses different values of b as c changes (b is set to c).

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 means that we use not just multiple cores but multiple CPUs



Parallel BBS $pskyline$

(1) Independent datasets



Parallel BBS $pskyline$

(2) Anti-correlated datasets

Fig. 6. Speedup $\frac{T_1}{T}$ $(d = 10)$

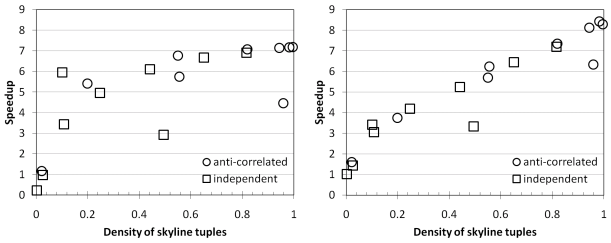
(namely two CPUs with four cores each), in which case the overhead is much higher. For other datasets, both parallel BBS and $pskyline$ achieve a speedup approximately proportional to the number of cores regardless of the dimensionality of the dataset.

Figure 6 shows the speedup relative to the case of $c = 1$ when n, the number of tuples in the dataset, varies from $10K$ to $1000K$. For each dataset, it shows the number s of skyline tuples and the actual computation time T_1 for the case of $c = 1$. Like the previous experiments, both parallel BBS and $pskyline$ exhibit a similar pattern of increase in speedup. Also an increase of c from 4 to 8 does not necessarily result in a performance improvement commensurate with the increase in c, as is the case for the independent dataset with $n = 10K$. For all other datasets, both parallel BBS and $pskyline$ achieve a speedup approximately proportional to the number of cores.

From these experiments, we draw the following conclusion:

- In general, both parallel BBS and $pskyline$ successfully utilize multiple cores and achieve a speedup approximately proportional to the number of cores.

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 both parallel BBS 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 $\frac{T_1}{T_8}$, the speedup from using eight cores relative to the case of $c = 1$, and $\frac{s}{n}$, the density of skyline tuples.



Parallel BBS $pskyline$

Fig. 7. Speedup $\frac{T_1}{T_8}$

Figure 7 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 7 for some independent dataset while it fails to achieve a speedup of 2 for some anti-correlated dataset.

Thus we draw the following conclusion:

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

C. Comparison between parallel BBS and $pskyline$

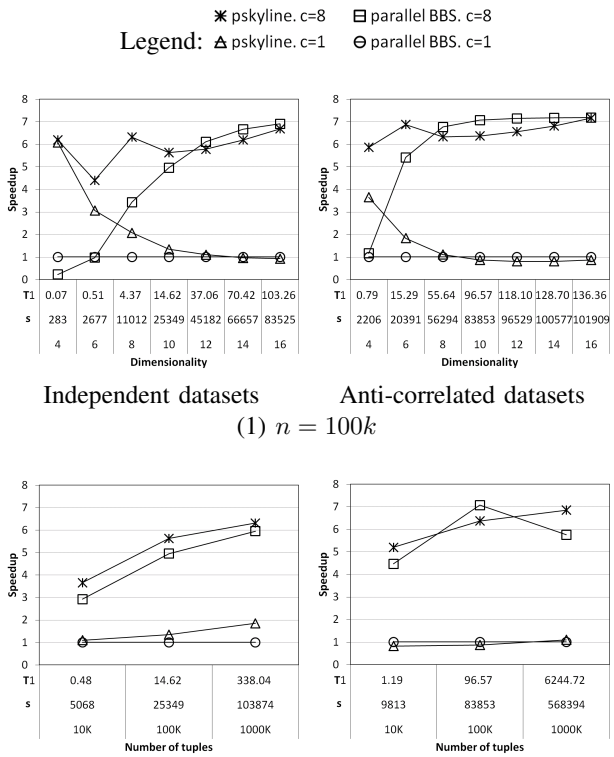
The second set of experiments compare parallel BBS and $pskyline$ for speed, I/O cost, and memory usage. We use the same datasets as in Section V-B, but analyze only the cases of $c = 1, 8$ to highlight the effect of using eight cores. Thus we compare the following four instances:

- Parallel BBS with $c = 1$, *i.e.*, sequential BBS
- Parallel BBS with $c = 8$
- $pskyline$ with $c = 1$, *i.e.*, $sskyline$
- $pskyline$ with $c = 8$

For speed, we calculate the speedup relative to sequential BBS in order to see how much improvement we can achieve over the state-of-the-art skyline algorithm. For I/O cost and memory usage, we identify the two instances of parallel BBS because the difference is zero (I/O cost) or negligible (memory usage).

