External-Memory Algorithms with Applications in Geographic Information Systems*

Lars Arge†

Department of Computer Science
Duke University
Durham, NC 27708–0129
USA

December, 1996
(updated September, 1997)

Abstract

In the design of algorithms for large-scale applications it is essential to consider the problem of minimizing Input/Output (I/O) communication. Geographical information systems (GIS) are good examples of such large-scale applications as they frequently handle huge amounts of spatial data. In this note we survey the recent developments in external-memory algorithms with applications in GIS. First we discuss the Aggarwal-Vitter I/O-model and illustrate why normal internal-memory algorithms for even very simple problems can perform terribly in an I/O-environment. Then we describe the fundamental paradigms for designing I/O-efficient algorithms by using them to design efficient sorting algorithms. We then go on and survey external-memory algorithms for computational geometry problems—with special emphasis on problems with applications in GIS—and techniques for designing such algorithms: Using the orthogonal line segment intersection problem we illustrate the *distribution-sweeping* and the *buffer tree* techniques which can be used to solve a large number of important problems. Using the batched range searching problem we introduce the *external segment tree*. We also discuss an algorithm for the reb/blue line segment intersection problem—an important subproblem in map overlaying. In doing so we introduce the *batched filtering* and the *external fractional cascading* techniques. Finally, we shortly describe TPIE—a Transparent Parallel I/O Environment designed to allow programmers to write I/O-efficient programs.

1 Introduction

Traditionally when designing computer programs people have focused on the minimization of the internal computation time and ignored the time spent on Input/Output (I/O). Theoretically one of the most commonly used machine models when designing algorithms is the Random Access Machine (RAM) (see e.g. [7, 88]). One main feature of the RAM model is that its memory consists of an (infinite) array, and that any entry in the array can be accessed at the same (constant)

*Lecture notes for CISM Advanced School on Algorithmic Foundations of Geographic Information Systems, September 16–20, 1996, Udine, Italy.

†Supported in part by the U.S. Army Research Office MURI grant DAAH04–96–1–0013. Part of this work was done while the author was with BRICS, Department of Computer Science, University of Aarhus, Denmark, supported in part by the ESPRIT Long Term Research Programme of the EU under project number 20244 (ALCOM–IT). Email: large@cs.duke.edu.

cost. Also in practice most programmers conceptually write programs on a machine model like the RAM. In an UNIX environment for example the programmer thinks of the machine as consisting of a processor and a huge ("infinite") memory where the contents of each memory cell can be accessed at the same cost (Figure 1). The task of moving data in and out of the limited main memory is then entrusted to the operating system. However, in practice there is a huge difference in access time of fast internal memory and slower external memory such as disks. While typical access time of main memory is measured in nanoseconds, a typical access time of a disk is on the order of milliseconds [36]. So roughly speaking there is a factor of a million in difference in the access time of internal and external memory, and therefore the assumption that every memory cell can be accessed at the same cost is questionable, to say the least!

In many modern large-scale applications the communication between internal and external memory, rather than the internal computation time, is actually the bottleneck in the computation. As geographic information systems (GIS) frequently store, manipulate, and search through enormous amounts of spatial data [42, 56, 67, 80, 87] they are good examples of such large-scale applications. The amount of data manipulated in such systems is often too large to fit in main memory and must reside on disk, hence the I/O communication can become a very severe bottleneck. An especially good example is NASA's EOS project GIS system [42], which is expected to manipulate petabytes (thousands of terabytes, or millions of gigabytes) of data!

The effect of the I/O bottleneck is getting more pronounced as internal computation gets faster, and especially as parallel computing gains popularity [76]. Currently, technological advances are increasing CPU speeds at an annual rate of 40–60% while disk transfer rates are only increasing by 7–10% annually [79]. Internal memory sizes are also increasing, but not nearly fast enough to meet the needs of important large-scale applications.

Modern operating systems try to minimize the effect of the I/O bottleneck by using sophisticated paging and prefetching strategies in order to assure that data is present in internal memory when it is accessed. However, these strategies are general purpose in nature and therefore they cannot take full advantage of the properties of a specific problem. Instead we could hope to design more efficient algorithms by explicitly considering the I/O communication when designing algorithms for specific problems. This could e.g. be done by designing algorithms for a model where the memory system consists of a main memory of limited size and a number of external memory devices (Figure 2), where the memory access time depends on the type of memory accessed. Algorithms designed for such a model are often called *external-memory (or I/O) algorithms*. In this note we consider basic paradigms for designing efficient external-memory algorithms and external-memory computational geometry algorithms with applications in GIS.

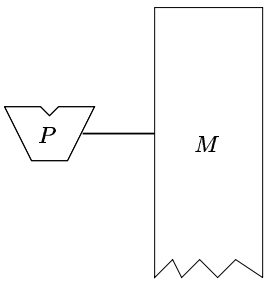

Figure 1: A RAM-like machine model.

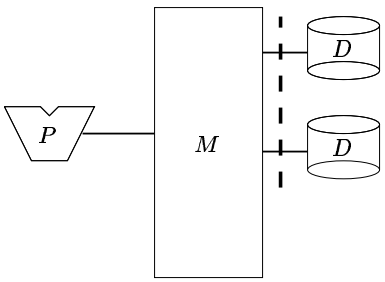

Figure 2: A more realistic model.

1.1 The Parallel Disk Model

Accurately modeling memory and disk systems is a complex task [79]. The primary feature of disks that we want to model is their extremely long access time relative to that of internal memory. In order to amortize the access time over a large amount of data, typical disks read or write large blocks of contiguous data at once. Therefore we use a theoretical model with the following parameters [6]:

$$\begin{aligned} N &= \text{number of elements in the problem instance;} \\ M &= \text{number of elements that can fit into internal memory;} \\ B &= \text{number of elements per disk block;} \end{aligned}$$

where $M < N$ and $1 \leq B \leq M/2$.

In order to study the performance of external-memory algorithms, we use the standard notion of I/O complexity [6]. We define an I/O operation to be the process of simultaneously reading or writing a block of B contiguous data elements to or from the disk. As I/O communication is our primary concern, we define the I/O complexity of an algorithm simply to be the number of I/Os it performs. Internal computation is free in the model. Thus the I/O complexity of reading all of the input data is equal to N/B. Depending on the size of the data elements, typical values for workstations and file servers in production today are on the order of $M = 10^6$ or 10^7 and $B = 10^3$. Large-scale problem instances can be in the range $N = 10^{10}$ to $N = 10^{12}$.

An increasingly popular approach to further increase the throughput of the I/O system is to use a number of disks in parallel [50, 51, 97]. Several authors have considered an extension of the above model with a parameter D denoting the number of disks in the system [21, 73, 71, 72, 97]. In *the parallel disk model* [97] one can read or write one block from each of the D disks simultaneously in one I/O. The number of disks D range up to 10^2 in current disk arrays.

The parallel disk model corresponds to the one shown in Figure 2, where we only count the number of blocks of B elements moved across the dashed line. Of course the model is designed for theoretical considerations and is thus very simple in comparison with a real system. For example one cannot always ignore internal computation time and one could try to model more accurately the fact that (in single user systems at least) reading a block from disk in most cases decreases the cost of reading the block succeeding it. Also today the memory of a real machine is typically made up of not only two but several levels of memory (e.g. on-chip data and instruction cache, secondary cache, main memory and disk) between which data are moved in blocks (Figure 3). The memory in such a hierarchy gets larger and slower the further away from the processor one gets, but as the access time of the disk is extremely large compared to that of all the other levels of memory we can in most practical situations restrict our attention to the two level case. Thus theoretical

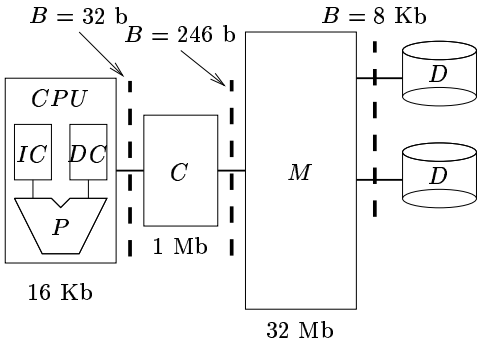

Figure 3: A "real" machine with typical memory and block sizes [36].

results obtained in the parallel disk model can help to gain valuable insight. This is supported by experimental results which show that implementing algorithms designed for the model can lead to significant runtime improvements in practice [32, 33, 89, 92]. We will discuss some of these experiments in later sections.

Finally, it should be mentioned that several authors have considered extended theoretical models that try to model the hierarchical nature of the memory of real machines [2, 3, 4, 5, 8, 60, 81, 95, 98, 96], but such models quickly become theoretically very complicated due to the large number of parameters. Therefore only very basic problems like sorting have been considered in these models.

1.2 Outline

In this note we survey the basic paradigms for designing efficient external-memory algorithms and especially for designing external-memory algorithms for computational geometry problems. As the area of external-memory algorithms is relatively young it is difficult to give really good examples of practical GIS applications. Therefore this note focuses on fundamental external-memory design techniques more than on algorithms for specific GIS problems. The presentation is survey-like with a more detailed discussion of the most important techniques and algorithms.

In Section 2 we first illustrate why normal internal-memory algorithms for even very simple problems can perform terribly when the problem instances get just moderately large. We also discuss the theoretical I/O lower bounds on fundamental problems like sorting. In Section 3 we then discuss the fundamental paradigms for designing I/O-efficient algorithms. We do so by using the different paradigms to design theoretically optimal sorting algorithms. Many problems in computational geometry are abstractions of important GIS operations, and in Section 4 we survey techniques and algorithms in external-memory computational geometry. We also discuss some experimental results. Finally, we in Section 5 shortly describe a Transparent Parallel I/O Environment (TPIE) designed by Vengroff [89] to allow programmers to write I/O-efficient programs.

We assume that the reader has some basic knowledge about e.g. fundamental sorting algorithms and data structures like balanced search trees (especially B-trees) and priority queues. We also assume that the reader is familiar with asymptotic notation ($O(\cdot)$, $\Omega(\cdot)$,$\Theta(\cdot)$). One excellent textbook covering these subjects is [40].

2 RAM-Complexity and I/O-Complexity

In order to illustrate the difference in complexity of a problem in the RAM model and the parallel disk model, consider the following simple problem: We are given an N-vertex linked list stored as an (unsorted) sequence of vertices, each with a pointer to the successor vertex in the list (Figure 4). Our goal is to determine for each vertex the number of links to the end of the list. This problem is normally referred to as the *list ranking* problem.

In internal memory the list ranking problem is easily solved in $O(N)$ time. We simply traverse the list by following the pointers, and rank the vertices $N-1, N-2$ and so on in the order we meet them. In external memory, however, this simple algorithm could perform terribly. To illustrate

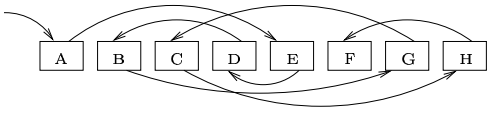

Figure 4: List ranking problem.

