Parallel Computation of Skyline Queries

Adan Cosgaya-Lozano
Faculty of Computer Science
Dalhousie University
Halifax, NS Canada
acosgaya@cs.dal.ca

Andrew Rau-Chaplin
Faculty of Computer Science
Dalhousie University
Halifax, NS Canada
arc@cs.dal.ca

Norbert Zeh
Faculty of Computer Science
Dalhousie University
Halifax, NS Canada
nzeh@cs.dal.ca

Abstract

Skyline queries have received considerable attention in the database community recently. The goal is to retrieve all records in a database that have the property that no other record is better according to all of a given set of criteria. While this problem has been well studied in the computational geometry literature, the solution of this problem in the database context requires techniques designed particularly to handle large amounts of data. In this paper, we show that parallel computing is an effective method to speed up the answering of skyline queries on large data sets. We also propose to preprocess the set of data points to quickly answer subsequent skyline queries on any subset of the dimensions.

1. Introduction

Given a collection S of database records, every record in S stores a collection of attributes. If an attribute is numerical, we often consider a record r to be better than another record r' w.r.t. this attribute if r's attribute value is less than or greater than that of r', depending on what this attribute represents. If, for example, the records in S represent hotels and the attribute we consider is the price of a room, we normally prefer hotel A over hotel B if a room in hotel A is cheaper than in hotel B. If this is the only attribute we are interested in, any standard database system can easily return the cheapest hotel. This hotel may be somewhere in Siberia, while we are looking for a hotel to stay at during a conference in Cancún. So distance to the conference site should also be taken into account. This leads to a trade-off, with some hotels being cheaper, while others may be closer to the conference site. The database system cannot determine what is more important to us: distance to the conference site or price. It can be assumed, however, that we would not be interested in a hotel A if there is another hotel B that is cheaper *and* closer to the conference site. We say that hotel B *dominates* hotel A. In order to give us the full range of choices, while not suggesting any hotels that are obviously inferior to others, the database system should therefore report all hotels in S that are not dominated by any other hotel. These hotels form the *skyline* of S. See Figure 1 for an illustration. In this example, every hotel is represented as a point in two-dimensional space, the x-coordinate being its distance from the conference site and the y-coordinate being its price. Hotels p_2, p_3, p_6, p_7, p_8 are dominated by hotel p_1, while hotels p_1, p_4, p_5 are not dominated by any other hotels. Hence, the latter form the skyline of this collection and constitute the set of candidates we would consider for booking a room.

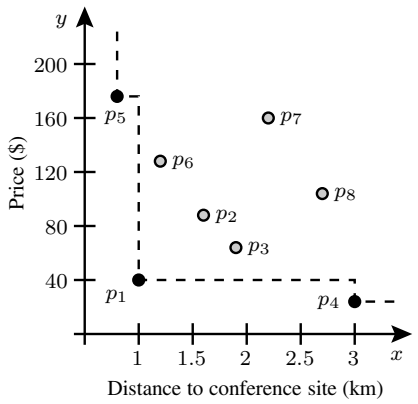


Figure 1. Skyline Example

More generally, we can consider the database records to be a set S of points in d-dimensional space. For a point p and all $1 \leq i \leq d$, we denote its i'th dimension by $x_i(p)$. We assume w.l.o.g. that all coordinates are non-negative. Point p_1 is said to *dominate* point p_2 if $x_i(p_1) \leq x_i(p_2)$, for all $1 \leq i \leq d$, and at least one of these inequalities is strict. A point p is a *skyline point* if it is not dominated by any other point in S. The *skyline* of S, denoted $\mathrm{sky}(S)$, is the collection of all its skyline points. Our goal in this paper is to compute this skyline quickly.

Due to its importance in a range of application areas, including statistics, economics, and many more, the computation of the skyline of a d-dimensional point set has received much attention in the computational geometry community [12], where a skyline point is called a *maximal el-*

21st International Symposium on High Performance Computing Systems and Applications(HPCS'07)
0-7695-2813-9/07 $20.00 © 2007 IEEE

ement. A theoretically efficient divide-and-conquer algorithm has been obtained by Kung et al. [10].