Figure 8 shows the speedup of parallel BBS and $pskyline$ relative to sequential BBS (1) when d varies from 4 to 16 and (2) when n varies from $10K$ to $1000K$. For each dataset, it shows the number s of skyline tuples and the actual computation time T_1 for sequential BBS. The speedup of sequential BBS always stays at 1.

When using eight cores ($c = 8$), $pskyline$ is comparable to parallel BBS in speed. $pskyline$, however, tends to outperform parallel BBS as the dimensionality of the dataset decreases in (1) and as the dataset size increases in (2). For example, it out-

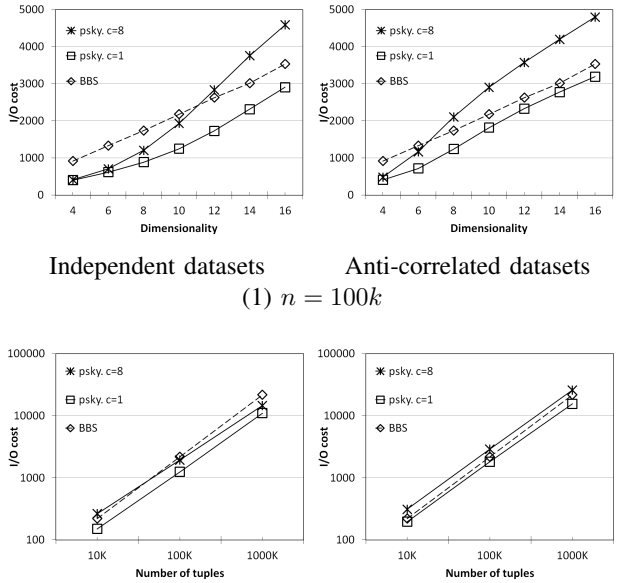


Independent datasets Anti-correlated datasets

(1) $n = 100k$

(2) $d = 10$

Fig. 8. Speedup of parallel BBS and $pskyline$



Independent datasets Anti-correlated datasets

(1) $n = 100k$

(2) $d = 10$

Fig. 9. I/O cost O

performs parallel BBS for datasets with low dimensionalities ($d = 4$) and large datasets ($n = 1000K$).

When using a single core ($c = 1$), $pskyline$ is still comparable to sequential BBS in speed. For example, for anti-correlated datasets in (1), sequential BBS starts to outperform $pskyline$ when d reaches 10, but the performance difference remains stable afterwards; for all independent datasets in (2), $pskyline$ runs faster than sequential BBS. This result is remarkable because $pskyline$ does not require sophisticated index structures for eliminating unnecessary dominance tests and only uses simple nested loops for sequential skyline computation. In particular, $pskyline$ far outperforms sequential BBS when the density of skyline tuples is low (*e.g.*, $d \leq 6$), or when the dataset is meaningful from the viewpoint of skyline computation.

Figure 9 shows the I/O cost (1) when d varies from 4 to 16 and (2) when n varies from $10K$ to $1000K$. The I/O cost increases both with d and with n because the raw size of the dataset increases accordingly. While $pskyline$ with $c = 1$ incurs a lower cost of I/O than BBS (and also $pskyline$ with $c = 8$) for every dataset, there is no clear winner between BBS and $pskyline$ with $c = 8$.

Figure 10 shows the memory usage (1) when d varies from 4 to 16 and (2) when n varies from $10K$ to $1000K$. In the case of (1), the memory usage for $pskyline$ lies between $|D|$ and $|D|(1 + \frac{1}{d})$ regardless of c (because we set b to c), whereas the memory usage for BBS heavily depends on both the dimensionality and the type of the dataset. Although the memory usage for $pskyline$ is lower than that for BBS for all anti-correlated datasets, there is no clear winner between BBS and $pskyline$ if independent datasets are also taken into account.