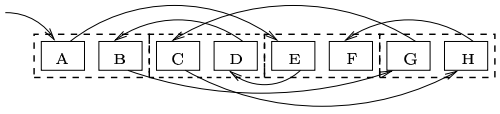

Figure 5: List ranking in external memory ($B = 2$).

this imagine that we run the algorithm on a (rather memory limited) machine where the internal memory is capable of holding two blocks with two data elements each. Assume furthermore that the operating system controlling the I/O uses a least recently used (LRU) paging strategy, that is, when a data item in a block not present in internal memory is accessed the block containing the least recently used data items is flushed from internal memory in order to make room for the new block. Finally, assume that the list is blocked as indicated in Figure 5. First we load block 1 and give A rank $N-1$. Then we follow the pointer to E, that is, we load block 3 and rank E. Then we follow the pointer to D, loading block 2 while removing block 1 from internal memory. Until now we have made an I/O every time we follow a pointer. Now we follow the pointer to B, which means that we have to load block 1 again. To do so we have to remove block 3 from internal memory. Next we remove block 2 to get room for block 4. This process continues and we see that we do not utilize the blocked disk access, but do an I/O every time we access a vertex. Put another way we use $O(N)$ I/Os instead of $O(N/B)$ I/Os which would correspond to the $O(N)$ internal-memory RAM bound because $O(N/B)$ is the number of I/Os we need to read all N vertices. As typical practical values of B are measured in thousands this difference can be extremely significant in practice.

In general the above type of behavior is characteristic for internal-memory algorithms when analyzed in an I/O-environment, and the main reason why many applications experience a dramatic decrease in performance when the problem instances get larger than the available internal memory. The lesson one should learn from this is that one should be very careful about following pointers, and be careful to ensure a high degree of locality in the access to data (what is normally referred to as *locality of reference*).

2.1 Fundamental External-Memory Bounds

After illustrating how simple internal-memory algorithms can have a terrible performance in an I/O environment, let us review the fundamental theoretical bounds in the parallel disk model. For simplicity we give all the bounds in the one-disk model. In the general D-disk model they should all be divided by D.

Initial theoretical work on I/O-complexity was done by Floyd [47] and by Hong and Kung [57] who studied matrix transposition and fast Fourier transformation in restricted I/O models. The general I/O model was introduced by Aggarwal and Vitter [6] and the notion of parallel disks was introduced by Vitter and Shriver [97]. The latter papers also deal with fundamental problems such as permutation, sorting and matrix transposition, and a number of authors have considered the difficult problem of sorting optimally on parallel disks [5, 21, 73, 71]. The problem of implementing various classes of permutations has been addressed in [38, 39, 41]. More recently researchers have moved on to more specialized problems in the computational geometry [12, 14, 19, 32, 53, 99], graph theoretical [13, 14, 32, 34, 52, 66] and string processing areas [15, 35, 45, 46].

As already mentioned the number of I/O operations needed to read the entire input is N/B and for convenience we call this quotient n. One normally uses the term *scanning* to describe the fundamental primitive of reading (or writing) all elements in a set stored contiguously in external memory by reading (or writing) the blocks of the set in a sequential manner in $O(n)$ I/Os. Furthermore, one says that an algorithm uses a linear number of I/O operations if it uses $O(n)$ I/Os. Similarly, we introduce $m = M/B$ which is the number of blocks that fit in internal memory. Aggarwal and Vitter [6] showed that the number of I/O operations needed to sort N elements is $\Omega(n \log_m n)$, which is then the external-memory equivalent of the well-known $\Omega(N \log_2 N)$ internal-memory bound.[1] Furthermore, they showed that the number of I/Os needed to rearrange N elements according to a given permutation is $\Omega(\min\{N, n \log_m n\})$.

[1] We define $\log_m n = \max\{1, \log n / \log m\}$. For extremely small values of M and B the comparison model is assumed in the sorting lower bound—see also [16, 17]

Taking a closer look at the above bounds for typical values of B and M reveals that because of the large base of the logarithm, $\log_m n$ is less than 3 or 4 for all realistic values of N and m. Thus in practice the important term is the B-term in the denominator of the $O(n \log_m n) = O(\frac{N}{B} \log_m n)$ bound, and an improvement from an $\Omega(N)$ bound (which we have seen is the worst case I/O performance of many internal-memory algorithms) to the sorting bound $O(n \log_m n)$ can be extremely significant in practice. Also the small value of $\log_m n$ in practice means that in all realistic cases the sorting term in the permutation bound will be smaller than N. Thus $\min\{N, n \log_m n\} = n \log_m n$ and the problem of permuting is as hard as the more general problem of sorting. This fact is one of the important facts distinguishing the parallel disk model from the RAM-model, as any permutation can be performed in $O(N)$ time in the latter. Actually, it turns out that the permutation bound is a lower bound on the list ranking problem discussed above [34], and as an $O(n \log_m n)$ I/O algorithm is known for the problem [12, 32, 34] we have an asymptotically optimal algorithm for all realistic systems. Even though the algorithm is more complicated than the simple RAM algorithm, Vengroff [90] has performed simulations showing that on large problem instances it has a much better performance than the simple internal-memory algorithm. In the parallel algorithm (PRAM) world list ranking is a very fundamental graph problem which extracts the essence in many other problems, and it is used as an important subroutine in many parallel algorithms [9]. This turns out also to be the case in external memory [32, 34].

2.2 Summary

- RAM algorithms typically use $\Omega(N)$ I/Os when analyzed in parallel disk model.
- Typical bounds in one-disk model (divide by D in D-disk model):
  - Scanning bound: $\Theta(\frac{N}{B}) = \Theta(n)$.
  - Sorting bound: $\Theta(\frac{N}{B} \log_{M/B} \frac{N}{B}) = \Theta(n \log_m n)$.
  - Permutation bound: $\Theta(\min\{N, n \log_m n\})$.
- In practice:
  - $\log_m n < 4$ and B is the important term in $O(\frac{N}{B} \log_m n)$ bound. Going from $\Omega(N)$ to $O(n \log_m n)$ algorithm extremely important.
  - Permutation bound equal to sorting bound.

3 Paradigms for Designing I/O-efficient Algorithms

Originally Aggarwal and Vitter [6] presented two basic paradigms for designing I/O-efficient algorithms; the *merging* and the *distribution* paradigms. In Section 3.1 and 3.2 we demonstrate the main ideas in these paradigms by showing how to use them to sort N elements in the optimal number of I/Os. Another important paradigm is to construct I/O-efficient versions of commonly used data structures. This enables the transformation of efficient internal-memory algorithms to efficient I/O-algorithms by exchanging the data structures used in the internal algorithms with the external data structures. This approach has the extra benefit of isolating the I/O-specific parts of an algorithm in the data structures. We call the paradigm the *data structuring* paradigm, and in Section 3.3 we illustrate it by way of the so called *buffer tree* designed in [12]. As we shall see later I/O-efficient data structures turn out to be a very powerful tool in the development of efficient

I/O algorithms. For simplicity we only discuss the paradigms in the one disk ($D = 1$) model. In Section 3.4 we then briefly discuss how to make the sorting algorithms work in the general model.

3.1 Merge Sort

External merge sort is a generalization of internal merge sort. First, in the "run formation phase", N/M ($= n/m$) sorted "runs" are formed by repeatedly filling up the internal memory, sorting the elements, and writing them back to disk. The run formation phase requires $2n$ I/Os as we read and write each block ones. Next we continually merge m runs together to a longer sorted run, until we end up with one sorted run containing all the elements—refer to Figure 6.

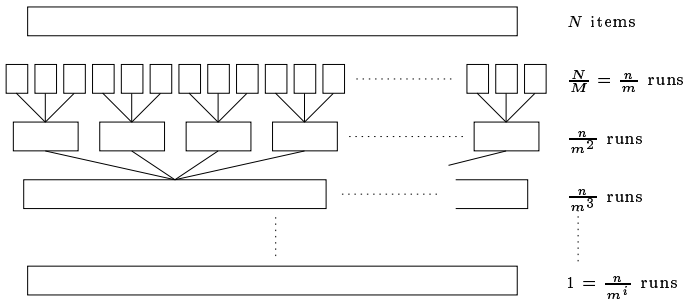

Figure 6: Merge sort.

The crucial property is that we can merge m runs together in a linear number of I/Os. To do so we simply load a block from each of the m runs and collect and output the B smallest elements. We continue this process until we have processed all elements in all runs, loading a new block from a run every time a block becomes empty. Since there are $\log_m n/m$ levels in the merge process, and since we only use $2n$ I/O operations on each level, we in total use $2n + 2n \cdot \log_m n/m = 2n + 2n(\log_m n - 1) = 2n \log_m n$ I/Os and have thus obtained an optimal $O(n \log_m n)$ algorithm.

3.2 Distribution Sort

External distribution sort is in a sense the reverse of merge sort and the external-memory equivalent of quick sort. Like in merge sort the distribution sort algorithm consists of a number of levels each using a linear number of I/Os. However, instead of repeatedly merging m run together, we repeatedly distribute the elements in a "bucket" into m smaller "buckets" of equal size. All elements in the first of these smaller buckets are smaller than all elements in the second bucket and so on. The process continues until the elements in a bucket fit in internal memory, in which case the bucket is sorted using an internal-memory sorting algorithm. The sorted sequence of elements is then obtained by appending the small sorted buckets—refer to Figure 7.

Like m-way merge, m-way distribution can also be performed in a linear number of I/Os, by just keeping a block in internal memory for each of the buckets we are distributing elements into—writing it to disk when it becomes full. However, in order to distribute the N elements in a bucket into m smaller buckets we need to find m "pivot" elements among the N elements, such that the buckets each defined by two such pivot elements are of equal size (corresponding to finding the median in the quicksort algorithm). In order to do so I/O-efficiently we need to decrease the distribution factor and distribute the elements in a bucket into $\sqrt{m}$ instead of m smaller buckets. If the elements are divided perfectly among the buckets this will only double the number of levels as $\log_{\sqrt{m}} n = \log_m n / \log_m \sqrt{m} = 2 \log_m n$. Thus we will still obtain an $O(n \log_m n)$ algorithm if we can process each level in a linear number of I/Os.