More recently, the database community has focused on skyline queries, aiming at the integration of a skyline operator into SQL [3] and the development of simple and practically efficient methods for answering these queries [5, 9, 11, 13, 14]. The proposed algorithms can be divided into two categories: Algorithms in the first category solve the problem without preprocessing. Algorithms in the second category first construct an efficient indexing structure of the point set and then use this structure to answer skyline queries quickly.

Most of the previous work on skyline queries has focused on the development of efficient sequential algorithms. Since, however, the data sets to be processed in real-world applications are of considerable size, we see a need for improved query performance, and parallel computing is a natural choice to achieve this performance improvement. In this paper, we focus on achieving good speed-up over sequential algorithms using a distributed-memory cluster. We develop a parallel algorithm for reporting the skyline of a point set and verify experimentally that this algorithm achieves good speed-up and scales well in practice. If the size of the skyline exceeds a certain value $k \leq n/p$ to be provided as an argument to our algorithm, we report a random sample of k skyline points. The motivation, as in [5], is that in most applications that use skyline queries, an exceedingly large skyline represents an overwhelming amount of information that has no added value over a smaller representative sample of the set of choices.

The rest of the paper is organized as follows. In Section 2, we discuss related work. In Section 3, we present our parallel algorithm. In Section 4, we discuss the experimental results obtained using an implementation of our algorithm. In Section 5, we provide a summary and discuss directions for future research.

2. Related Work

A number of algorithms have been proposed in the literature for answering skyline queries. Borzsonyi et al. [3] propose the Block-Nested-Loops algorithm, which takes the straightforward approach of comparing every point with every other point in the point set and adding it to the skyline if it is not dominated by any other point. Kossmann et al. [9] propose the Nearest-Neighbor algorithm, which uses a combination of R-trees and divide-and-conquer to answer skyline queries. Another R-tree-based algorithm, which we discuss in more detail below, is the branch-and-bound skyline algorithm by Papadias et al. [11]. Chan et al. [5] and Yuan et al. [14] extend the computation of skylines in different directions. The former paper proposes to report only the k most frequent skyline points if the skyline is too big, which is often the case in higher-dimensional space. The motivation is that a very large skyline usually provides little added information over a representative sample of skyline points. The latter paper focuses on answering skyline queries over the whole data cube defined by a given subset of dimensions.

Work on skyline queries in the computational geometry community has led to an efficient sequential algorithm by Kung et al. [10], which employs divide and conquer to solve the problem. This algorithm is discussed in more detail below. Dehne et al. [6] propose an optimal coarse-grained parallel algorithm for computing skylines in three dimensions; but their approach does not seem to lead to practical algorithms for higher-dimensional point sets.

Our parallel algorithm uses the divide-and-conquer algorithm by Kung et al. [10] and the branch-and-bound algorithm by Papadias et al. [11] as building blocks. Therefore, we discuss these two algorithms in more detail here.

2.1. Divide and Conquer (DC)

The divide-and-conquer algorithm of [10] performs a double recursion on the number of dimensions and the size of the point set. Given a d-dimensional point set S, the first step is to find the median coordinate in the d'th dimension, partition S into two sets L and R around this coordinate, and recursively find the skylines of L and R. The points in $\mathrm{sky}(L)$ are easily seen to belong to $\mathrm{sky}(S)$, while a point in $\mathrm{sky}(R)$ belongs to $\mathrm{sky}(S)$ if and only if it is not dominated by a point in $\mathrm{sky}(L)$. Since all points in R have a higher x_d-coordinate than any point in L, the latter depends only on the first $d-1$ dimensions, and the algorithm uses a recursive filtering procedure to filter out all points in R that are dominated by points in L in these dimensions.

The filtering procedure finds the median coordinate of the points in L in the $(d-1)$'st dimension and partitions L and R into two sets each around this coordinate, as shown in Figure 2. It can be observed that no point in R_b can be dominated by a point in L_t. Hence, it suffices to recursively filter all points in R_b against the points in L_b and to recursively filter all points in R_t against the points in L_b and L_t. For the filtering of the points in R_t against the points in L_b, we observe that all points in R_t have higher x_{d-1}- and x_d-coordinates than any point in L_b. Hence, this filtering step has to take only the first $d-2$ dimensions into account. The recursion in this procedure stops as soon as one of the two involved point sets has size at most one or the number of relevant dimensions has reduced to two, at which point a direct approach finishes the computation in linear time.

