Highly Scalable Multiprocessing Algorithms for Preference-Based Database Retrieval

Joachim Selke, Christoph Lofi, and Wolf-Tilo Balke

Institut für Informationssysteme, Technische Universität Braunschweig
Mühlenpfordtstraße 23, Braunschweig, Germany
{selke, lofi, balke}@ifis.cs.tu-bs.de

Abstract. Until recently algorithms continuously gained free performance improvements due to ever increasing processor speeds. Unfortunately, this development has reached its limit. Nowadays, new generations of CPUs focus on increasing the number of processing cores instead of simply increasing the performance of a single core. Thus, sequential algorithms will be excluded from future technological advances. Instead, highly scalable parallel algorithms are needed to fully tap new hardware potentials. In this paper we establish a design space for parallel algorithms in the domain of personalized database retrieval, taking skyline algorithms as a representative example. We will investigate the spectrum of base operations of different retrieval algorithms and various parallelization techniques to develop a set of highly scalable and high-performing skyline algorithms for different retrieval scenarios. Finally, we extensively evaluate these algorithms to showcase their superior characteristics.

1 Introduction

Retrieval algorithms are at the heart of every database system and the search for ever more efficient algorithms in terms of scalability and runtimes has propelled research. The basic way of showing an algorithm's superiority is to implement it together with its competitors within the same framework and then evaluate how it behaves in different retrieval scenarios, usually exploring different problem instances, database sizes, and data distributions. It is interesting to note that this experimental evaluation is usually not depending on the hardware used (except for absolute runtime measurements, where however all competitors are affected by the hardware in a similar manner). Thus, any algorithm outperforming its competitors will do so on every platform and due to Moore's law will even get faster in absolute terms over time.

Moore's law states that the density of transistors on a CPU about doubles every two years and held for the last decades. Until recently it basically meant that also algorithms' performance doubles every two years, because increased transistor density was employed to increase clock rates and to support more complex CPU operations. But now processor designers have hit a ceiling as the benefits of even more transistors on a circuit cannot be harvested by traditional optimization techniques due to thermal problems. Chip manufacturers now opt to use the additional potential for replicating parallel processing cores. Quad-core machines are commonplace today, with 6-core processors projected for early 2010 and 80-core chips already existing as research prototype in Intel's Tera-scale project[1].

However, such additional cores do not necessarily result in increased performance, since most applications are build using sequential algorithms. Thus, taking advantage of this new potential depends on the aptitude of the application developer: the potential for parallel performance increase is best described by Amdahl's law. Amdahl's law stipulates that the speedup of any algorithm using multiple processors is strictly limited by the time needed to run its sequential fraction. For example, a program having a sequential fraction of as little as 10% can achieve a speedup factor of at most 10, no matter how many CPUs are available. On the other hand, the speedup factor achievable for its parallel portion is only capped by the actual number of processors. Thus, only highly parallel algorithms can benefit from future hardware improvements.

Actually, this problem already raised considerable attention. In its 2008 report on the IT landscape 2008–2012 Gartner Research ranks the need for multicore software architectures among the 10 most disruptive technologies. Making effective use of this technology will therefore need considerable changes in today's software development: "Running advanced multicore machines with today's software is like putting a Ferrari engine in a go-cart"[2]. The fact is illustrated in Fig. 1: with the advent of the multi-core era, the performance potential between parallel and sequential algorithms is continuously diverging.

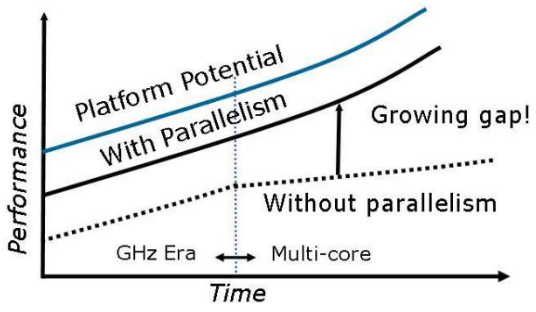

Fig. 1. The need for parallelism for algorithms' performance[3].

Of course, these developments also affect the field of databases which now needs to adapt their retrieval algorithms to the changing hardware landscape, too. In the course of this paper we will argue that for designing efficient and highly scalable multiprocessor retrieval algorithms even the development cycle has to be rethought. Instead of developing new algorithms only based on the drawbacks and shortcomings of the most current top-performers, algorithm designers have to take a closer look at

[1] http://techresearch.intel.com/articles/Tera-Scale/1421.htm
[2] "The Impact of Multicore Architectures on Server Scaling", Gartner, 2008
[3] Image from "An Overview of the Parallelization Implementation Methods in Intel C++ Compilers", Intel Corporation, 2008

the entire palette of possible base algorithms and exploit parallelizable features wherever possible. Only in this way the sequential part of the algorithm can be effectively minimized. For this it is also mandatory to deeply understand the basic operations needed for each algorithm. Our paper's contribution thus is threefold:

- Using the example of skyline queries we show how to use the variety of algorithms to set up the design space for parallel retrieval algorithms.
- We investigate which current techniques in parallelization can be used to efficiently implement modular operations.
- We apply our design process to BNL, one of the most prominent algorithms for skyline queries, and show that it leads to parallel implementations showing an almost linear scalability behavior in multi-core architectures.