The obvious way to find the $\sqrt{m}$ pivot elements would be to find every $N/\sqrt{m}$'th element using the k-selection algorithm $\sqrt{m}$ times. The k-selection algorithm [26] finds the k'th smallest

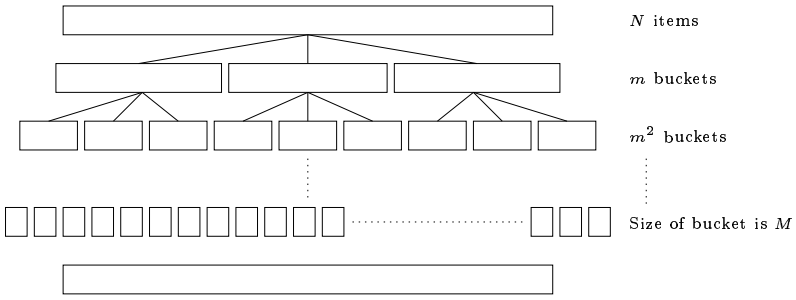

Figure 7: Distribution sort.

element in $O(N)$ time in internal memory and it can easily be modified to work in $O(n)$ I/Os in external memory. The $\sqrt{m}$ elements found this way would give us buckets of exact equal size, but unfortunately we would use $O(n \cdot \sqrt{m})$ and not $O(n)$ I/Os to compute them. Therefore we first choose $4N/\sqrt{m}$ of the N elements by sorting the N/M memory loads individually and choosing every $\sqrt{m}/4$'the element from each of them. Then we can use k-selection $\sqrt{m}$ times on these elements to obtain the pivot elements using only $\sqrt{m} \cdot O(n/4\sqrt{m}) = O(n)$ I/Os. However, now we cannot be sure that the $\sqrt{m}$ buckets have equal size $N/\sqrt{m}$, but fortunately one can prove that they are of approximately equal size, namely that no bucket contains more than $5N/4\sqrt{m}$ elements [6, 63]. Thus the number of levels in the distribution is less than $\log_{4/5\sqrt{m}} n/m = O(\log_m n)$ and the overall complexity remains the optimal $O(n \log_m n)$ I/Os.

Even though merge sort is a lot simpler and efficient in practice than distribution sort [64], the distribution paradigm is the most frequently used of the two paradigms. Mainly two factors make distribution sort less efficient in practice than merge sort, namely the larger number of levels in the recursion and the computation of the pivot elements. Especially the last factor (the k-selection algorithm) can be very I/O expensive. However, in many applications it turns out that one can compute all the pivot elements used during the whole algorithm before the actual distribution starts, and thus *both* avoid the expensive k-selection algorithm and obtain an m distribution factor. We will see an example of this in Section 4.1.

Both the distribution sort and merge sort algorithm demonstrates two of the most fundamental and useful features of the I/O-model, which is used repeatedly when designing I/O algorithms. First the fact that we can do m-way merging or distribution in a linear number of I/O operations, and secondly that we can solve a complicated problem in a linear number of I/Os *if* it fits in internal memory. In the two algorithms the sorting of a memory load is an example of the last feature, which is also connected with what is normally referred to as "locality of reference"—one should try to work on data in chunks of the block (or internal memory) size, and do as much work as possible on data once it is loaded into internal memory.

3.3 Buffer Tree Sort

In internal memory we can sort N elements in $O(N \log_2 N)$ time using a balanced search tree; we simply insert all N elements in the tree one by one using $O(\log_2 N)$ time on each insertion, and then we can easily obtain the sorted set of element in $O(N)$ time. Similarly, we can use a priority queue to sort optimally; first we insert all N elements in the queue and then we perform N deletemin operations. Why not use the same algorithms in external memory, exchanging the data structures with I/O-efficient versions of the structures?

The standard well-known search tree structure for external memory is the B-tree [22, 37, 65]. On this structure insert, delete, deletemin and search operations can be performed in $O(\log_B n)$ I/Os.

Thus using the structure in the algorithms above results in $O(N \log_B n)$ I/O sorting algorithms which is a factor of $B\frac{\log m}{\log B}$ away from optimal. This factor can be very significant in practice. In order to obtain an optimal sorting algorithm we need a structure where the operations can be performed in $O(\frac{\log_m n}{B})$ I/Os.

The inefficiency of the B-tree sorting algorithm is a consequence of the fact that the B-tree is designed to be used in an "on-line" setting, where queries should be answered immediately and within a good worst-case number of I/Os, and thus updates and queries are handled on an individual basis. This way one is not able to take full advantage of the large internal memory. Actually, using a decision tree like argument as in [61], one can show that the search bound is indeed optimal in such an "on-line" setting (assuming the comparison model). However, in an "off-line" environment where we are only interested in the overall I/O use of a series of operations on the involved data structure, and where we are willing to relax the demands on the search operations, we could hope to develop data structures on which a series of N operations could be performed in $O(n \log_m n)$ I/Os in total. Below we sketch such a basic tree structure developed using what is called the *buffer tree* technique [12]. The structure can be used in the normal tree sort algorithm. In Section 3.3.1 we also sketch how the structure can be used to develop an I/O-efficient external priority queue.

Basically the buffer tree is a fan-out $m/2$ tree structure built on top of n leaves, each containing B of the N elements stored in the structure. Thus the tree has height $O(\log_m n)$—refer to Figure 8. A *buffer* of size $m/2$ *blocks* is assigned to each internal node and operations on the structure are done in a "lazy" manner. In order to insert a new element in the structure we do not (like in a normal tree) search all the way down the tree to find the place among the leaves to insert the element. Instead, we wait until we have collected a block of insertions, and then we insert this block in the buffer of the root (which is stored on disk). When a buffer "runs full" its element are then "pushed" one level down to buffers on the next level. We call this a *buffer-emptying process*, and it is basically performed as a $m/2$-way distribution step; we load the $M/2$ elements from the buffer and the $m/2$ partition elements into internal memory, sort the elements from the buffer, and write them back to the appropriate buffers on the next level. If the buffer of any of the nodes on the next level becomes full by this process the buffer-emptying process is applied recursively.

The main point is now that we can perform the buffer-emptying process in $O(m)$ I/Os, basically because the elements in a buffer fit in memory and because the fan-out of the structure is $\Theta(m)$. We use $O(m)$ I/Os to read and write all the elements, plus at most one I/O for each of the $O(m)$ children to distribute elements in non-full blocks. Thus we push $m/2$ blocks one level down the tree using $O(m)$ I/Os, or put another way, we use a constant number of I/Os to push one block one level down. In this way we can argue that every *block* is touched a constant number of times on each of the $O(\log_m n)$ levels of the tree, and thus inserting N elements (or n blocks) in the structure requires $O(n \log_m n)$ I/Os in total. Of course one also has to consider how to empty a buffer of a node on the last level in the tree, that is, how to insert elements among the leaves of the

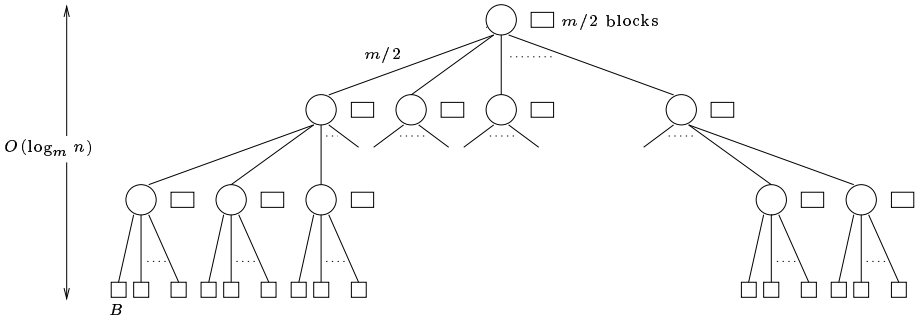

Figure 8: Buffer tree.

tree and perform rebalancing. In [12] it is shown that by using an (a, b)-tree [58] as the basic tree structure this can also be handled in the mentioned I/O bound. An (a, b)-tree is a generalization of the B-tree. Deletions (and queries) can be handled using similar ideas [12].

Note that while N insertions (and deletions) in total take $O(n \log_m n)$ I/Os, a single insertion (or deletion) can take a lot more than $O(\frac{\log_m n}{B})$ I/Os, as a single operation can result in a lot of buffer-emptying processes. Thus we do not as in the B-tree case have a *worst-case* I/O bound on performing an update. Instead we say that each operation can be performed in $O(\frac{\log_m n}{B})$ I/Os *amortized* [85].

In order to use the structure in a sorting algorithm we also need an operation that reports all the elements in the structure in sorted order. To do so we first empty all buffers in the structure using the buffer-emptying process from the root of the tree and down. As the number of internal nodes is $O(n/m)$ this can easily be done in $O(n)$ I/Os. After this all the elements are stored in the leaves and can be reported in sorted order using a simple scan. Using the buffer idea we have thus obtained a structure with the operations needed to sort N elements in $O(n \log_m n)$ I/O with precisely the same tree sort algorithm as can be used in internal memory. The algorithm has the extra benefit of isolating the I/O-specific parts in the data structure. Preliminary simulation results suggests that in practice the algorithm has a bit worse performance than merge sort, but that it performs much better than distribution sort.

3.3.1 External-Memory Priority Queue

Normally, we can use a search tree structure to implement a priority queue because we know that the smallest element in a search tree is in the leftmost leaf. Thus when we want to perform a deletemin operation we simply delete and return the element in the leftmost leaf. The same general strategy can be used to implement an external priority queue based on the buffer tree. However, in the buffer tree we cannot be sure that the smallest element is in the leftmost leaf, since there can be smaller elements in the buffers of the nodes on the leftmost path. There is, however, a simple strategy for performing a deletemin operation in the desired *amortized* I/O bound.

When we want to perform a deletemin operation, we simply do a buffer-emptying process on all nodes on the path from the root to the leftmost leaf. This requires $O(m) \cdot O(\log_m n)$ I/Os. Now we can be sure not only that the leftmost leaf consists of the B smallest elements in the tree, but also that the $m/2 \cdot B = M/2$ smallest elements in the tree are in the $m/2$ leftmost leaves. As these elements fit in internal memory we can delete them all and hold them in internal memory in order to be able to answer future deletemin operations without having to do any I/Os at all. In this way we have used $O(m \log_m n)$ I/Os to answer $M/2$ deletemin operations which means that we amortized use $O(\frac{\log_m n}{B})$ I/Os on one such operation. There is one complication, however, as insertions of small elements may be performed before we have performed $M/2$ deletemin operations. Therefore we also on each insertion check if the element to be inserted is smaller than the largest of the minimal elements we currently hold in internal memory. If this is the case we keep the new element in memory as one of the minimal elements and insert the largest of the smallest elements in the buffer tree instead. Note that this extra check do not require any extra I/Os.

To summarize we have sketched an external priority queue on which insertions and deletemin operations can be performed in $O(\frac{\log_m n}{B})$ I/Os amortized and thus we are able to sort optimally with yet another well-known algorithm. It should be noted that recently an alternative priority queue was developed in [66].

3.4 Sorting on Parallel Disks

In the previous sections we have discussed a number of sorting algorithms working in the one-disk model. As mentioned in the introduction, one approach to increase the throughput of I/O systems

is to use a number of disks in parallel. In this section we briefly survey results on sorting optimally using D independent disks. We assume that by the start of the algorithm the N elements to be sorted are spread among the D disks with n/D blocks on each disk.

One very simple method of using D disks in parallel is *disk striping*, in which the heads of the disks are moving synchronously, so that in a single I/O operation each disk reads or writes a block in the same location as each of the others. In terms of performance, disk striping has the effect of using a single large disk with block size $B' = DB$. Even though disk striping does not in theory achieve asymptotic optimality when D is very large, it is often the method of choice in practice for using parallel disks [92]. The non-optimality of disk striping can be demonstrated via the sorting bound. While sorting N elements using disk striping and one of the previously described one-disk sorting algorithms requires $O(\frac{n}{D} \log_{m/D} n)$ I/Os, the optimal bound is $O(\frac{n}{D} \log_m n)$ I/Os [6]. Note that the optimal bound gives a linear speedup in the number of disk.