DC has a worst-case running time of $O(N \log^{d-2} N)$ and takes expected linear time on a random point set. In practice, it works very well for small point sets.

2.2. Branch-and-Bound Skyline (BBS)

BBS [11] is an index-based algorithm that finds the skyline points using an R-Tree index [1, 7]. An R-tree is a B-tree storing all objects in its leaves; every internal node stores the smallest axis-parallel box that contains all objects in the leaves that are descendants of this node. This box is called the *minimum bounding rectangle* (MBR) of the node.

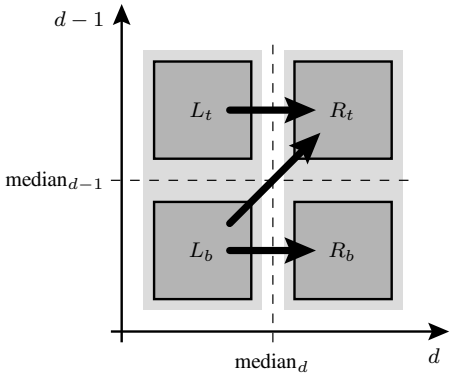


Figure 2. Recursive filtering in DC.

Note that, while the MBR's along any root-leaf path are properly nested, the bounding rectangles of sibling nodes are not necessarily disjoint. This is true even if the stored objects are points unless the subsets of points to be stored at the leaves are carefully chosen using a recursive space partition. One way to obtain such a partition is by splitting the dimensions in a round-robin fashion. The tree obtained using this procedure, when applied to the point set in Figure 1, is shown in Figure 3.

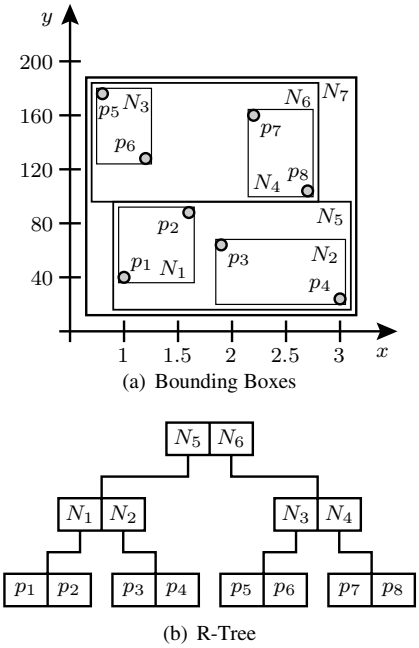


Figure 3. R-Tree for point set from Figure 1

BBS incrementally constructs a list L of skyline points it has identified; initially, $L = \emptyset$. In order to find the skyline points, it performs a traversal of the R-tree, pruning entire subtrees if the root of such a subtree is dominated by a point in L, where we say that a node is dominated by a point p if the same is true for the corner of the node's MBR that is closest to the origin.

More precisely, the algorithm maintains a priority queue Q of tree nodes, which is initialized to contain the root of the R-tree. It then retrieves nodes from Q one by one and processes them. For an internal node v, processing v involves the following steps: First the algorithm checks whether v is dominated by a point in L. If so, the same is true for every point stored in v's subtree, and the subtree can be pruned. Otherwise, v's children are inspected and each child that is not dominated by a point in L is inserted into Q. If v is a leaf, the algorithm again checks whether v is dominated by a point in L. If not, it inspects the points stored at v and adds each point that is not dominated by a point in L to L.

To ensure early pruning of large subtrees, the nodes in Q are processed by increasing *mindist*, which is defined to be the L_1-distance of the closest corner of the MBR of the node from the origin.

Papadias et al. [11] show that BBS accesses every node at most once and visits a node only if its subtree contains at least one skyline point. They also present experimental results that demonstrate that the performance of BBS mainly depends on the size of the computed skyline and the dimensionality of the data. This last limitation is inherited from the used index structure, that is, from the R-tree.