2 The Parallelization Design Space for Retrieval Algorithms

It is interesting to note that most database retrieval algorithms are using a rather limited set of basic operations. Thus, the actual efficiency and the fine-tuning for specific scenarios is mainly achieved by innovative control flows and additional optimizations like e.g., using specialized index structures. For instance in the area of ranked query processing it was recently shown that is indeed possible to break down algorithms for retrieval tasks as different as skyline queries, top-k queries, and k-dominance queries to only three basic operations [23]. The idea is that such basic operations can be efficiently implemented within the database core and then offer interfaces for retrieval algorithms. Given the new hardware challenges it now becomes necessary to optimize those base operations for parallelization in order to benefit from future performance improvements by Moore's law.

In the following we will demonstrate by the example of skylining algorithms how such operations (in particular object comparisons in a nested-loop) can be parallelized with great effect by abstaining almost completely from sequential parts. The design space for improved algorithms is spanned on the one hand by the different evaluation approaches and their basic operations, on other hand by the novel parallelization techniques for these operations. The 'right' mixture of both will enable us to derive high-performing algorithms as we will see later in the experimental section.

2.1 The Design Space for Skyline Query Evaluation

Taking a closer look at recent algorithms for skyline query evaluation, we found that the algorithms can be classified into several distinguishable groups. In the following, we will cover the most important ones: block-nested-loops algorithms, divide-and-conquer algorithms, and multi-scan algorithms.

```
function BNL($A$)
    $M \leftarrow \emptyset$
    for each $a \in A$ do
        $M \leftarrow$ BNL-STEP($M, a$)                ▷ update $M$
    end for
    return $M$
end function

function BNL-STEP($M, a$)
    for each $m \in M$ do
        if $m \succ a$ then
            return $M$                          ▷ $a$ is not a maximum
        else if $m \prec a$ then
            $M \leftarrow M \setminus \{m\}$                ▷ $m$ is not a maximum
        end if
    end for
    return $M \cup \{a\}$                 ▷ $a$ and all $m \in M$ are maxima
end function
```

Fig. 2. Block-Nested-Loop (BNL) algorithm

```
function DC($A$)
    if $|A| \geq 2$ then
        split $A$ into $A_1$ and $A_2$ (preferably of equal size)
        $M_1 \leftarrow$ DC($A_1$)
        $M_2 \leftarrow$ DC($A_2$)
        for each $(m_1, m_2) \in M_1 \times M_2$ do
            if $m_1 \succ m_2$ then
                $M_2 \leftarrow M_2 \setminus \{m_2\}$
            else if $m_1 \prec m_2$ then
                $M_1 \leftarrow M_1 \setminus \{m_1\}$
            end if
        end for
        return $M_1 \cup M_2$
    else
        return $A$
    end if
end function
```

Fig. 3. Divide & Conquer algorithm

```
function BEST($A$)
    $M \leftarrow \emptyset$
    $A' \leftarrow A$
    while $A' \neq \emptyset$ do
        choose some $m \in A'$                    ▷ choose candidate
        $A' \leftarrow A' \setminus \{m\}$
        for each $a \in A'$ do
            if $a \succ m$ then
                $m \leftarrow a$                  ▷ $a$ is the new candidate
                $A' \leftarrow A' \setminus \{m\}$
                restart the for-each-loop
            else if $a \prec m$ then
                $A' \leftarrow A' \setminus \{a\}$            ▷ $a$ is not a maximum
            end if
        end for
        $M \leftarrow M \cup \{m\}$                   ▷ $m$ is a maximum
    end while
    return $M$
end function
```

Fig. 4. The Best/sskyline algorithm

Algorithms of the *block-nested-loop class (BNL)* [3, 12, 13] are probably the most prominent algorithms for computing skylines. In fact the basic operation of collecting maxima during a single scan of the input data can be found at the core of several state-of-the-art skyline algorithms [5, 8, 9, 10, 11] and is illustrated in Figure 2. The "block" in its name refers to the fact that it linearly scans over the data set A and continuously maintains a block (or window) of data elements M containing the maximal elements with respect to the data read so far. For each data record $a \in A$ that BNL processes, the function BNL-STEP is called, which eliminates all records in M being dominated by a, and adds a to M, if a is not dominated. The major advantage of BNL is its simplicity and suitability for solving general comparison-based preference queries [4, 13] (i.e., BNL can also be used to compute the maxima of arbitrary partial orders). Furthermore, a multitude of optimization techniques is applicable to BNL algorithms like e.g. dynamic sorting or indexing [5].

A second class of algorithms for skyline evaluation is based on a straightforward *divide-and-conquer strategy*, as shown in Fig. 3. Given a data set A, it first checks the cardinality of A. In case $|A| = 1$ the algorithm simply returns A. Otherwise, A is split into two sets A_1 and A_2 and the algorithm applies itself recursively on both of them. The two results M_1 and M_2 are subsequently cleaned from local maxima by comparing each element of M_1 to each element of M_2 and removing all dominated elements during this process. The set of A's maxima is constructed by joining the reduced sets M_1 and M_2. Although the algorithm has excellent theoretical properties [3, 17], there is no efficient implementation of this recursive process [5]. The main problem preventing an efficient implementation seems to be that the algorithm either requires massive disk IO, or needs to keep a large amount of intermediate results in main memory. However, when abstaining from recursion, its basic split-and-merge scheme is definitely applicable in a parallel scenario.