In order to use the D disks optimally in merge sort we should be able to merge $\Theta(m)$ sorted runs containing N elements in $O(n/D)$ I/Os, that is, every time we do an I/O we should load $\Theta(D)$ useful blocks from the disks. However, as we only have room in internal memory for a constant number of blocks from each input run, we cannot hold D blocks from each run and just load the next D blocks once the old ones expire. Instead, every time we want to read the next block of a run, we have to predict which $\Theta(D)$ block we will need to load next and "prefetch" them together with the desired block. The prediction can be done with a technique due to Knuth called *forecasting* [65]. However, in order to prefetch the blocks efficiently they must reside on different disks, and that is the main reason why merge sorting on parallel disk is difficult—during one merge pass one has to store the output blocks on disks in such a way that they can be efficiently prefetched in the next merge pass. But the way the blocks should be assigned to disks depends on the merge steps forming the other $m-1$ runs which will participate in the next merging pass, and therefore it seems hard to figure out how to assign the blocks to disks. Nevertheless, Nodine and Vitter [73] managed to developed a (rather complicated) D-disk sorting algorithm based on merge sort. Very recently Barve et al. [21] develop a very simple and practical randomize D-disk merge sort algorithm.

Intuitively, it seems easier to make distribution sort work optimally on parallel disks. During one distribution pass we should "just" make sure to distribute the blocks belonging to the same bucket evenly among the D disks, such that we can read them efficiently in the next pass. Vitter and Shriver [97] used randomization to ensure this and developed an algorithm which performs optimally with high probability. Later Nodine and Vitter [71] managed to develop a deterministic version of D disk distribution sort. An alternative distribution-like algorithm is develop by Aggarwal and Plaxton [5].

The buffer tree sorting algorithm can also be modified to work on D disks. Recall that the buffer-emptying process basically was performed like a distribution step, where a memory load of elements were distribute to $m/2$ buffers one level down the tree. Thus using the techniques developed in [71] the buffer-emptying algorithm can be modified to work on D disks, and we obtain an optimal D-disk sorting algorithm. As already mentioned, distribution sort is rather inefficient in practice, mainly because of the overhead used to compute the pivot elements. Also the deterministic D-disk merge sorting algorithms is rather complicated. As the computation of the pivot elements is avoided in the buffer tree, the D-disk buffer tree sorting algorithm could be very efficient in practice. In the future we hope to investigate this experimentally.

3.5 Summary

- Three main paradigms: Merging, distributing, and data structuring.
- Main features used:
  - m-way merging/distribution possible in linear number of I/Os.
  - Complicated *small* problems can be solved in linear number of I/Os.
  - Buffered data structures. Using B-trees typically yields algorithms a factor B away from optimal.
- In practice:
  - All three paradigms can be used to develop optimal sorting algorithms. One-disk merge sort fastest, followed by buffer and distribution sort.

4 External-Memory Computational Geometry Algorithms

Most GIS systems at some level store map data as a number of layers. Each layer is a thematic map, that is, it stores only one type of information. Examples are a layer storing all roads, a layer storing all cities, and so on. The theme of a layer can also be more abstract, as for example a layer of population density or land utilization (farmland, forest, residential). Even though the information stored in different layers can be very different, it is typically stored as geometric information like line segments or points. A layer for a road map typically stores the roads as line segments, a layer for cities typically contains points labeled with city names, and a layer for land utilization could store a subdivision of the map into regions labeled with the use of a particular region.

One of most fundamental operations in many GIS systems is map overlaying—the computation of new scenes or maps from a number of existing maps. Some existing software packages are completely based on this operation [10, 11, 75, 87]. Given two thematic maps the problem is to compute a new map in which the thematic attributes of each location is a function of the thematic attributes of the corresponding locations in the two input maps. For example, the input maps could be a map of land utilization and a map of pollution levels. The map overlay operation could then be used to produce a new map of agricultural land where the degree of pollution is above a certain level. One of the main problems in overlaying of maps stored as line segments is "line-breaking"—the problem of computing the intersections between the line segments making up the maps. This problem can be abstracted as the in computational geometry well-known problem of red/blue line segment intersection. In this problem one is given a set of non-intersecting red segments and a set of non-intersecting blue segments and should compute all intersection red-blue segment pairs.

In general many important problems from computational geometry are abstractions of important GIS operations [43, 87]. Examples are range searching which e.g. can be used in finding all objects inside a certain region, planar point location which e.g. can be used when locating the region a given city lies in, and region decomposition problems such as trapezoid decomposition, (Voronoi or Delaunay) triangulation, and convex hull computation. The latter problems are useful for rendering and modeling. Furthermore, as mentioned in the introduction, GIS systems frequently store and manipulate enormous amounts of data, and they are thus a rich source of problems that require good use of external-memory techniques. In this section we therefore consider external-memory algorithms for computational geometry problems. Like we in the previous section focused on the fundamental paradigms for designing efficient sorting algorithms, we will present the fundamental

paradigms and techniques for designing computational geometry algorithms, and at the same time present some of the algorithms for problems with applications in GIS systems. In order to do so we define two additional parameters in our model:

$$\begin{aligned} K &= \text{number of queries in the problem instance;} \\ T &= \text{number of elements in the problem solution.} \end{aligned}$$

In analogy with the definition of n and m we define $k = K/B$ and $t = T/B$ to be respectively the number of query blocks and number of solution element blocks.

In internal memory one can prove what might be called sorting lower bounds $O(N \log_2 N + T)$ on a large number of important computational geometry problems. The corresponding bound $O(n \log_m n + t)$ can be obtained for the external versions of the problems either by redoing standard proofs [17, 53], or by using a conversion result from [16].

Computational geometry problems in external memory were first considered by Goodrich et al. [53], who developed a number of techniques for designing I/O-efficient algorithms for such problems. They used their techniques to develop I/O algorithms for a large number of important problems. In internal memory the *plane-sweep* paradigm [77] is a very powerful technique for designing computational geometry algorithms, and in [53] an external-memory version of this technique called *distribution sweeping* is developed. As the name suggests the technique relies on the distribution paradigm. In [12] it is shown how the data structuring paradigm can also be use to solve computational geometry problems. It is shown how data structures based on the buffer tree can be used in the standard internal-memory plane-sweep algorithm for a number of problems. In [53] two techniques called *batched construction of persistent B-trees* and *batched filtering* are also discussed. In [18] some results from [53, 12] are extended and generalized, and some external-memory computational geometry results are also reported in [49, 99]. In [19] efficient I/O algorithms for a large number of problems involving line segments in the plane are designed by combining the ideas of distribution sweeping, batched filtering, buffer trees and a new technique, which can be regarded as an external-memory version of *fractional cascading* [31]. Most of these problems have important applications in GIS systems. In [32, 33, 18] some experimental results on the practical performance of external-memory algorithms for computational geometry problems are reported.

We divide our survey of external-memory computational geometry into four main parts. In the next section we illustrate the distribution sweeping and the data structure paradigm using the orthogonal line segment intersection problem. We also present some experimental results. In Section 4.2 we then use the batched range searching problem to introduce the external segment tree data structure. Section 4.3 is then devoted to a discussion of the red/blue line segment intersection problem. In that section we also discuss batched filtering and external fractional cascading. Finally, we in Section 4.4 survey some other important external-memory computational geometry results.

For simplicity we restrict the discussion to the one-disk model. Some of the algorithms can be modified to work optimally in the general model and we refer the interested reader to the research papers for a discussion of this. For completeness it should be mentioned that recently a number of researchers have considered the design of worst-case efficient external-memory "on-line" data structures, mainly for (special cases of) two and three dimensional range searching [20, 25, 59, 61, 78, 84, 93]. While B-trees [22, 37, 65] efficiently support range searching in one dimension they are inefficient in higher dimensions. Similarly the many sophisticated internal-memory data structures for range searching are not efficient when mapped to external memory. This has lead to the development of a large number of structures that do not have good theoretical worst-case update and query I/O bounds, but do have good average-case behavior for common problems—see [70, 61]. Range searching is also considered in [74, 82, 83] where the problem of maintaining range trees in external memory is considered. However, the model used in this work is different from

the one considered here. In [27] an external on-line version of the topology tree is developed and this structure is used to obtain structures for a number of dynamic problems, including approximate nearest neighbor searching and closest pair maintenance. Very recently, an algorithm has been given [1] for preprocessing a TIN into an external data structure such that the contour lines of a query elevation can be computed I/O optimally.

4.1 The Orthogonal Line Segment Intersection Problem

The orthogonal line segment intersection problem is that of reporting all intersecting orthogonal pairs in a set of N line segment in the plane parallel to the axis. In internal memory a simple optimal solution to the problem based on the plane-sweep paradigm [77] works as follows (refer to Figure 9): We imagine that we sweep with a horizontal sweep line from the top to the bottom of the plane and every time we meet a horizontal segment we report all vertical segments intersecting the segment. To do so we maintain a balanced search tree containing the vertical segments currently crossing the sweep line, ordered according to x-coordinate. This way we can report the relevant segments by performing a range query on the search tree with the x-coordinates of the endpoints of the horizontal segment. To be more precise we start the algorithm by sorting all the segment endpoints by y-coordinate. We use the sorted sequence of points to perform the sweep, that is, we process the segments in endpoint y order. When the top endpoint of a *vertical* segment is reached the segment is inserted in the search tree. The segment is removed again when its bottom endpoint is reached. This way the tree at all times contains the segments intersection the sweep line. When a *horizontal* segment is reached a range query is made on the search tree. As inserts and deletes can be performed in $O(\log_2 N)$ time and range querying in $O(\log_2 N + T')$ time, where T' is the number of reported segments, we obtain the optimal $O(N \log_2 N + T)$ solution.

As discussed in Section 3.3 a simple natural external-memory modification of the plane-sweep algorithm would be to use a B-tree as the tree data structure, but this would lead to an $O(N \log_B n + t)$ I/O solution, while we are looking for an $O(n \log_m n + t)$ I/O solution. In the next two subsections we discuss I/O-optimal solutions to the problem using the distribution sweeping and buffer tree techniques.

4.1.1 Distribution Sweeping

Distribution sweeping [53] is a powerful technique obtained by combining the distribution and the plane-sweep paradigms. Let us briefly sketch how it works in general. To solve a given problem we divide the plane into m vertical *slabs*, each of which contains $\Theta(n/m)$ input objects, for example points or line segment endpoints. We then perform a vertical top to bottom sweep over all the slabs

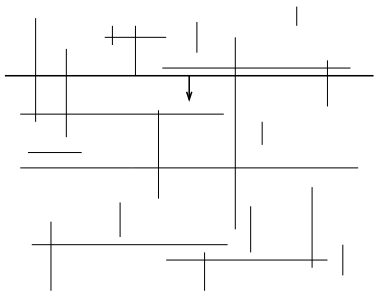

Figure 9: Solution to the orthogonal line segment intersection problem using plane-sweep.