3. Parallel Skyline Algorithm

Our goal in this paper is to achieve better performance than any of the algorithms discussed previously by designing a parallel algorithm for answering skyline queries. Our model is that of a distributed-memory cluster; that is, if n is the problem size, our computational environment consists of p independent processors (workstations) $P_0, P_1, \ldots, P_{p-1}$, each with $O(n/p)$ main memory, connected through an interconnection network. Communication between the processors happens through the explicit exchange of messages.

As the divide-and-conquer algorithm of [10], our algorithm (see Algorithm 1) exploits that, for any decomposition of the point set S into subsets $S_0, S_1, \ldots, S_{p-1}$, we have $\text{sky}(S) = \text{sky}(\text{sky}(S_0) \cup \text{sky}(S_1) \cup \cdots \cup \text{sky}(S_{p-1}))$. Thus, we can assign a subset S_i of n/p points to each processor P_i. In the first round, each processor P_i locally computes $\text{sky}(S_i)$. The second round then computes $\text{sky}(S)$ from the local skylines computed by the processors.

To avoid that some processor computes a big local skyline all of whose member points are dominated by points assigned to some other processor, our goal is to ensure that each set S_i is a sample of S whose structure is similar to S. To this end, we distribute the points randomly over the processors. After this distribution step, the first round of the algorithm is straightforward: each processor locally applies a sequential skyline algorithm to its point set.

The second round requires more care. Let S' be the union of the local skylines computed by all processors. If $|S'| \leq n/p$, we use an all-to-all communication to send S'

0-7695-2813-9/07 $20.00

Algorithm 1 PARALLEL SKYLINE QUERY$(S_0, \ldots, S_{p-1}, k)$

Input: Sets $S_0, S_1, \ldots, S_{p-1}$; set S_i is stored on processor P_i. $k \leq n/p$ is the desired number of skyline points.
Output: Set L of $\min(k, |\operatorname{sky}(S)|)$ skyline points.

1: **for all** $0 \leq i \leq p-1$ **do**
2: P_i builds an R-tree $R(S_i)$ for S_i and runs BBS on $R(S_i)$ to compute $\operatorname{sky}(S_i)$. Let $l_i = |\operatorname{sky}(S_i)|$.
3: **end for**
4: Perform an all-to-all communication to compute $l = \sum_{i=0}^{p-1} l_i$ at each processor.
5: **if** $l < \frac{n}{p}$ **then**
6: Perform an all-to-all communication to collect $S' = \bigcup_{i=0}^{p-1} \operatorname{sky}(S_i)$ at each processor. Divide S' into p chunks of size $|S'|/p$. Let S'_i be the i'th such chunk.
7: **for all** $0 \leq i \leq p-1$ **do**
8: P_i identifies $S'_i \cap \operatorname{sky}(S)$. This is done either using DC or by building an R-tree $R(S')$ and running BBS on $R(S')$ and S'_i (see text for details).
9: **end for**
10: Perform an all-to-all communication to collect $\operatorname{sky}(S)$ at P_0.
11: **return** a set $L \subseteq \operatorname{sky}(S)$ of size $\min(k, |\operatorname{sky}(S)|)$.
12: **else**
13: Let $g = 0$ be the current size of $\operatorname{sky}(S)$; each processor stores the number g_i of points it has contributed to $\operatorname{sky}(S)$ so far. Let $S'_i = \operatorname{sky}(S_i)$.
14: Build an R-tree $R(\operatorname{sky}(S_i))$ on $\operatorname{sky}(S_i)$.
15: **while** $g < k$ and not all sets S'_i are empty **do**
16: **for all** $0 \leq i \leq p-1$ **do**
17: P_i selects and removes a set S''_i of $l_i \cdot \frac{n/p}{l}$ points from S'_i.
18: **end for**
19: Perform an all-to-all communication to construct the set $S'' = \bigcup_{i=0}^{p-1} S''_i$ at each processor.
20: **for all** $0 \leq i \leq p-1$ **do**
21: P_i runs BBS on S'' and $R(\operatorname{sky}(S_i))$ to mark all points in S'' dominated by a point in $\operatorname{sky}(S_i)$.
22: **end for**
23: Perform an all-to-all communication to send all marked points back to their original processors.
24: **for all** $0 \leq i \leq p-1$ **do**
25: P_i removes the received points from S''_i and adds the remaining points in S''_i to its local portion of $\operatorname{sky}(S)$, increasing g_i by the number of added points.
26: **end for**
27: Perform an all-to-all communication to compute $g = \sum_{i=0}^{p-1} g_i$ at each processor.
28: **end while**
29: Perform an all-to-all communication to collect all g skyline points in P_0.
30: **return** a set $L \subseteq S'' \subseteq \operatorname{sky}(S)$ of size $\min(k, |\operatorname{sky}(S)|)$.
31: **end if**