From these observations, we draw the following conclusion:

- Despite its simple design, $pskyline$ is comparable to parallel BBS in speed. As a sequential skyline algorithm, $pskyline$ far outperforms sequential BBS when the density of skyline tuples is low.
- With respect to both the I/O cost and the memory usage, there is no clear winner between parallel BBS and $pskyline$.

D. Real datasets

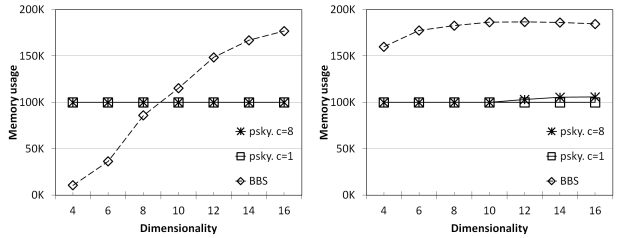
The third set of experiments test $pskyline$ and parallel BBS on two real datasets, Household and NBA.[1] Household is a six-dimensional dataset following an independent distribution where each tuple records the percentage of an American family's annual income spent on six types of expenditures. NBA is an eight-dimensional dataset following an anti-correlated distribution where each tuple records the statistics of an NBA player's performance in eight categories. For both parallel BBS and $pskyline$, we try $c = 1, 8$.

Figure 11 shows the results on the real datasets. When using eight cores, $pskyline$ achieves a moderate speedup of 4.40 for Household, but only a speedup of 1.39 for NBA because of the small dataset size. Parallel BBS also achieves a moderate

[1]These datasets can be collected from `http://www.ipums.org` and `http://www.nba.com`.

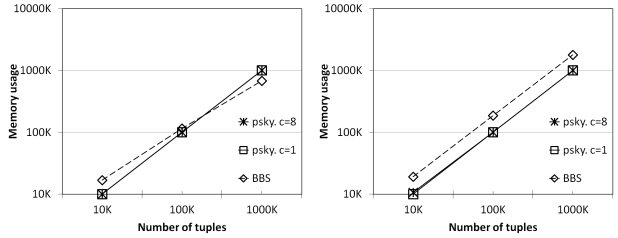
Dataset				Computation time T				I/O cost O			Memory usage M		
	d	n	s	$pskyline$		parallel BBS		$pskyline$		BBS	$pskyline$		BBS
				$c=1$	$c=8$	$c=1$	$c=8$	$c=1$	$c=8$		$c=1$	$c=8$	
Household	6	127931	5774	0.753	0.171	1.415	0.411	784	874	422	127931	127931	12761
NBA	8	17265	1796	0.061	0.044	0.117	0.103	150	210	278	17264	17264	3243

Fig. 11. Results on real datasets



Independent datasets Anti-correlated datasets

(1) $n = 100k$



Independent datasets Anti-correlated datasets

(2) $d = 10$

Fig. 10. Memory usage M

speedup of 3.44 for Household and a small speedup of 1.14 for NBA. As a sequential skyline algorithm, $pskyline$ runs faster than parallel BBS with $c = 1$ for both Household and NBA, which further indicates that $pskyline$ is a competitive algorithm even for sequential skyline computation. The I/O cost and the memory usage roughly agree with the pattern given in Section V-C.

VI. Discussion

This section discusses strengths and weaknesses of our parallel skyline algorithm $pskyline$. It also examines design principles of skyline computation for finding all skyline tuples.

A. Strengths and weaknesses of $pskyline$

Börzsönyi *et al.* make the following observation in their seminal paper on skyline computation [3]:

> *Our experimental results indicated that a database system should implement a block-nested-loops algorithm for good cases and a divide-and-conquer algorithm for tough cases.*

In essence, our parallel skyline algorithm combines both approaches to maximize their synergy: it first applies the divide-and-conquer strategy to reduce a tough case to multiple good cases and then applies an iterative sequential algorithm to deal with these good cases. By carefully designing a sequential skyline algorithm and a parallel skyline merging algorithm, we obtain a parallel skyline algorithm that exhibits speedups approximately proportional to the number of cores, *i.e.*, a "multicore-ready" skyline algorithm. Since it does not require special index structures or preprocessing, our algorithm is also trivial to adapt for such variants as constrained skyline computation or subspace skyline computation [24].