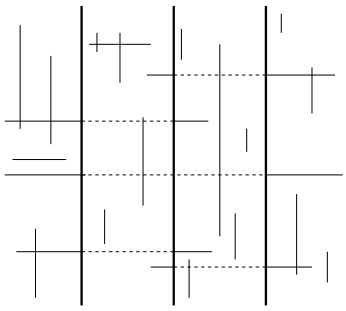

Figure 10: Solution to the orthogonal line segment intersection problem using distribution sweeping.

in order to locate components of the solution that involve interaction between objects in different slabs or objects (such as line segments) that completely span one or more slabs. The choice of m slabs ensures that one block of data from each slab fits in main memory. To find components of the solution involving interaction between objects residing in the same slab, the problem is then solved recursively in each slab. The recursion stops after $O(\log_m n/m) = O(\log_m n)$ levels when the subproblems are small enough to fit in internal memory. In order to get an $O(n \log_m n)$ algorithm we thus need to be able to perform one sweep in $O(n)$ I/Os.

To use this general technique to solve the orthogonal line segment intersection problem we first sort the endpoints of all the segments twice and create two lists, one with the endpoints sorted according to x-coordinate and the other by y-coordinate. The list sorted by y-coordinate is used to perform sweeps from top to bottom as in the plane-sweep algorithm. The list sorted according to x-coordinate is used to locate the pivot elements used throughout the algorithm to distribute the input into m vertical slabs. In this way we avoid using the complicated k-selection algorithm as discussed Section 3.2.

The algorithm now proceeds as follows (refer to Figure 10): We divide the plane into m slabs and sweep from top to bottom. When a top endpoint of a vertical segment is reached, we insert the segment in an *active list* (a stack where we keep the last block in internal memory) associated with the slab containing the segment. When a horizontal segment is reached we scan through all the active lists associated with the slabs it completely spans. During this scan we know that every vertical segment in an active list is either intersected by the horizontal segment, or will not be intersected by any of the following horizontal segments and can therefore be removed from the list. The process finds all intersections except those between vertical segments and horizontal segments (or portions of horizontal segments) that do not completely span vertical slabs (the solid parts of the horizontal segments in Figure 10). These are found after distributing the segments to the slabs, when the problem is solved recursively for each slab. A horizontal segment may be distributed to two slabs, namely the slabs containing its endpoints, but it will at most be represented twice on each level of the recursion. It is easy to realize that if T' is the number of intersections reported, one sweep can be performed in $O(n + t')$ I/Os—every vertical segment is only touched twice where an intersection is not discovered, namely when it is distributed to an active list and when it is removed again. Also blocks can be used efficiently because of the distribution factor of m. Thus by the general discussion of distribution sweeping above we report all intersections in the optimal $O(n \log_m n + t)$ I/O operations.

4.1.2 Using the Buffer tree

As discussed previously, the idea in the data structuring paradigm is to develop efficient external data structures and use them in the standard internal-memory algorithms. In order to make the plane-sweep algorithm for the orthogonal line segment intersection problem work in external memory, we thus need to extend the basic buffer tree with a rangesearch operation.

Basically a rangesearch operation on the buffer tree is done in the same way as insertions and deletions. When we want to perform a rangesearch we create a special element which is pushed down the tree in a lazy way during buffer-emptying processes, just as all other elements. However, we now have to modify the buffer-emptying process. The basic idea in the modification is the following (see [12, 14] for details). When we meet a rangesearch element in a buffer-emptying process, instructing us to report elements in the tree between x_1 and x_2, we first determine whether x_1 and x_2 are contained in the same subtree among the subtrees rooted at the children of the node in question. If this is the case we just insert the rangesearch element in the corresponding buffer. Otherwise we "split" the element in two, one for x_1 and one for x_2, and report the elements in those subtrees where *all* elements are contained in the interval $[x_1, x_2]$—refer to Figure 11. The splitting only occurs once and after that the rangesearch element is treated like inserts and deletes

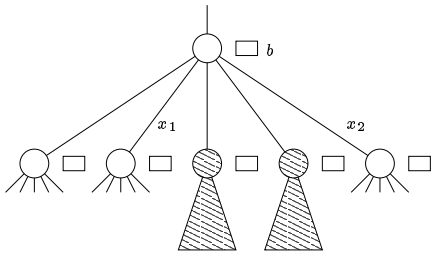

Figure 11: Buffer-emptying process with rangesearch-elements. Elements in marked subtrees are reported when buffer b is emptied

in buffer-emptying processes, except that we report the elements in the sub-trees for which all elements are contained in the interval. In [12, 14] it is show how we can report all elements in a subtree (now containing other rangesearch elements) in a linear number of I/Os. Using the normal argument it then follows that a rangesearch operation requires $O(\frac{\log_m n}{B} + t')$ I/Os amortized.

Note that the above procedure means that the rangesearch operation gets batched in the sense that we do not obtain the result of a query immediately. Actually, parts of the result will be reported at different times as the query element is pushed down the tree. However, this suffices in the plane-sweep algorithm in question, since the updates performed on the data structure do not depend on the results of the queries. This is the crucial property that has to be fulfilled in order to used the buffer tree structure. Actually, in the plane-sweep algorithm the entire sequence of updates and queries on the data structure is known in advance, and the only requirement on the queries is that they must all eventually be answered. In general such problems are known as *batched dynamic problems* [44].

To summarize, the buffer tree, extended with a rangesearch operation, can be used in the normal internal-memory plane-sweep algorithm for the orthogonal segment intersection problem, and doing so we obtain an optimal $O(n \log_m n + t)$ I/O solution to the problem.

4.1.3 Experimental Results

One main reason why we choose the orthogonal line segment intersection problem as our initial computational geometry problem is that Chiang [32, 33] has performed experiments on the practical performance of several of the described algorithms for the problem.

Chiang considered four algorithms, namely the distribution sweeping algorithm, denoted `Distribution`, and three variants of the plane-sweep algorithm, denoted `B-tree`, `234-Tree`, and `234-Tree-Core`. As discussed the theoretical I/O cost of the distribution sweeping algorithm is the optimal $O(n \log_m n + t)$. The plane-sweep algorithms differ by the sorting method used in the preprocessing step and in the dynamic data structure used in the sweep. The first variation, `B-tree`, uses external merge sort and a B-tree as search tree structure. As discussed previously this is the simple natural way to modify the plane-sweep algorithm to external memory. It uses $O(n \log_m n)$ I/Os in the preprocessing phase and $O(N \log_B n + t)$ I/Os to do the sweep. The second variation, `234-Tree`, also uses external merge sort but uses a 2-3-4 Tree [40] (a generic search tree structure equivalent to a red-black tree) as sweep structure, viewing the internal memory as having an infinite size and letting the virtual memory feature of the operating systems handle the page faults during the sweep. This way $O(N \log_2 N + t)$ I/Os is used to do the sweep. Finally, the third variation, `234-Tree-Core`, uses internal merge sort and a 2-3-4 tree, letting the operating system handle page faults at all times. The last variant is the most commonly used algorithm in practice, as viewing the internal memory as virtually having infinite size and letting the operating system

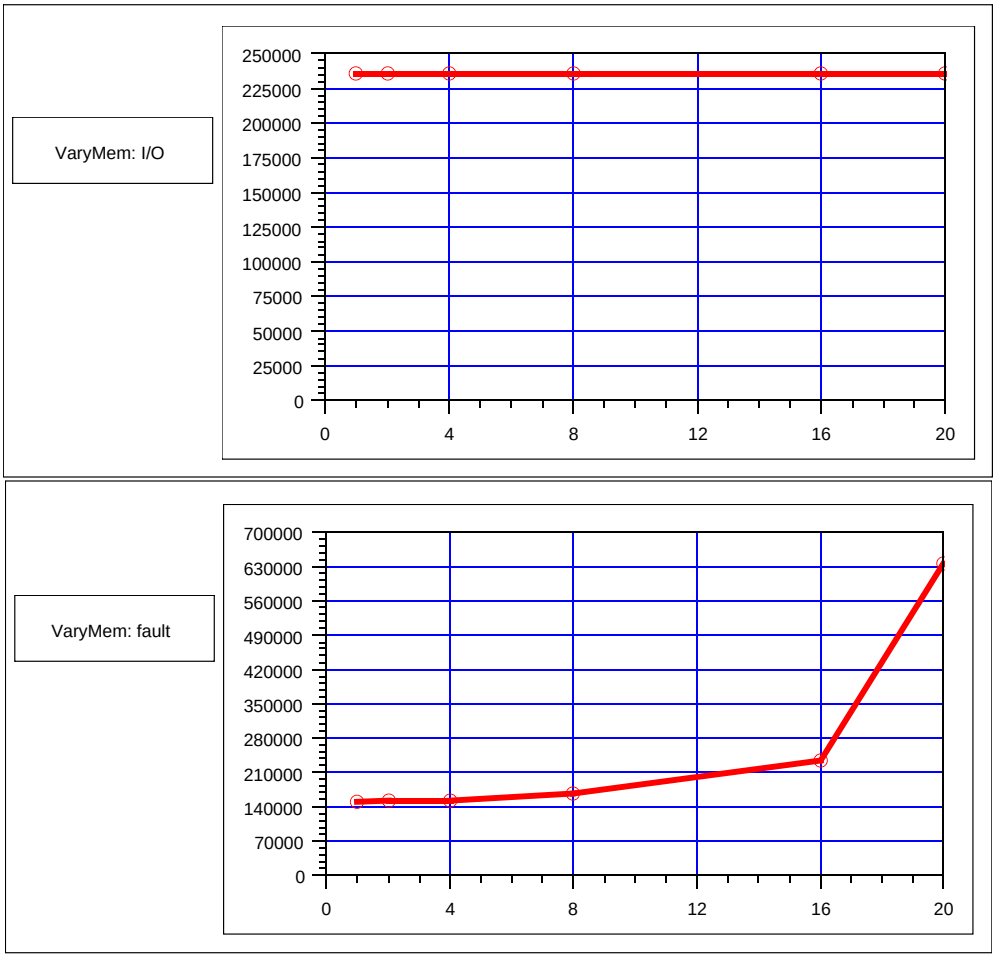

Figure 12: Number of I/Os and page faults when running `Distribution` on a data set of $1.5 \cdot 10^6$ segments with various sizes of main memory.

handle the swapping is conceptually simplest. The I/O cost is $O(N \log_2 N)$ and $O(N \log_2 N + t)$ in the two phases, respectively.

In order to compare the I/O performance of the four algorithms, Chiang generated test data with particular interesting properties. One can prove that if we just randomly generate segments with lengths uniformly distributed over [0,N], place them randomly in a square with side length N, and make horizontal and vertical segments equally likely to occur, then the expected number of intersections is $\Theta(N^2)$. In this case any algorithm must use $O(N^2/B)$ I/Os to report these intersections and thus the reporting cost will dominate in all four algorithms. Thus Chiang generated test data with only a linear number of intersections. Also, it is conceivable that the number of vertical overlaps among vertical segments at a given time decides the tree size at that moment of the plane-sweep and also the total size of the active lists at that time of the distribution sweep. Thus we would like the vertical overlap to be relatively large in order to study I/O issues. In the three data sets generated by Chiang the average number of vertical overlaps among vertical segments, that is, the average number of vertical segments intersected by the horizontal sweep line when it passes through an event, is $\frac{1}{4}\sqrt{N}$, $\frac{1}{8}N$ and $\frac{1}{4.8}N$, respectively. The average is taken over all sweeping events.