to every processor. Each processor P_i then determines for a subset $S'_i \subseteq S'$ of points which points in S'_i are dominated by points in S' and removes these points from S'_i. At the end of this round, we perform another all-to-all communication to collect the points in sets S'_i that were not deleted in processor P_0. These points form $\operatorname{sky}(S)$, and processor P_0 returns this set or a sample $L \subseteq \operatorname{sky}(S)$ of size k, whichever is smaller.

If $|S'| > n/p$, we pick a set S'' of n/p random points from S' and send S'' to every processor. Each processor P_i now determines which points in S'' are dominated by points in $\operatorname{sky}(S_i)$ and marks them. The marked points are then sent back to the processors whence they came, and every processor declares the points it contributed to S'' and which are not sent back to it by any other processor to be members of $\operatorname{sky}(S)$. We iterate this process until we have processed all points in S' or we have identified at least k members of $\operatorname{sky}(S)$. The fact that we choose each sample of size n/p uniformly at random from S' ensures that, if we do not output all of $\operatorname{sky}(S)$, the sample we output is uniformly distributed and is representative of the points in $\operatorname{sky}(S)$.

This general framework in principle allows the use of any sequential skyline algorithm to perform the local skyline computations in the first round, as well as the filtering step in the second round. For the first round, we choose BBS because only few points in each set S_i belong to $\operatorname{sky}(S_i)$, which makes BBS's pruning strategy highly effective. In the second round, the structure of S' as the union of p local skylines results in about a quarter of the points in S' belonging to $\operatorname{sky}(S)$. Hence, the pruning strategy is less effective, and DC outperforms BBS on sets of up to 50,000 points and queries on up to 5 dimensions. Thus, we use DC when the queries are low-dimensional and S' is small; otherwise, we use BBS.

4. Experimental Analysis

We evaluated the performance of our algorithm using an extensive set of experiments, which demonstrated that our method (1) achieves linear speed-up in most cases and (2) scales well to large data sets beyond the reach of any single node in our cluster.

Our experiments were performed on a 32-node Beowulf-style cluster with 1.8GHz Intel Xeon processors. Each node was equipped with 1 GB of RAM and two 40GB 7200 RPM IDE disk drives. Each node was running Linux Redhat 7.2 with gcc 2.95.3 and MPI/LAM 6.5.6, as part of a ROCKS cluster distribution. All nodes were interconnected via a Cisco 6509 GigE switch. The implementation was done in C++, using MPI to perform interprocessor communication. Our implementation of BBS uses the R*-tree implementation of [8].

In our experiments, we used synthetic and real data sets of up to 6 dimensions to evaluate the performance of our algorithm. The coordinates were integers. For the synthetic data, these coordinates are chosen uniformly at random. The real-world data was taken from [4] and con-

0-7695-2813-9/07 $20.00

tained geographic data that provides hydrological information on a continental scale. For our speed-up experiments we both used synthetic and real data sets of either 1,000,000 or 5,000,000 records. For our final scale-up experiments we used synthetic data of 2-6 dimensions and up to 80,000,000 records.

Our timing results denote wall clock time, measured from the start of the first process in our parallel algorithm till the termination of the last process. These timings include the time to read the input files from disk.

Note that the part of our algorithm that addresses the case $|S'| > n/p$ is only required in extreme cases because, as shown in Bentley et al. [2], the expected size of the skyline of a random point set is $O(\log^{d-1} n)$, which is typically less than n/p.