The main weakness of our algorithm is that it does not produce skyline tuples in a progressive manner. This is because our algorithm is basically a brute-force algorithm that uses no index structures and inspects the entire dataset to identify skyline tuples. The main goal in this work, however, is not to develop a parallel skyline algorithm that is versatile in all aspects. Rather it is to demonstrate that skyline computation (and certainly other database operations as well) can benefit from multicore architectures.

B. Design principles of skyline computation

Lee *et al.* [8] present three design principles of skyline computation which are indeed the foundation for existing skyline algorithms that use index structures:

(1) The access order of tuples has a direct impact on the performance of a skyline algorithm.
(2) Dominance tests are expensive and an efficient skyline algorithm should eliminate from dominance tests a block of tuples at once.
(3) The organization of candidate skyline tuples is critical for the efficiency of a skyline algorithm.

For the problem of finding *all* skyline tuples (as opposed to finding skyline tuples in a progressive manner), however, our analysis below indicates that these principles are not necessarily indispensable because of another crucial aspect of skyline computation, namely dominance tests *between skyline tuples.*

Consider a dataset of size n with s skyline tuples. Without a means of eliminating skyline tuples from dominance tests, we have to perform at least $\frac{s(s-1)}{2}$ dominance tests between skyline tuples in order to find all skyline tuples. The three design principles given above are intended to reduce the number of remaining dominance tests which are either between two incomparable tuples or between a dominating tuple (not necessarily a skyline tuple) and a dominated tuple. Unfortunately a significant amount of time is usually spent performing dominance tests between skyline tuples, on which the three design principles has no effect. For example, BBS exhibits

the following statistics for two 10-dimensional datasets from Section V:

- For the independent dataset with $n = 100K$ and $s = 25349$, a total of 601987455 dominance tests occur where at least 321273226 ($=\frac{s(s-1)}{2}$) dominance tests (53.3%) are between skyline tuples. If R-tree entries surviving dominance tests in lines 8 and 11 of Figure 1 are counted as skyline tuples, 393329207 dominance tests (65.3%) are between skyline tuples.
- For the anti-correlated dataset with $n = 100K$ and $s = 83853$, a total of 4259189493 dominance tests occur where at least 3515620878 ($=\frac{s(s-1)}{2}$) dominance tests (82.5%) are between skyline tuples. If we count R-tree entries as skyline tuples in the same way as above, 3720403796 dominance tests (87.4%) are between skyline tuples.

The design principle of $pskyline$ is to perform only as many dominance tests between skyline tuples as required (*i.e.*, $\frac{s(s-1)}{2}$ dominance tests) and to minimize the overhead of maintaining candidate skyline tuples, thereby ignoring the principle (3). It obeys the principle (1) in its implementation of $sskyline$, but ignores the principle (2) which applies to no more than $n - s$ dominance tests involving non-skyline tuples. It turns out that most of dominance tests performed by $pskyline$ are indeed between skyline tuples:

- For the independent dataset with $n = 100K$ and $s = 25349$, a total of 360859220 dominance tests occur where 321273226 ($=\frac{s(s-1)}{2}$) dominance tests (89.0%) are between skyline tuples.
- For the anti-correlated dataset with $n = 100K$ and $s = 83853$, a total of 3798855122 dominance tests occur where 3515620878 ($=\frac{s(s-1)}{2}$) dominance tests (92.5%) are between skyline tuples.

Thus we may regard $pskyline$ as a highly optimized algorithm among all skyline algorithms based on the same design principle. As it stores candidate skyline tuples in arrays, $pskyline$ is also easy to implement and parallelize, which is another important strength.