The third class of skyline algorithms is based on *multiple scans* of the database instance and includes algorithms like *Best* or *sskyline* [14, 15, 16]. They can especially provide highly efficient cache-conscious implementations. The complete algorithm is shown in Fig. 3. After creating an in-memory copy A' of A, an arbitrary element $m \in A'$ is selected as maximum candidate. After removing m from A', the whole set A' is scanned, and dominated elements are removed from it. If some element is found in A' that dominates the current candidate m, it is removed from A' and used as the new candidate; then, the scan of A' is restarted. If no element of A' dominates the current candidate, then it is a maximum. The whole process is repeated until A' is empty. In our experiments, we found Best/sskyline to make around half the number of comparisons as BNL due to the early elimination of maxima. But to reach its performance, Best/sskyline must scan (and even modify) the data set numerous times. This can only be done efficiently, if the whole input data set A fits in main memory; a requirement not desirable for general skyline algorithms. Nevertheless, as we will see in Section 3.1 Best/sskyline can form a useful building block in a parallel algorithm.

2.2 The Design Space for Parallelization of Basic Operations

For parallelizing the skyline problem, different strategies are at hand: The first is *classical distribution*, i.e. splitting data into multiple work packages which are distributed among the worker threads. Such threads can work independently of each other and finally, their results are combined. This strategy adapts the split-and-merge concept found within algorithms of the divide-and-conquer class. The advantage of this style of algorithm is that the threads do not need shared memory and thus could also be deployed to different machines (e.g. computer clusters). However, it is necessary to combine the results of the threads which introduces some overhead in terms of the program's sequential part and may lead to suboptimal scalability. In summary, skyline algorithms following the split-and-merge approach show good performance when only few threads can be used, since the overhead introduced increases with a higher degree of parallelism.

The second strategy is to employ algorithms working on a *shared data structure*, i.e., each thread can read and modify the same dataset. This style of algorithm has just recently become viable due to the advent of tightly coupled multicore processors. Still, the main problem of shared-memory algorithms remains at hand: One has to ensure that no data is read or written which has just been accessed by another thread (dirty reads or writes) in order to avoid data inconsistency. When considering for example a block-nested loops algorithm two major critical situations can be identified: a) overtaking threads: in this case, the overtaking thread will lose a comparison with the current element of the slower thread b) deleting/appending: the list structure may corrupt, if two threads try to delete or append the same nodes simultaneously due to the resulting inconsistent linkage of node.

Like in transaction systems, these problems can be tackled by synchronization and locking protocols. However, there is a variety of different approaches to secure a shared data structure, each showing individual runtime performance. We will briefly introduce these common approaches also used in our following algorithms:

- *Full Synchronization:* this simple protocol locks the whole data structure for every access. Obviously, this will not allow any parallelism and is, although secure, unsuitable for performance-oriented algorithms.
- *Continuous Locking:* the data structure is locked at node level for every access and is a straight-forward semi-naïve approach to the problem. However, locking still carries an expensive overhead despite recent hard- and software progress. Thus, this technique suffers severely from the overhead induced by acquiring and releasing such a high number of locks.
- *Lazy Locking:* this approach is similar to continuous locking. However, it aims at using as few locks as possible. Locks are only acquired when they are really needed, i.e. when deleting or inserting nodes. Unfortunately, this approach leads to more complex algorithms which are harder to design and debug. For example, it is necessary to identify all critical situations and provide according safeguards. Also, in case of our implementations, additional security mechanisms (like using flags) are necessary to ensure the data structure's consistency. But from a performance point of view lazy locking algorithms are definitely superior to the previous two locking protocols in highly parallel scenarios. Still, for cases using very few threads (e.g. two threads) split-and-merge algorithms may be a better choice.
- *Lock Free Synchronization:* this technique completely abstains from using locks. Instead, a special hardware instruction within the CPU core is used to implement an optimistic protocol. The base idea is to perform changes to the data structure and check later whether any concurrent modifications had occurred. When a conflict occurs, all modifications are undone and repeated. The performance of lock free protocols scale with the probability of conflicts: the more likely conflicts occur; the worse is the algorithm's expected performance.
  Interestingly, we observed in our experiments a similar or slightly lower performance of lock free synchronization compared to the lazy locking variants. However, the performance ratio of those two techniques is depending on the hardware efficiency of locking compared to the instructions used in lock free synchronization; thus performance may change when using different CPU architectures or operation systems and both algorithm styles seem viable alternatives.

3 Parallel Skyline Computing

In this section, we will utilize our design space for designing parallel algorithms with the mentioned techniques. Up to now, all parallel skyline algorithms proposed in literature directly rely on the basic divide-and-conquer scheme. Following our design considerations, these algorithms already cover an important application scenario and thus will serve as our baseline in the later experiments. In addition, for the *multi-core shared-memory* scenario we will design novel algorithms and show that they indeed outperform their current parallel competitors.

3.1 Algorithms using Split-And-Merge Parallelization

The multi-processor scenario without shared memory directly calls for algorithms based on the split-and-merge parallelization scheme used by the divide-and-conquer class. Basically, the input data set A is first split into k parts $A_1, \dots, A_k$. Then, in parallel, the skyline of each part is computed (using any sequential algorithm for each part). Finally, the resulting k local skylines $M_1, \dots, M_k$ are merged to produce the global skyline. To foster parallelization, it is sensible to split A in at least as many parts as there are processors.