4.1. Experiments with R-Tree on Skyline Dimensions

Our first set of experiments focused on evaluating the overall speed-up of our algorithm. Figure 4a shows the relative speed-up with 1–16 processors on data sets of 2–6 dimensions; Figure 4b shows the corresponding wall clock times. We observe that our algorithm achieves superlinear speed-up, which we believe to be the result of cache effects. Given more processors, each processor has to process less data, resulting in a higher fraction of data that fits into cache.

4.2. Experiments with R-Tree on All Dimensions

While the speed-up results in the previous section are promising, the corresponding wall clock times are substantial (as is the case for previous sequential algorithms). It turns out that at least 90% of the processing time are spent on constructing the R-trees. This begs the question whether one should build the R-tree once and subsequently use it for multiple queries on different subsets of the dimensions. In our next set of experiments, we therefore build an R-tree based on all 6 dimensions in our data set and then use the same tree to answer skyline queries on any subset of 2–6 dimensions.

Our evaluation focuses on two performance measures: The first one is the overall running time of building the 6-dimensional R-tree and answering one d'-dimensional skyline query using this tree, where $2 \leq d' \leq 6$. Our goal here is to compare these running times to those achieved when building the R-tree on only the dimensions relevant to the answered skyline queries. The second performance measure is the query time, given the R-tree. This reflects the idea that we are willing to pay a substantial preprocessing time to construct the R-tree if subsequently queries are fast. Note that, if our algorithm uses BBS in the second round, the query procedure still has to build an R-tree on the point set produced by the first round of our algorithm (Line 8 in Algorithm 1). The time to build this R-tree and the communication time required to answer a query are included in the query time.

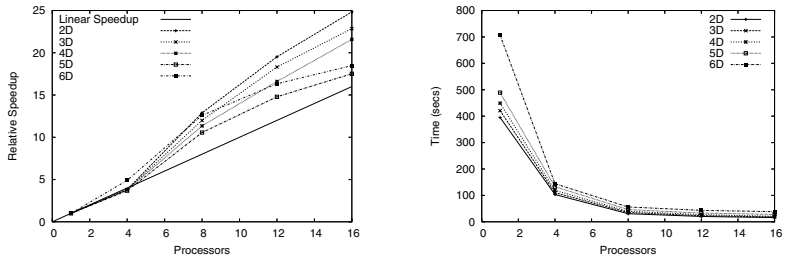
Figures 5 and 6 show our speed-up results on hydrological data sets of 1,000,000 and 5,000,000 points, respectively. We observe in Figures 5a and 6a that our algorithm again achieves superlinear speed-up. Comparing Figures 4a and 5a, we observe that the speed-up for dimensions less than 6 is greater when building the tree on all 6 dimensions than when building the tree on only the dimensions relevant to the skyline query. Figures 4b and 5b provide an explanation for this effect: building a 6-dimensional R-tree is significantly more expensive than building a lower-dimensional R-tree, thus establishing a higher baseline on a single processor with which the running times on multiple processors are compared. When the number of dimensions in the query increases, this effect is less pronounced because the query cost contributes more substantially to the overall running time. In summary, even when building the R-tree on all dimensions in the data set, the speed-up remains superlinear, and the overall running time less than doubles.

The increased construction cost of the R-tree is worthwhile if it can be amortized over a large number of queries that can subsequently be answered using the tree. Figures 5c and 6c show our speed-up results for answering queries, once the R-tree has been constructed, on 1,000,000 and 5,000,000 points, respectively. We observe that our algorithm scales well up to 4 processors and reasonably well up to 8 processors. After this point, the speed-up curve tails off, particularly for higher-dimensional queries. The reason is that the total number of local skyline points increases with the number of processors, thus increasing the cost of the second round of our algorithm (Lines 7–13). This effect becomes more pronounced with higher-dimensional queries.

Note that this increased total number of local skyline points does not adversely affect the number local skyline points every processor has to check for membership in the global skyline: the number of local skyline points per processor decreases. It does, however, increase the size of the R-tree on S' and, consequently, the cost of building this tree. Since each processor builds its own copy of this tree, this step is effectively not parallelized and constitutes the major share of the cost of the second round.