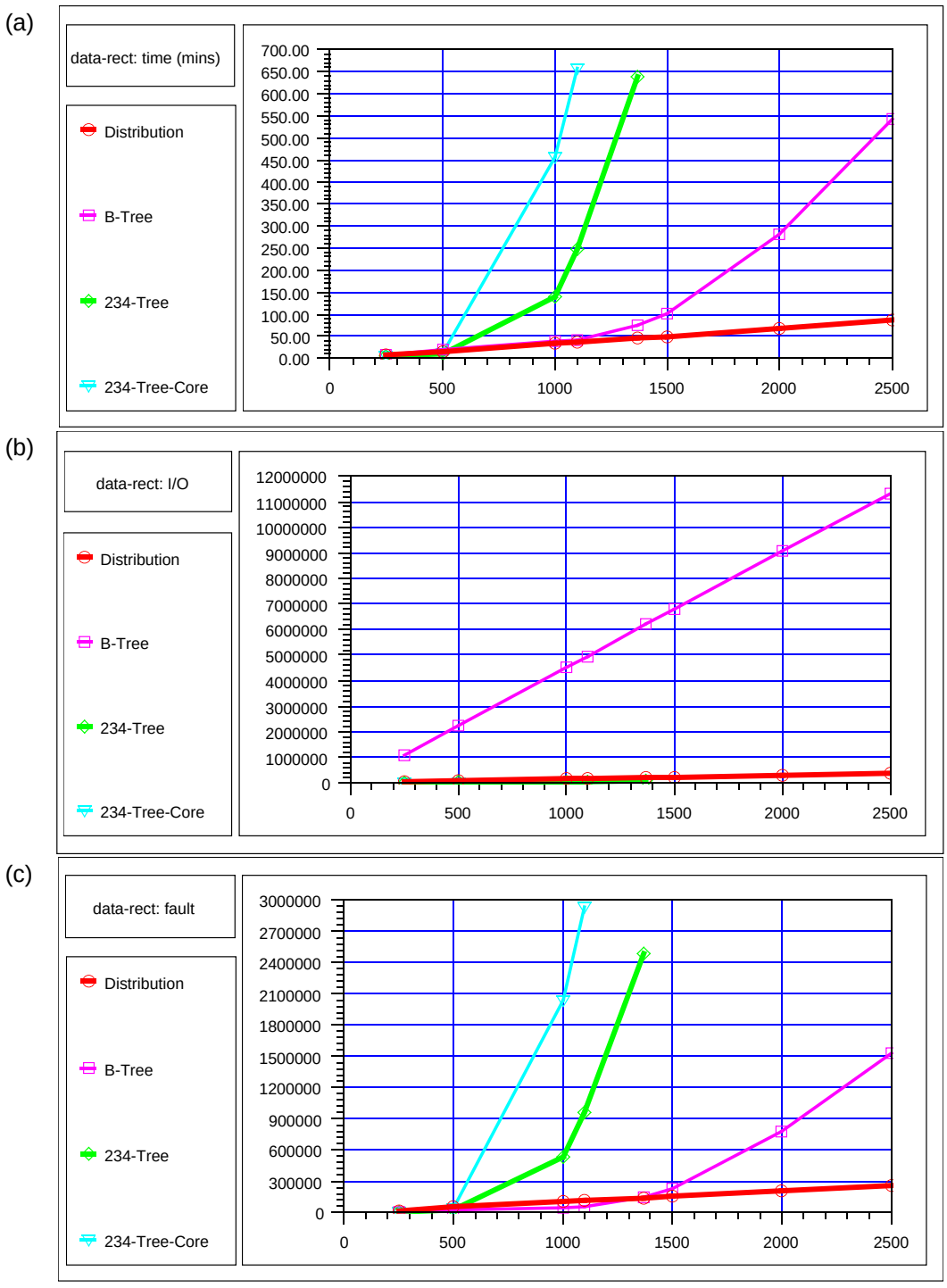

Figure 13: Experimental results for the algorithms running on the data set with an average number of vertical overlaps of $\frac{1}{4.8}N$. (a) running time in minutes, (b) number of I/O operations, (c) number of page faults.

Chiang experimented on a Sun Sparc-10 workstation with a main memory size of 32Mb and with a page size of 4Kb. The performance measures used was total running time (wall not cpu), number of I/O operations performed (i.e. number of blocks read and written by the program), and the number of page faults occurred (I/Os controlled by the operating system)—see [33] for a precise description of the experimental setting. The first surprising result of the experiments was that the main memory available for use is typically much smaller than what would be expected. The algorithms were implemented such that the amount of main memory used could be parameterized, and `Distribution` was run on a fixed data set with various sizes of main memory. In theory one would expect that the performance would increase with main memory size up close to the actual 32Mb of main memory, but it turned out that 4Mb gave the best performance—refer to Figure 12 where the number of I/Os and page faults is plotted as a function of the memory used. Going from 1Mb to 4Mb the same number of page faults occur and the number of I/Os decrease slightly, so that the actual running time is decreasing slightly. Going from 4Mb to 20Mb the number of I/Os again decreases only slightly while the number of page faults increase significantly, thus resulting in a much worse overall performance. The reason is that a lot of daemon programs are also taking up memory in the machine. Chiang thereafter performed all experiments with the parameter of the main memory size set to 4Mb

Experiments were performed with the four algorithms on data sets of sizes ranging from 250 thousand segments to 2.5 million segments. The overall conclusion made by Chiang is that while the performance of the tree variations of plane-sweep algorithms depends heavily on the average number of vertical overlaps, the performance of distribution sweeping is both steady and efficient. As could be expected `234-Tree-Core` performs the best for very small inputs in all experiments, but as input size grows the performance quickly becomes considerably worse than that of the other algorithms. Excluding `234-Tree-Core`, `234-Tree` always runs the fastest and `Distribution` always the slowest for the data set with a small average number of vertical overlaps ($\frac{1}{4}\sqrt{N}$)—because the search tree structure is small enough to fit into internal memory. Another reason is that `Distribution` sorts the data twice in the preprocessing step, while `234-Tree` only sorts ones. However, for the data set with a large average number of vertical overlaps ($\frac{1}{4.8}N$) `Distribution` runs much faster than the other algorithms for just moderately large data sets. For example, for $N = 1.37 \cdot 10^6$ `Distribution` runs for 45.29 minutes, `B-Tree` for 74.54 minutes, but `234-Tree` for more than 10.5 hours. Also, for $N = 2.5 \cdot 10^6$ `Distribution` runs for less than 1.5 hours, while `B-Tree` runs for more than 8.5 hours. The full result of Chiang's experiments with this data set is shown in Figure 13. Note that `Distribution` always performs less I/Os than `B-Tree`. Recall that the I/O cost of the two algorithms is $O(n \log_m n + t)$ and $O(N \log_B n + t)$, respectively. With the parameter of the main memory size set to 4Mb the two logarithmic terms in these bounds are almost the same, so it is the $1/B$ term that makes the difference significant.

From the experiments performed by Chiang one may conclude that explicitly considering the I/O cost of solving a problem can be very important, and that algorithms developed for the theoretical parallel disk model can very well have a good practical performance and can lead to the solution of problems one would not be able to solve in practice with algorithms developed for main memory. Chiang did not perform experiments with the buffer tree solution described in the last subsection. Even though the constants in the I/O bounds of the buffer tree operations are small, the buffer emptying algorithm for buffers containing rangesearch elements is quite complicated, so a worse performance could be expected of the buffer tree algorithm compared to the distribution sweeping algorithm. On the other hand the buffer tree algorithm does not need to sort the input twice as the distribution sweeping algorithm does. We plan to perform experiments with the buffer tree solution in the future. Finally, it should be mentioned that Vengroff and Vitter [92] and Arge et al. [18] have also performed some experiments with I/O algorithms. We will return to these experiments in Section 5 when we discuss the TPIE environment developed by Vengroff [89].

4.2 The Batched Range Searching Problem

In this section we consider another computational geometry problem with applications in GIS which is normally solved using plane-sweep; the batched range searching problem. Given N points and N (axis-parallel) rectangles in the plane (Figure 14) the problem consists of reporting for each rectangle all points that lie inside it.

The optimal internal-memory plane-sweep algorithm for the problem uses a data structure called a segment tree [23, 77]. The segment tree is a well-known dynamic data structure used to store a set of N segments in one dimension, such that given a query point all segments containing the point can be found in $O(\log_2 N + T)$ time. Such queries are normally called *stabbing queries*—refer to Figure 15. Using a segment tree the algorithm works as follows: A vertical sweep with a horizontal line is made. When the top horizontal segment of a rectangle is reached it is inserted in a segment tree. The segment is deleted again when the corresponding bottom horizontal segment is reached. When a point is reached in the sweep, a stabbing query is performed with it on the segment tree and in this way all rectangles containing the point are found. As insertions and deletions can be performed in $O(\log_2 N)$ time on a segment tree the algorithm runs in the optimal $O(N \log_2 N + T)$ time.

In external memory the batched range searching problem can be solved optimally using distribution sweeping [53] or an external buffered version of the segment tree [12]. As we will also use the external segment tree structure to solve the red/blue line segment intersection problem in Section 4.3, we will sketch the structure below.

4.2.1 The External Segment Tree

In internal memory a static segment tree consists of a binary base tree storing the endpoints of the segments, and a given segment is stored in up to two nodes on each level of the tree. More precisely a segment is stored in all nodes v where it contains the interval consisting of all endpoints below v, but not the interval associated with $parent(v)$. The segments stored in a node are just stored in an unordered list. A stabbing query can be answered efficiently on such a structure, simply by searching down the base tree for the query value, reporting all segments stored in the nodes encountered.

When we want to "externalize" the segment tree and obtain a structure with height $O(\log_m n)$, we need to increase the fan-out of the nodes in the base tree. This creates a number of problems when we want to store segments space-efficiently in secondary structures such that queries can be answered efficiently. Therefore we make the nodes have fan-out $\Theta(\sqrt{m})$ instead of the normal $\Theta(m)$. As discussed in Section 3.2 this smaller branching factor at most doubles the height of the tree, but as we will see it allows us to efficiently store segments in a number of secondary structures of each node.

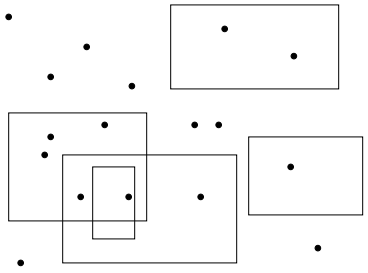

Figure 14: The batched range searching problem.