In particular, the following distributed merging method has been applied in a distributed scenario without shared memory [24]: First, assign the i-th local skyline M_i to node i and make the union of all other local skylines $\overline{M_i} = M_1 \cup \cdots \cup M_{i-1} \cup M_{i+1} \cup \cdots \cup M_k$ accessible to this node. Then, node i compares each element of M_i to all elements in $\overline{M_i}$, removing all dominated elements from M_i. After this step, M_i only contains elements of A's global skyline. Finally, the set of all global maxima $M = M_1 \cup \cdots \cup M_k$ is constructed at some central node. This algorithm will be referred to as Distributed in the following. It is interesting to note that the Distributed algorithm does not rely on shared-memory at any point, thus it particularly applies to cluster-style distributed scenarios. However, its performance suffers from the complicated merge step.

Focusing on scenarios with shared memory this shortcoming is remedied by the pskyline method proposed in [16], which also applies the split-and-merge scheme. After splitting the dataset similar to the Distributed algorithm, pskyline uses a main memory algorithm (Best/sskyline) for the computation of the k local skylines. But by deciding for an efficient main memory implementation, it cannot process all k parts of A in parallel on large data sets. Instead, if r cores are available, the total number of chunks is chosen such that r chunks jointly fit into main memory. After all k local skylines have been computed (and written to disk), an improved merge scheme relying on shared memory is used: in deep-left-tree style, local skylines are merged successively, two at a time. In this shared memory merging step, all r cores can be used simultaneously. The experimental evaluation in [1] indicates that the psykline algorithm is indeed highly scalable in cases where the skyline is large relative to the number of database tuples. However, scalability degrades rapidly for smaller skyline sizes. For example, a speedup of just about 4 is reported for 8 cores and a skyline size of about 20% of the database size.

3.2 Continuously Locked Parallel BNL

Exploiting our algorithmic design space, we now explore algorithms for a shared-memory multi-core scenario departing from split-and-merge techniques. As we have seen the aim is true parallelization with a minimum of sequential overhead. Since algorithms of the BNL class only need a single pass over the input data, the basic idea of the following algorithms is to extend the BNL algorithm in such a way that multiple worker threads can simultaneously share and modify BNL's window. In the following, the window is represented by a linked list data structure.

Of course entering the shared memory part of the design space means that we have to care for the thread safety, i.e., concurrent access to data must be guarded by using an adequate locking scheme or conflict resolution. The simplest viable locking scheme is to continuously lock each node being accessed. Thus, each thread traversing the linked list releases a node's lock only after acquiring the lock for its successor. In particular, this strategy prevents that threads can pass each other. Passing might result in an unnoticed removal of a node by some other thread. This technique of always holding two locks per thread and adhering to the lock/unlock order is known as lock coupling (or hand-over-hand locking) [19]. Within BNL's outer loop, each thread continuously requests a new data record from the central data manager. For each data record, the list is traversed and each node is compared to the current data record, performing deletions of nodes, if needed. In case no node dominated the current record, it is appended to the list. Although this approach is a valid parallel implementation of the BNL algorithm, its way of locking significantly limits its performance. If two threads access neighboring nodes, they will continuously interfere with each other while traversing the list. Furthermore, the omnipresent locking and unlocking operations introduce a large computational overhead.

3.3 Lazy List Parallel BNL

Actually—and in contrast to the continuously locking scheme—the autonomy of threads only has to be constrained when a node has to be modified. This means that for each addition or deletion of a node in the list, a modification lock has to be obtained, whereas simple comparison operations do not require explicit locks. This idea of modification locks can be implemented using a novel concurrent data structure called the Lazy List [20]. We adjusted the basic structure for the use in skyline computation and show the resulting code in Figure 5.

In particular, our algorithm uses a binary flag $deleted$ to guarantee that no adjacent nodes are modified concurrently, which might otherwise result in violating the pointer integrity. Before a node can be deleted, locks for the current node and its predecessor have to be acquired. The removal of the current node $curr$ then is always a two-step process: First, $curr.deleted$ is set to true, indicating $curr$'s logical removal (and thus locking it effectively for modifications by its successor), and second, the predecessor's pointer $pred.next$ has to be set to $curr.next$, thus unlinking the current node. By calling the $validate$ function it is checked whether $pred$ and $succ$ have been deleted in the meantime or some node has been inserted in-between.

```
procedure ParBNL-LazyList-Thread()
  NextRecord:
  while $(a \leftarrow \text{GetNextRecord}()) \neq$ null do
    TraverseList:
    $pred \leftarrow$ head
    $curr \leftarrow pred$.next
    while $curr \neq$ tail do                              ▷ iterate
      if $curr$.item $\succ a$ then
        goto NextRecord                       ▷ process next record
      else if $curr$.item $\prec a$ then
        $pred$.lock                    ▷ try to delete current node
        $curr$.lock
        if Validate($pred$, $curr$) then
          $curr$.deleted = true              ▷ looks good, go on
          $pred$.next = $curr$.next
          $curr$.unlock
          $pred$.unlock
        else
          $curr$.unlock                        ▷ error, restart
          $pred$.unlock
          goto TraverseList
        end if
        $curr$ = $curr$.next;
      else
        $pred \leftarrow curr$
        $curr \leftarrow curr$.next
      end if
    end while
    $pred$.lock               ▷ it is $curr$ = tail, try to append $a$
    if Validate($pred$, $curr$) then
      $new \leftarrow$ Node($a$)                  ▷ looks good, go on
      $pred$.next $\leftarrow new$
      $new$.next $\leftarrow$ tail
      $pred$.unlock
    else
      $pred$.unlock                            ▷ error, restart
      goto TraverseList
    end if
  end while
end procedure

function Validate($pred$, $curr$)
  return $\neg pred$.deleted $\wedge \neg curr$.deleted $\wedge\, pred$.next $= curr$
end function
```