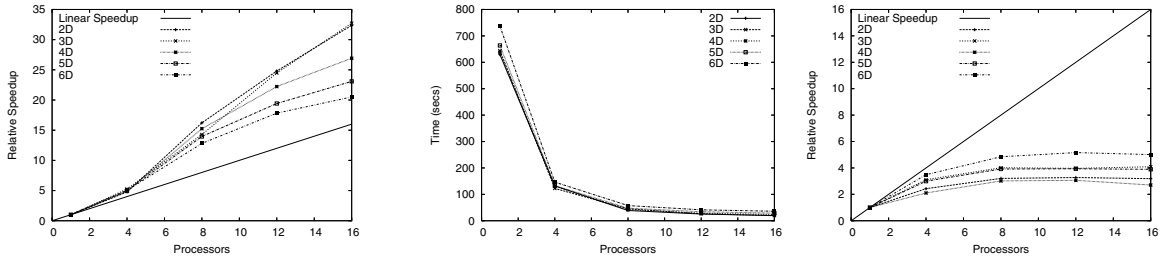
A careful interpretation of these results leads to the conclusion that there is a trade-off between parallelizing the first round and paying a penalty in the second round. Round one gets faster with an increasing number of processors, while the total number of points that survive after this round increases, leading to a substantial increase in the cost of the second round. This effect is best understood if one considers the extreme case of $p = n$. Then the first round is trivial because every processor has only one point; the second round then deteriorates to constructing an R-tree on the entire point set.

It is therefore useful to increase the number of processors only when increasingly large data sets need to be processed. One would hope that a good parallel algorithm does not only allow us to process a given data set faster than a se-



(a) Relative speed-up (b) Wall clock time

Figure 4. Timing and speed-up results on hydrological data set (1,000,000 points), building the R-tree on only the skyline dimensions.



(a) Total relative speed-up (b) Total wall clock time (c) Query relative speed-up

Figure 5. Timing and speed-up results on hydrological data set (1,000,000 points), building the R-tree on all six dimensions.

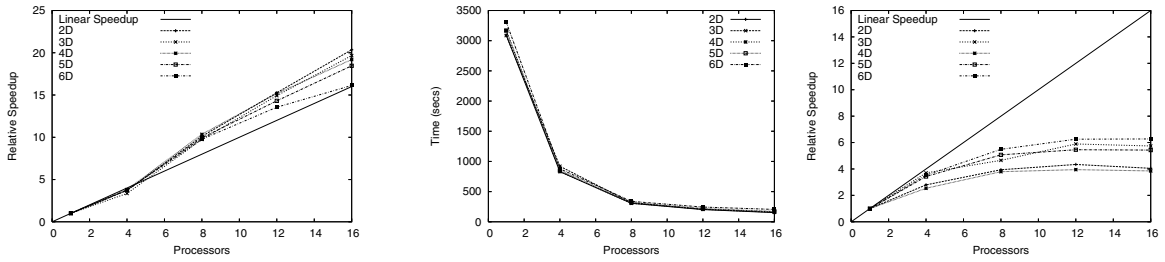
quential algorithm; but the increased amount of main memory and processing power should also allow the processing of data sets beyond the reach of sequential machines. Thus, our last experiment focuses on measuring the scale-up of our algorithm, where the size of the processed data set increases proportionally with the number of processors, and one hopes to be able to process p times as much data using p processors as one can process with a single processor in the same amount of time. Figure 7 shows our experimental results.

Overall, the scale-up results shown in Figure 7 are very encouraging. Consistent with our speed-up results for the total time of building the R-tree and answering a query, the scale-up of the total time, shown in Figure 7a, is a perfect 1 for all dimensions. For the query time, the scale-up is above 0.75 for all dimensions and numbers of processors; see Figure 7b. For 2 dimensions the slight drop in scale-up is likely due to an insufficient amount of work to parallelize effectively, as the total query time never exceeds 0.2 seconds. For 4 dimensions scale-up drops off early between 1 and 4 processors but then holds steady. For reasons we cannot yet explain, processing S' takes a relatively greater proportion of the time for 4 dimensions leading to this early drop in scale-up. For the other cases, including 5- and 6-dimensional cases, which are by far the hardest to solve sequentially, the algorithm achieves nearly perfect scale-up. For 5 and 6 dimensions on 16 processors and 80,000,000 records a scale-up of over .9 is achieved.