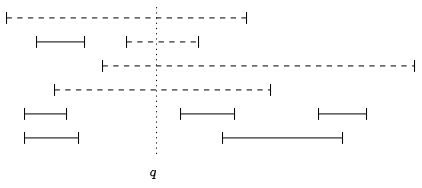

Figure 15: Stabbing query with q. Dotted segments are reported.

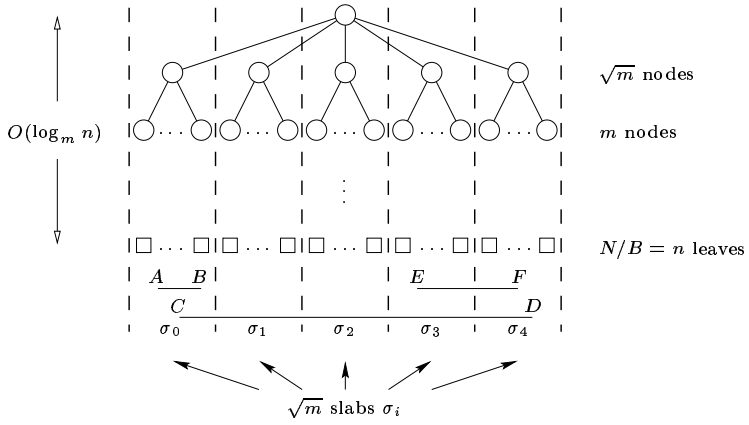

Figure 16: An external-memory segment tree over a set of N segments, three of which, $\overline{AB}$, $\overline{CD}$, and $\overline{EF}$, are shown.

The external segment tree [12] is sketched in Figure 16. The base structure is a perfectly balanced tree with branching factor $\sqrt{m}$ over the endpoints. A buffer of size $m/2$ blocks and $m/2 - \sqrt{m}/2$ lists of segments are associated with each node in the tree. A list (block) of segments is also associated with each leaf. A set of segments is stored in this structure as follows: The first level of the tree partitions the data into $\sqrt{m}$ intervals σ_i—for illustrative reasons we call them *slabs*—separated by dotted lines in Figure 16. *Multislabs* are then defined as contiguous ranges of slabs, such as for example $[\sigma_1, \sigma_4]$. There are $m/2 - \sqrt{m}/2$ multislabs and the lists associated with a node are precisely a list for each multislab. The key point is that the number of multislabs is a quadratic function of the branching factor. Thus by choosing the branching factor to be $\Theta(\sqrt{m})$ rather than $\Theta(m)$ we have room in internal memory for a constant number of blocks for each of the $\Theta(m)$ multislabs. Segments such as $\overline{CD}$ in Figure 16 that spans at least one slab completely are called *long segments*. A copy of each long segment is stored in the list of the largest multislab it spans. Thus, $\overline{CD}$ is stored in the list associated with the multislab $[\sigma_1, \sigma_3]$. All segments that are not long are called *short segments*. They are not stored in any multislab, but are passed down to lower levels of the tree where they may span recursively defined slabs and be stored. $\overline{AB}$ and $\overline{EF}$ are examples of short segments. Additionally, the portions of long segments that do not completely span slabs are treated as small segments. There are at most two such synthetically generated short segments for each long segment. Segments passed down to a leaf are just stored in one list. Note that we at most store one block of segments in each leaf. A segment is thus at most stored in two lists on each level of the tree and hence total space utilization is $O(n \log_m n)$ blocks.

Given an external segment tree *with empty buffers* a stabbing query can in analogy with the internal case be answered by searching down the tree for the query value, and at every node encountered report all the long segments associated with each of the multislabs that spans the query value. However, because of the size of the nodes and the auxiliary multislab data, the external segment tree is inefficient for answering single queries. But by using the general idea from Section 3.3 and 4.1 and make updates and queries buffered, we can perform the whole batch of operations needed to solve the batched range searching problem in the optimal $O(n \log_m n + t)$ I/Os. When we want to perform an update or a query we thus just keep an element for the operation in internal memory, and when we have collected a block of such operations we insert it in the buffer of the root. If this buffer now contains more than $m/2$ blocks we perform a buffer-emptying process on it as follows: First we load the elements in the buffer into internal memory. Then we in internal memory collect all the segment that need to be stored in the node and distribute a copy of them to the relevant multislab lists. After that we report the relevant stabbings, by for every multislab list

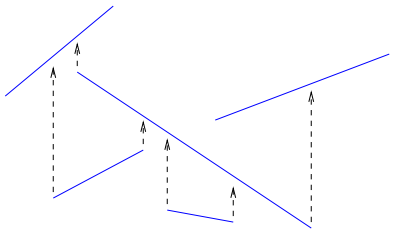

Figure 17: The endpoint dominance problem.

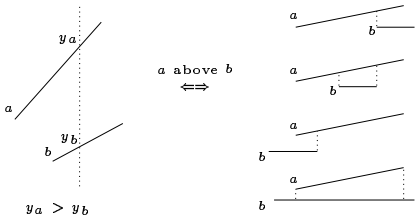

Figure 18: Comparing segments. Two segments can be related in four different ways.

in turn decide if the segments in the list are stabbed by any of the query points from the buffer. Finally, we distribute the segments and queries to the buffers of the nodes on the next level of the tree. As previously we obtain the $O(\frac{\log_m n}{B})$ and $O(\frac{\log_m n}{B} + t)$ amortized I/O bounds for the update and query operation, respectively, if the buffer-emptying process can be performed in $O(m + t')$ I/Os. That this is indeed the case can easily be realized by observing that we use $O(m)$ I/Os to load the element in the buffer and to distribute them back to the buffers one level down, and that we only use $O(m)$ I/Os extra to manage the multislab lists—basically because the number of such lists is $\Theta(m)$. Note that like the range searching operation on the buffer tree, the stabbing queries become batched. Thus, as discussed previously, the external segment tree can in general only be used to solve batched dynamic problems.

As mentioned in the beginning of this section distribution sweeping can also be used to solve the batched range searching problem. Also having obtained an algorithm for this problem, as well as for the orthogonal line segment intersection problem, one can also obtain an optimal external algorithm for the pairwise rectangle intersection problem—the problem of given N rectangles in the plane (with sides parallel to the axes) to report all intersecting pairs of rectangles [24]. Also the external segment tree approach for solving the batched range searching problem can be extended from 2 to d dimensions. A couple of new ideas is needed and the resulting algorithm uses $O(n \log^{d-1} n + t)$ I/Os [18]. The same technique works for a general class of problems called *colorable eternal-decomposable problems*.

4.3 The Red/Blue Line Segment Intersection Problem

After having presented the basic paradigms for designing I/O-efficient computational geometry algorithms through the development of optimal algorithms for the relatively simple problems of orthogonal line segment intersection and batched range searching, we now turn to the more complicated red/blue line segment intersection reporting problem: Given two internally non-intersecting sets of line segments, the problem is to report all intersections between segments in the two sets.

As previously discuss the red/blue line segment intersection problem is at the core of the important GIS problem of map overlaying. Unfortunately, it turns out that distribution sweeping and buffer trees are inadequate for solving the problem, as well as other problems involving line segments which are not axis-parallel. In the next subsection we try to illustrate this before we in subsections 4.3.2 and 4.3.3 sketch how to actually solve the problem in the optimal number of I/Os.

4.3.1 The Endpoint Dominance Problem

Let us consider the endpoint dominance (EPD) problem defined as follows [19]: Given N non-intersecting line segments in the plane, find the segment directly above each endpoint of each segment—refer to Figure 17.

Even though EPD seems to be a rather simple problem, it is a powerful tool for solving other important problems. As an example EPD can be used to sort non-intersecting segments in the plane, an important subproblem in the algorithm for the red/blue line segment intersection problem. A segment $\overline{AB}$ in the plane is *above* another segment $\overline{CD}$ if we can intersect both $\overline{AB}$ and $\overline{CD}$ with the same vertical line l, such that the intersection between l and $\overline{AB}$ is above the intersection between l and $\overline{CD}$. Two segments are incomparable if they cannot be intersected with the same vertical line. The problem of sorting N non-intersecting segments is to extend the partial order defined in this way to a total order.

Figure 18 demonstrates that if two segments are comparable then it is sufficient to consider vertical lines through the four endpoints to obtain their relation. Thus one way to sort N segments [19] is to add two "extreme" segments as indicated in Figure 19, and use EPD twice to find for each endpoint the segments immediately above and below it. Using this information we create a (planar s,t-) graph where nodes correspond to segments and where the relations between the segments define the edges. Then the sorted order can be obtained by topologically sorting this graph in $O(n \log_m n)$ I/Os using an algorithm developed in [34]. This means that if EPD can be solved in $O(n \log_m n)$ I/Os then N segments can be sorted in the same number of I/Os.

In internal memory EPD can be solved optimally with a simple plane-sweep algorithm. We sweep the plane from left to right with a vertical line, inserting a segment in a search tree when its left endpoint is reached and removing it again when the right endpoint is reached. For every endpoint we encounter, we also perform a search in the tree to identify the segment immediately above the point (refer to Figure 17). One might think that it is equally easy to solve EPD in external memory, using distribution sweeping or buffer trees. Unfortunately, this is not the case.

One important property of the internal-memory plane-sweep algorithm for EPD is that only segments that actually cross the sweep-line are stored in the search tree at any given time during the sweep. This means that all segments in the tree are comparable and that we can easily compute their order. However, if we try to store the segments in a buffer tree during the sweep, the tree can (because of the "laziness" in the structure) also contain "old" segments which do not cross the sweep-line. This means that we can end up in a situation where we try to compare two incomparable segments. In general the buffer tree only works if we know a total order on the elements inserted in it or if we can compare all pair of elements. Thus we cannot directly use the buffer tree in the plane-sweep algorithm. We could try to compute a total order on the segments before solving EPD, but as discussed above the solution to EPD is one of the major steps towards finding such an order so this seems infeasible.

For similar reasons using distribution sweeping seems infeasible as well. Recall that in distribution sweeping we need to perform one sweep in a linear number of I/Os to obtain an efficient solution. Normally this is accomplished by sorting the objects by y-coordinate in a preprocessing phase. This e.g. allows one to sweep over the objects in y order without sorting on each level of recursion, because as the objects are distributed to recursive subproblems their y ordering is retained. In the orthogonal line segment intersection case we presorted the segments by endpoint in order to sweep across them in endpoint y order. In order to use distribution sweeping to solve

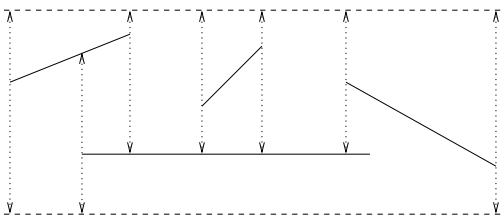

Figure 19: Algorithm for the segment sorting problem.

EPD it seems that we need to presort the segments and not the endpoints.