Fig. 5. The parallel BNL Lazy List algorithm.

```
procedure ParBNL-LockFree-Thread()
  NextRecord:
  while $(a \leftarrow \text{GetNextRecord}()) \neq$ null do
    TraverseList:
    $pred \leftarrow$ head
    $curr \leftarrow pred$.next
    while $curr \neq$ tail do
      $(succ, currdel) \leftarrow (curr.\text{next}, curr.\text{deleted})$
      if $currdel$ then
        $cas \leftarrow$ CAS$\big(\langle pred.\text{next}, pred.\text{deleted}\rangle,$
                    $\langle curr, \text{false}\rangle \rightsquigarrow \langle succ, \text{false}\rangle\big)$
        if $cas$ then
          $curr \leftarrow succ$
        else
          goto TraverseList
        end if
      else
        if $curr$.item $\succ a$ then
          goto NextRecord
        else if $curr$.item $\prec a$ then
          $cas \leftarrow$ CAS$\big(\langle curr.\text{next}, curr.\text{deleted}\rangle,$
                      $\langle succ, \star\rangle \rightsquigarrow \langle succ, \text{true}\rangle\big)$
          if $cas$ then
            CAS$\big(\langle pred.\text{next}, pred.\text{deleted}\rangle,$
                    $\langle curr, \text{false}\rangle \rightsquigarrow \langle succ, \text{false}\rangle\big)$
          else
            goto TraverseList
          end if
          $curr \leftarrow succ$
        else
          $pred \leftarrow curr$
          $curr \leftarrow curr$.next
        end if
      end if
    end while
    $new \leftarrow$ Node($a$)
    $new$.next $\leftarrow$ tail
    $cas \leftarrow$ CAS$\big(\langle pred.\text{next}, pred.\text{deleted}\rangle,$
                $\langle \text{tail}, \text{false}\rangle \rightsquigarrow \langle new, \text{false}\rangle\big)$
    if $\neg cas$ then
      goto TraverseList
    end if
  end while
end procedure
```

Fig. 6. The lock-free parallel BNL algorithm using a linked list.

Depending on the result, either the current node can be deleted, or there had been some concurrent modification of this list, from which the algorithm recovers by restarting the iteration from the beginning of the window. Appending a new node works similar. First, a lock is obtained (locking $pred$ is sufficient in this case), second, the algorithm checks for concurrent modifications, and then either appends the node physically or restarts the iteration.

3.4 Lock-Free Parallel BNL

Most approaches falling into our parallelization design space are using locking protocols, e.g., as our previous algorithm. The alternative is to completely abstain from locking and to implement a non-blocking *optimistic protocol*. This goal is supported by a special hardware operation called compare-and-swap (CAS). The CAS operation *atomically* compares a variable to some given value and, if both are the same, sets the first variable to some given new value. In our algorithm shown in Figure 6, it is denoted as $CAS(x, y \rightsquigarrow z)$: it atomically compares the variable x to the value y and, in case $x = y$, sets x to the value z. The function returns $true$ if the operation succeeded, otherwise it returns $false$. In our case, we will always use compare-and-swap to guard the node pointer $next$ and the deletion flag $deleted$.

Our approach to lock-free lists is inspired by the Harris-Michael algorithm [43, 44]. The linked list is traversed as in the sequential BNL algorithm, except for the following: first, an additional pointer variable $succ$ is used to store the successor of the current node, and second, in each traversal step, the $deleted$ flag of the current node is checked in order to physically remove nodes that have previously been marked for deletion. If the flag is set to $true$, the algorithm tries to unlink the current node using a CAS operation. In case of failure, a concurrent modification has occurred and the list traversal is restarted.

Dominated nodes are removed by first marking the current node as deleted using CAS (line 21). If this operations succeeds, the algorithms tries to physically delete the node (line 24, the star indicates that the current value of $curr.deleted$ does not matter); otherwise, the traversal is restarted. It does not matter, whether the physical removal succeeds or not, since logically deleted nodes will be cleaned up anyway by other threads during their list traversal as previously described (line 10).

Finally, new nodes are appended to the list also by a CAS instruction (line 38). In case of failure, the list traversal is restarted from the beginning. Although we do not use any locking mechanism in this algorithm, checking the status flag of all (critical) CAS operations allows us to detect all concurrent list modifications and respond accordingly.

4 Experiments

In the previous sections we explored our design space from an algorithmic point of view and derived several algorithms for shared-memory skyline computation. But similar to physical tuning methods, only thorough experimentation will reveal the real-world performance of the outlined techniques. In order to investigate the full variety of performance characteristics in an unbiased fashion, the following evaluations will be executed on synthetic datasets. To create these datasets, we used the Independent data generator commonly used in skyline research [3]. We also applied all evaluations to the Anticorrelated, and Correlated data generators. However, all results show the same trends as those seen in the Independent evaluations and were thus omitted for brevity. Of course, data sets of varying size ($n = 100K, 1M, 10M$) and varying dimensionality ($d = 5, 6, 7, 8, 9$) have been used.