5. Summary and Future Work

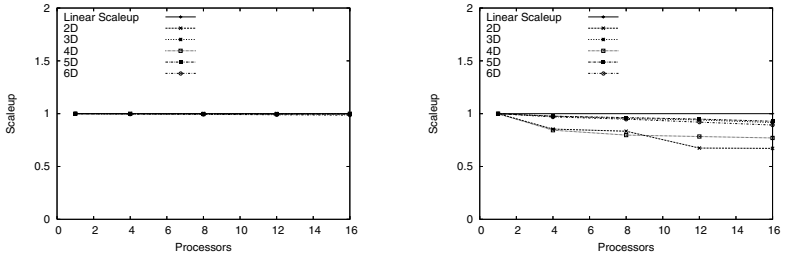
Our results demon strate that parallel computing can be used effectively to speed up the computation of skylines of large data sets of a moderate number of dimensions, which is what is relevant in practice [9].

Our experiments do, however, suggest several directions for future work, aiming at speeding up the second round of our algorithm. The most important improvement necessary to achieve better performance is to remove the bottleneck of sequentially building an R-tree on the local skyline points at the beginning of the second round. We hope to be able to parallelize this step by letting every processor build an R-tree on $1/p$ of the points; once this is done, the processors communicate their local R-trees to each other, and every processor combines the p local R-trees it has received to obtain an R-tree on the whole point set. This requires the construction of a subtree of size $O(p)$ on top of the received local R-trees. The main challenge here is to represent the local R-trees so that they can be communicated and combined efficiently.



(a) Total relative speed-up (b) Total wall clock time (c) Query relative speed-up

Figure 6. Timing and speed-up results on hydrological data set (5,000,000 points), building the R-tree on all six dimensions.



(a) Total scale-up (b) Query scale-up

Figure 7. Scale-up results on synthetic data set with 5,000,000 points per processor and building the R-tree on all size dimensions.

References

[1] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: an efficient and robust access method for points and rectangles. In *SIGMOD '90*, pages 322–331, New York, NY, USA, 1990. ACM Press.

[2] J. L. Bentley, H. T. Kung, M. Schkolnick, and C. D. Thompson. On the average number of maxima in a set of vectors and applications. *J. ACM*, 25(4):536–543, 1978.

[3] S. Borzsonyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.

[4] cgmLab Portal. Hydro1k elevation derivative database, http://cgmlab.cs.dal.ca/downloadarea/datasets/.

[5] Chee Yong Chan, H. V. Jagadish, Kian-Lee Tan, Anthony K. H. Tung, and Zhenjie Zhang. On high dimensional skylines. In *EDBT*, pages 478–495, 2006.

[6] Frank K. H. A. Dehne, Andreas Fabri, and Andrew Rau-Chaplin. Scalable parallel geometric algorithms for coarse grained multicomputers. In *Symposium on Computational Geometry*, pages 298–307, 1993.

[7] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD '84*, pages 47–57, New York, NY, USA, 1984. ACM Press.

[8] Marios Hadjieleftheriou. Spatial index library, http://u-foria.org/marioh/spatialindex/index.html.

[9] Donald Kossmann, Frank Ramsak, and Steffen Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *VLDB*, 2002.

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

[11] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. An optimal and progressive algorithm for skyline queries. In *SIGMOD '03*, pages 467–478, New York, NY, USA, 2003. ACM Press.

[12] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction.* Springer, 1985.

[13] Yufei Tao, Xiaokui Xiao, and Jian Pei. Subsky: Efficient computation of skylines in subspaces. In *ICDE*, page 65, 2006.

[14] Yidong Yuan, Xuemin Lin, Qing Liu, Wei Wang, Jeffrey Xu Yu, and Qing Zhang. Efficient computation of the skyline cube. In *VLDB*, pages 241–252, 2005.

0-7695-2813-9/07 $20.00