VII. 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.

Acknowledgments

We are grateful to Seung-won Hwang for discussions on the problem of skyline computation, Youngdae Kim for providing the data generation program, Joachim Selke for fixing a bug in $sskyline$, and the authors of [12] for making available their implementation of BBS. This work was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology(MEST)/Korea Science and Engineering Foundation(KOSEF), grant number 11-2008-007-03001-0.

References

[1] H. Sutter, "The free lunch is over: a fundamental turn toward concurrency in software," *Dr. Dobb's Journal*, vol. 30, no. 3, Mar. 2005.
[2] H. Sutter and J. Larus, "Software and the concurrency revolution," *Queue*, vol. 3, no. 7, pp. 54–62, 2005.
[3] S. Börzsönyi, D. Kossmann, and K. Stocker, "The skyline operator," in *ICDE*, 2001, pp. 421–430.
[4] D. Kossmann, F. Ramsak, and S. Rost, "Shooting stars in the sky: an online algorithm for skyline queries," in *VLDB*, 2002, pp. 275–286.
[5] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang, "Skyline with presorting," in *ICDE*, 2003, pp. 717–816.
[6] P. Godfrey, R. Shipley, and J. Gryz, "Maximal vector computation in large data sets," in *VLDB*, 2005, pp. 229–240.
[7] D. Papadias, Y. Tao, G. Fu, and B. Seeger, "Progressive skyline computation in database systems," *ACM Transactions on Database Systems*, vol. 30, no. 1, pp. 41–82, 2005.
[8] K. C. K. Lee, B. Zheng, H. Li, and W.-C. Lee, "Approaching the skyline in Z order," in *VLDB*, 2007, pp. 279–290.
[9] P. Wu, C. Zhang, Y. Feng, B. Y. Zhao, D. Agrawal, and A. E. Abbadi, "Parallelizing skyline queries for scalable distribution," in *EDBT*, 2006, pp. 112–130.
[10] A. Cosgaya-Lozano, A. Rau-Chaplin, and N. Zeh, "Parallel computation of skyline queries," in *HPCS*, 2007, p. 12.
[11] S. Wang, B. C. Ooi, A. K. H. Tung, and L. Xu, "Efficient skyline query processing on peer-to-peer networks," in *ICDE*, 2007, pp. 1126–1135.
[12] Y. Tao, X. Xiao, and J. Pei, "Efficient skyline and top-k retrieval in subspaces," *IEEE Transactions on Knowledge and Data Engineering*, vol. 19, no. 8, pp. 1072–1088, 2007.
[13] F. A. Rabhi and S. Gorlatch, Eds., *Patterns and skeletons for parallel and distributed computing*. Springer-Verlag, 2003.
[14] L. Dagum and R. Menon, "OpenMP: An industry-standard API for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998.
[15] S. Peyton Jones, Ed., *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.
[16] H. T. Kung, F. Luccio, and F. P. Preparata, "On finding the maxima of a set of vectors," *Journal of the ACM*, vol. 22, no. 4, pp. 469–476, 1975.
[17] I. Stojmenovic and M. Miyakawa, "An optimal parallel algorithm for solving the maximal elements problem in the plane," *Parallel Computing*, vol. 7, no. 2, pp. 249–251, 1988.
[18] J. Matousek, "Computing dominances in E^n," *Information Processing Letters*, vol. 38, no. 5, pp. 277–278, 1991.
[19] 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*. ACM, 1993, pp. 298–307.
[20] F. Ramsak, V. Markl, R. Fenk, M. Zirkel, K. Elhardt, and R. Bayer, "Integrating the ub-tree into a database system kernel," in *VLDB*, 2000, pp. 263–272.
[21] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," in *OSDI*, 2004, pp. 137–150.
[22] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating mapreduce for multi-core and multiprocessor systems," in *HPCA*, 2007, pp. 13–24.
[23] H. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker, "Map-Reduce-Merge: simplified relational data processing on large clusters," in *SIGMOD*, 2007, pp. 1029–1040.
[24] Y. Tao, X. Xiao, and J. Pei, "SUBSKY: Efficient computation of skylines in subspaces," in *ICDE*, 2006, p. 65.