All our experiments have been conducted on a *single* node of a 12-node cluster running SUSE Linux Enterprise 10. The node we used is equipped with two Intel Xeon E5472 3.0 GHz quad-core processors, thus providing a total of 8 cores at each node. Our algorithms have been implemented using the Java programming language version 6.0. We only used the built-in mechanisms for locking, compare-and-swap operations, and thread management. The experiments are executed on a Sun Java 6.0u13 64Bit server JVM in HotSpot mixed mode.

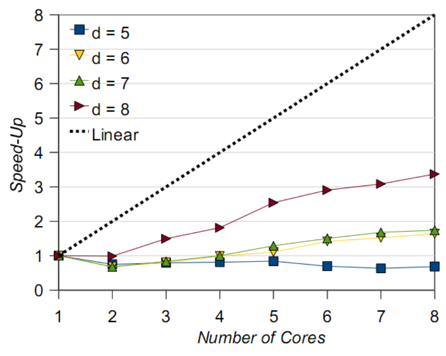

Fig. 7. Speed-Up of Locked (Independent, n = 1M).

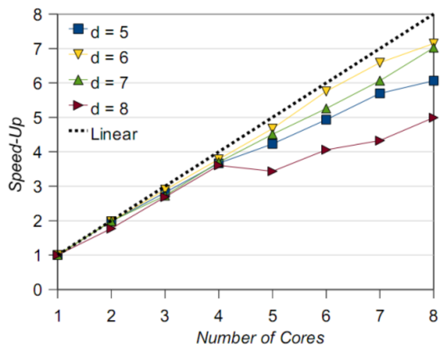

Fig. 8. Speed-Up of Lock-Free (Independent, n = 1M).

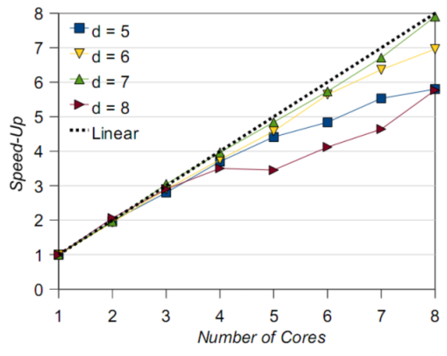

Fig. 9. Speed-Up of Lazy List (Independent, n = 1M).

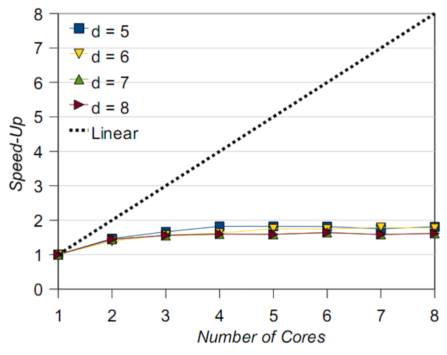

Fig. 10. Speed-Up of Distributed (Independent, n = 1M).

We will compare the following algorithms: (i) pskyline (ii) the continuously-locked parallel BNL (referred to as Locked in the following), (iii) the Lazy List parallel BNL (Lazy List), (iv) the lock-free parallel BNL (Lock-Free), and (v) the distributed merging method from Section 3.1 (Distributed; to provide a fair comparison here, we used BNL as underlying sequential algorithm). Unless stated otherwise, all experiments have been performed on one up to eight CPU cores.

4.1 Memory Usage and the Role of pskyline

As discussed in Section 3.1, pskyline relies on the sskyline algorithm for computing local skylines, which subsequently are merged by a parallel merging scheme. As sskyline requires data to be present in main memory, this imposes further restrictions on pskyline as well. Investigating this issue in detail is particularly interesting since pskyline has been reported to perform extremely well if the whole database can be loaded into main memory [16]. Since such a scenario is hardly realistic in database retrieval. In the following, we investigated the performance of pskyline under main memory restrictions.

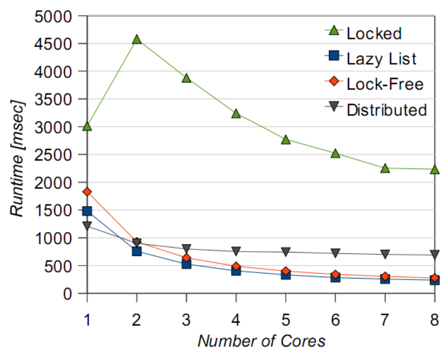

Fig. 11. Runtime comparison (Independent, n =100k, d = 7).

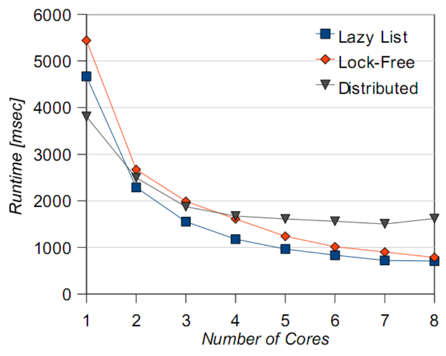

Fig. 12. Runtime comparison (Independent, n = 10M, d = 5).

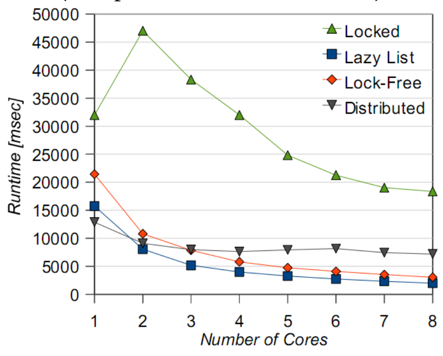

Fig. 13. Runtime comparison (Independent, n = 1M, d = 7).

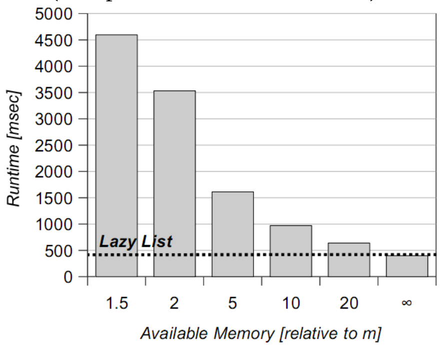

Fig. 14. Runtime of pskyline (Independent, n = 1M, d = 7).

We evaluated pskyline exemplarily on the Independent dataset with $n = 1M$ and $d = 6$ using a fixed number of $r = 8$ CPU cores, resulting in a skyline size of $m = 5577$ in average. We restricted the main memory available to pskyline relative to the skyline size m. In particular, we tested pskyline's performance with memory sizes $1.5m$, $2m$, $5m$, $10m$, and $20m$. Finally, in order to allow the algorithm to perform with its maximal performance, we allow it to load the full database into main memory. We also abstained from writing unused working sets to the hard disk as proposed in the original work on pskyline [16], thus giving pskyline a significant performance boost. Our results are presented in Figure 14, showing the computation time in milliseconds for different main memory sizes.

As a comparison baseline, we also evaluated Lazy List on the same dataset (indicated by the dotted line in Figure 14). In fact, Lazy List only requires as much main memory as necessary to hold the result set. It can be observed that the performance of pskyline suffers severely under main memory restrictions. However, if no memory restrictions are applied, pskyline shows competitive and even slightly better performance to Lazy List. These general observations also hold for other scenarios, e.g. using the Correlated or Anticorrelated data. To summarize, pskyline performs very well in main memory database scenarios, but on the whole is memory inefficient. Due to this fact, we will exclude it from the following evaluations. In contrast, the memory consumption of Locked, Lazy List, and Lock-Free had always been around the number $1.5m$, thus being close the optimum of m. The Distributed algorithm requires roughly $2m$ of main memory.

4.2 Speed-Up and Scaling

In this section, we evaluate and compare the multiprocessing scalability and overall performance of the remaining four algorithms. For pure scalability observations, we will use the Independent data set, where n = 1M and d = 5−8. Setting d to higher values gives unrealistically large result sets ($m \gg 50$K). Each algorithm has been executed on $r = 1, \dots, 8$ cores. The respective results are presented in Figures 7−10. For the Locked algorithm, only a small speed-up can be observed due to high lock contention thrashing. The speedup factor grows with m, and peaks in our test with a value of 3.3 for 8 cores and $d = 8$. It is also interesting to note that the algorithm performs better using just one core than using two of them. This can be explained by the Java VM being able to detect the needlessness of locking, if there are no concurrent accesses, thus dynamically disabling the use of locks.

In contrast, Lazy List (Fig. 9) and Lock-Free (Fig. 8) show significantly better speed-up behavior compared to the Locked version. While showing similar performance to each other, Lazy List performs slightly better overall. It can be observed that both algorithms show nearly linear scaling up to four cores. Starting with the fifth core, the performance gain moderately ceases for both algorithms and data sets with $d = 5$ and $d = 8$, but only slightly degenerates for the other cases. This phenomenon may be explained with decreasing cache locality and increasing communication overhead as our test system uses two quad-core processors. Starting with the fifth core, the second processor must constantly communicate with the first one over the slower Front Side Bus (compared to communication among cores within a single processor). However, we also expect a nearly linear speed-up for true 8-core-processors, which will be available in the near future. For both Lazy List and Lock-Free, we measured maximum speed-ups of 7.9 (in case $d = 7$) and 5.8 (in case $d = 8$) using 8 cores.

Finally, the Distributed algorithm (Fig. 10) shows almost no speedup. Peak scaling was measured with a speed-up factor of 1.74 for 8 cores and d = 7.

For assessing the runtime performance of the algorithms in absolute numbers, we measured the computation time for $n = 100K$, 1M and $d = 7$ as well as for $n = 10$M and $d = 5$. These cases have been selected according to their representativeness regarding practical cases in preference-based retrieval. All results can be found in Figures 11−13. During this evaluation, it turned out that for the 8-core cases, Lazy List shows the highest performance (1992 ms for $n = 1$M), followed by Lock-Free (3052 ms for $n = 1$M), Distributed (7802 ms for $n = 1$M), and, finally, Locked (18,375 ms for $n = 1$M). Please note that Locked has been left out in Fig. 13 due to its extremely poor performance.

In summary, we have shown that Lazy List as well as the Lock Free algorithm show very good scaling behavior and overall performance, while ensuring near optimal memory efficiency due to their block-nested-loops lineage.

Source Code and Repeatability: In order to provide a reliable and referencable foundation for further algorithm development, all our presented algorithms can be accessed and downloaded at http://www. ifis.cs.tu-bs.de/javalib/skysim.zip.

5 CONCLUSION AND OUTLOOK

In this paper, we established a design space for parallel database retrieval algorithms, in particular focusing on skyline algorithms as a representative example. Identifying basic operations of those algorithms, we investigated their respective potential for parallelization using different techniques. Finally, we designed exemplarily two innovative and highly scalable algorithms based on the popular block-nested-loops class for the scenario of shared-memory multi-processor systems. In our extensive evaluations, we showcased the superior characteristics and scalability of these algorithms in different settings. Although we did not exploit any special skyline-specific optimization techniques, we were able to outperform state-of-the-art approaches to skyline computation on multiprocessor systems in these scenarios. Especially our lazy list algorithm and lock-free BNL algorithm showed excellent overall performance with nearly linear scaling.

The design considerations and implementation techniques presented in this paper pave the way for also tapping the parallel potential of state-of-the art algorithms also for other retrieval scenarios. Moreover, future work will adapt our techniques to further optimizations for skyline algorithms like tree-based indexing or dynamic sorting.

References

1. J. S. Vitter, “Algorithms and data structures for external memory,” Foundations and Trends in Theoretical Computer Science, vol. 2, no. 4, pp. 305–474, 2006.
2. J. Larus, “Spending Moore’s dividend,” Communications of the ACM, vol. 52, no. 5, pp. 62–69, 2009.
3. S. Börzsönyi, D. Kossmann, and K. Stocker, “The Skyline operator,” in Proceedings of the 17th International Conference on Data Engineering (ICDE 2001). IEEE Computer Society, 2001, pp. 421430.
4. J. Chomicki, “Preference formulas in relational queries,” ACM Transactions on Database Systems, vol. 28, no. 4, pp. 427–466, 2003.
5. P. Godfrey, R. Shipley, and J. Gryz, “Algorithms and analyses for maximal vector computation,” The VLDB Journal, vol. 16, no. 1, pp. 5–28, 2007.
6. M. Morse, J. M. Patel, and H. V. Jagadish, “Efficient skyline computation over low-cardinality domains,” in Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB 2007), ACM Press, 2007, pp. 267–278.
7. T. Preisinger and W. Kießling, “The Hexagon algorithm for Pareto preference queries,” in Proceedings of the 3rd Multidisciplinary Workshop on Advances in Preference Handling (M-PREF 2007), 2007.

8. P.-K. Eng, B. C. Ooi, and K.-L. Tan, “Indexing for progressive skyline computation,” Data 4 Knowledge Engineering, vol. 46, no. 2, pp. 169–201, 2003.
9. J. Chomicki, P. Godfrey, J. Gryz, and D. Liang, “Skyline with presorting,” in Proceedings of the 19th International Conference on Data Engineering (ICDE 2003), U. Dayal, K. Ramamritham, and T. M. Vijayaraman, Eds. IEEE Computer Society, 2003, pp. 717–719.
10. 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.
11. I. Bartolini, P. Ciaccia, and M. Patella, “Efficient sort-based skyline evaluation,” ACM Transactions on Database Systems, vol. 33, no. 4, 2008.
12. J. L. Bentley, K. L. Clarkson, and D. B. Levine, “Fast linear expected-time algorithms for computing maxima and convex hulls,” Algorithmica, vol. 9, no. 2, pp. 168–183, 1993.
13. C. Daskalakis, R. M. Karp, E. Mossel, S. Riesenfeld, and E. Verbin, “Sorting and selection in posets,” in Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2009), C. Mathieu, Ed. SIAM, 2009, pp. 392–401.
14. R. Torlone and P. Ciaccia, “Finding the best when it’s a matter of preference,” in Proceedings of the 10th Italian Symposium on Advanced Database Systems (SEBD 2002), P. Ciaccia, F. Rabitti, and G. Soda, Eds., 2002, pp. 347–360.
15. P. Boldi, F. Chierichetti, and S. Vigna, “Pictures from Mongolia: Extracting the top elements from a partially ordered set,” Theory of Computing Systems, vol. 44, no. 2, pp. 269288, 2009.
16. S. Park, T. Kim, J. Park, J. Kim, and H. Im, “Parallel skyline computation on multicore architectures,” in Proceedings of the 25th International Conference on Data Engineering (ICDE 2009). IEEE Computer Society, 2009, pp. 760–771.
17. 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,” Journal of the ACM, vol. 25, no. 4, pp. 536–543, 1978.
18. M. Sun, “A primogenitary linked quad tree data structure and its application to discrete multiple criteria optimization,” Annals of Operations Research, vol. 147, no. 1, pp. 87–107, 2006.
19. R. Bayer and M. Schkolnick, “Concurrency of operations on B-trees,” Acta Informatica, vol. 9, no. 1, pp. 1–21, 1977.
20. S. Heller, M. Herlihy, V. Luchang co, M. Moir, W. N. Scherer, III, and N. Shavit, “A lazy concurrent list-based set algorithm,” Parallel Processing Letters, vol. 17, no. 4, pp. 411–424, 2007.
21. T. L. Harris, “A pragmatic implementation of non-blocking linked-lists,” in Proceedings of the 15th International Conference on Distributed Computing (DISC 2001), ser. Lecture Notes in Computer Science, J. Welch, Ed., vol. 2180. Springer, 2001, pp. 300–314.
22. M. M. Michael, “High performance dynamic lock-free hash tables and list-based sets,” in Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 2002). ACM Press, 2002, pp. 73–82.
23. J. Levandoski , M. Mokbel, M. Khalefa: FlexPref: A Framework for Extensible Preference Evaluation in Database Systems. International Conference on Data Engineering (ICDE), Long Beach, CA, USA, 2010.
24. A. Cosgaya-Lozano, A. Rau-Chaplin, and N. Zeh, “Parallel computation of skyline queries,” in Proceedings of the 21st International Symposium on High Performance Computing Systems and Applications (HPCS 2007). IEEE Computer Society, 2007.