4.3.2 External-Memory Fractional Cascading

As attempts to solve EPD optimally using the buffer tree or distribution sweeping fail we are led to other approaches. It is possible to come close to solving EPD by first constructing an external-memory segment tree over the projections of the segments onto the x-axis and then performing stabbing queries at the x-coordinates of the endpoints of the segments. However, what we want is the single segment directly above each query point in the y dimension, as opposed to all segments it stabs. This segment could be found if we were able to compute the segment directly above a query point among the segments stored in a given node of the external segment tree. We call such a segment a *dominating segment*. Then we could examine each node on the path from the root to the leaf containing the query point, and in each such node find the dominating segment and compare it to the segment found to be closest to the query so far. When the leaf is reached we would then know the "global" dominating segment.

However, there are a number of problems that have to be dealt with in order to find the dominating segment of a query point among the segments stored in a node. The main problems are that the dominating segment could be stored in a number of multislab lists, namely in all lists containing segments that contain the query point, and that a lot of segments can be stored in a multislab list. Both of these facts seem to suggest that we need a lot of I/Os to find the dominating segment. However, as we are looking for an $O(n \log_m n)$ solution, and as the segment tree has $O(\log_m n)$ levels, we are only allowed to use a linear number of I/Os to find the positions of *all* the N query points among the segments stored in one level of the tree. This gives us less than one I/O per query point per node!

Fortunately, it is possible to modify the external segment tree and the query algorithm to overcome these difficulties [19]. To do so we first strengthen the definition of the external segment tree and require that the segments in the multislab lists are sorted. Note that all pairs of segments in the same multislab list can be compared just by comparing the order of their endpoints on one of the boundaries of the multislab, and that a multislab list thus can be sorted using a standard sorting algorithm. In [19] it is shown how to build an external segment tree with sorted multislab lists on N non-intersecting segments in $O(n \log_m n)$ I/Os. The construction is basically done using distribution sweeping.

The sorting of the multislab lists makes it easier to search for the dominating segment in a given multislab list but it may still require a lot of I/Os. We also needs to be able to look for the dominating segment in many of the multislabs lists. However, one can overcome these problems using *batched filtering* [53] and a technique similar to what in internal memory is called *fractional cascading* [30, 31, 86]. The idea in batched filtering is to process all the queries at the same time and level by level, such that the dominating segments in nodes on one level of the structure are found for all the queries, before continuing to consider nodes on the next level. In internal memory the idea in fractional cascading is that instead of e.g. searching for the same element individually in S sorted lists containing N elements each, each of the lists are in a preprocessing step augmented with sample elements from the other lists in a controlled way, and with "bridges" between different occurrences of the same element in different lists. These bridges obviate the need for full searches in each of the lists. To perform a search one only searches in one of the lists and uses the bridges to find the correct position in the other lists. This results in a $O(\log_2 N + S)$ time algorithm instead of an $O(S \log_2 N)$ time algorithm.

In the implementation of what could be called *external fractional cascading*, we do not explicitly build bridges but we still use the idea of augmenting some lists with elements from other lists. The construction is rather technical, but the general idea is the following (the interested reader is referred to [19] for details): First a preprocessing step is used (like in fractional cascading) to sample a set

of segments from each slab in each node and the multislab lists of the corresponding child are augmented with these segments. The sampling is done in $O(n \log_m n)$ I/Os using the distribution paradigm. Having preprocessed the structure the N queries are filtered through it. In order to do so in the optimal number of I/Os the filtering is done in a rather untraditional way—from the leaves towards the root. First the query points are sorted and distributed to the leaves to which they belong. Then for each leaf in turn the dominating segment among the segments stored in the leaf is found for all query points assigned to the leaf. This can be done efficiently using an internal-memory algorithm, because the segments stored in a leaf easily fit in internal memory. This is also the reason for the untraditional search direction—one cannot in the same way efficiently find the dominating segments among the segments stored in the root of the tree, because more than a memory load of segments can be stored there. Next the actual filtering up the $O(\log_m n)$ levels is performed, and on each level the dominating segment is found for all the query points. This is done I/O efficiently using the merging paradigm and the sampled segments from the preprocessing phase. In [19] it is shown that one filtering step can be performed in $O(n)$ I/Os, and thus EPD can be solved in $O(n \log_m n)$ I/O operations.

4.3.3 External Red/Blue Line Segment Intersection Algorithm

Using the solution to the EPD problem, or rather the ability to sort non-intersecting segments, we can now solve the red/blue line segment intersection problem with (a variant of) distribution sweeping. Recall that we in the solution to the orthogonal line segment intersection problem presorted the endpoints of the segments by y-coordinate, and used the sorted sequence throughout the algorithm to perform vertical sweeps. The key to solving the red/blue problem is to presort the red and the blue *segments* (not endpoints) individually, and perform the sweep in segment order rather than in y order of the endpoints. Thus given input sets S_r of non-intersecting red segments and S_b of non-intersecting blue segments, we construct two intermediate sets

$$\begin{aligned} T_r &= S_r \cup \bigcup_{(p,q) \in S_b} \{(p,p),(q,q)\} \\ T_b &= S_b \cup \bigcup_{(p,q) \in S_r} \{(p,p),(q,q)\} \end{aligned}$$

Each new set is the union of the input segments of one color and the endpoints of the segments of the other color (or rather zero length segments located at the endpoints). Both T_r and T_b are of size $O(|S_r| + |S_b|) = O(N)$ and can thus be sorted in $O(n \log_m n)$ I/Os.

We now locate intersections with distribution sweeping with a branching factor of $\sqrt{m}$. Recall that the structure of distribution sweeping is that we divide the plane into $\sqrt{m}$ slabs, and that we then find intersections involving parts of segments that completely span one or more slabs, before we solve the problem recursively in each slab. The recursion continues through $O(\log_m n)$ levels until the subproblems are small enough to be solved in internal memory. If we can do a sweep in $O(n)$ I/Os plus the number of I/O's used to report intersections, we obtain the optimal $O(n \log_m n + t)$ I/O solution.

So let us consider the sweep algorithm. In one sweep we define *long segments* as those crossing one or more slabs and *short segments* as those completely contained in a slab. Furthermore, we shorten the long segments by "cutting" them at the right boundary of the slab that contains their left endpoint, and at the left boundary of the slab containing their right endpoint. Thus our task in a sweep is to report all intersections between long segments of one color and long and short segments of the other color—refer to Figure 20. To find the intersections between long and short segments we use the sweep algorithm used in Section 4.1 to solve the orthogonal line segment intersection problem—except that we sweep in the order of T_r and T_b. We use the algorithm twice,

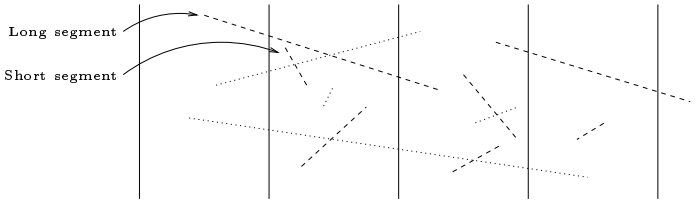

Figure 20: Long and short segments (red segments dotted, blue segments dashed).

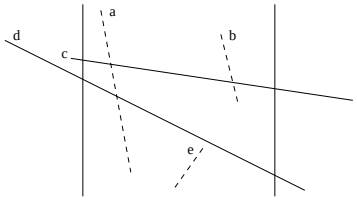

Figure 21: Why sweeping in segment-order rater than y-order is necessary.

treating long segments of one color as horizontal segments and short segments of the other color as vertical segments. For long red and short blue segments we proceed as follows: We sweep from top to bottom by scanning through the sorted list T_r of red segments and blue endpoints. When a top endpoint of a small blue segment is encountered we insert the segment in the active list associated with the slab containing the segment. When a long red segment is encountered we then scan through all the active lists associated with the slabs it completely spans. During this scan we know that every small blue segment is either intersected by the red segment, or will not be intersected by any of the following red segments and can therefore be removed from the list. As previously we use $O(n + t')$ I/Os to do the sweep.

Note that whereas the important property that a small segment not intersected by a long segment is not intersected by any of the following long segments in the orthogonal case followed from the fact that we were working on the segments in y order, it now follows from the fact that we are sweeping the segments and endpoints in sorted order. As an illustration of this refer to Figure 21. In the sweep we will meet the segments in order a, b, c, d, e, whereas we would meet them in order a, b, d, c, e if we were sweeping in endpoint y order. In the latter case the important property would not hold as segment b actually intersect segment c, even though it does not intersect d which is encountered after b but before c.

In order to report intersections between long segments of different colors the notion of multislab (as in Section 4.2) is used. First we scan through T_r and distribute the long red segments to the $O(m)$ multislabs. Next, we scan through the blue set T_b, and for each long blue segment we report intersections with the relevant long red segments. This is the same as reporting intersections with the appropriate red segments in each of the multislab lists. As each of the multislabs lists are sorted, and as we also process the blue segments in sorted order, it turns out that this can be done in a simple and efficient way using a merging idea, where it again is crucial that the number of multislab lists is $O(m)$ (that the distribution factor is $\sqrt{m}$). Details in the algorithm appear in [19], where it is also proved that the sweep can be performed in $O(n + t')$ I/Os as required.

To summarize, the red/blue line segment intersection problem can be solved in the optimal $O(n \log_m n + t)$ I/Os. However, as the segment sorting algorithm used in the solution is relatively complicated, its practical importance may be limited. It would be interesting to experimentally compare the algorithms performance to that of other (internal-memory) algorithms for the problem [28, 29, 30, 68, 75]. An experimental comparison of internal-memory algorithms for the problem is already reported in [10].

4.4 Other External-Memory Computational Geometry Algorithms

In the previous sections we have discussed the basic techniques for designing efficient external-memory computational geometry algorithms. We have illustrated the powerful distribution sweeping technique using the orthogonal line segment intersection problem. We have also already men-

tioned that the technique can be used to solve the batched range searching problem. In [53] it is discussed how it can be used to develop optimal algorithms for a number of other important problems, including for the problems of finding the pairwise intersection of N rectangles, finding all nearest neighbors for a set of N points in the plane, computing the measure (area) of the union of N rectangles in the plane, and for several geometric dominance problems. Several of these problems have applications in GIS systems. In [18] some of the algorithms are extended to work in d dimensions.

Goodrich et al. [53] also discussed external-memory algorithms for the convex hull problem, that is, the problem of computing the smallest convex polytope completely enclosing a set of N points in d-dimensional space. In the two-dimensional case the internal-memory algorithm known as Graham's scan [54, 77] can be modified in a simple way to obtain an $O(n \log_m n)$ I/O external algorithm. They also discussed how to obtain an output-sensitive algorithm based upon an external version of the marriage-before-conquest technique [62]. The algorithm uses $O(n \log_m t)$ I/Os. Finally, they developed $O(n \log_m n)$ algorithms for the three-dimensional case which is particularly interesting because of the close relation to the two-dimensional versions of the problems of computing the Voronoi diagram and the Delaunay triangulation of a set of N points. Using the reduction described in [55] the 3-d convex hull algorithm immediately gives algorithms for the two latter problems with the same I/O performance.

The $O(n \log_m n)$ solution to the EPD problem discussed in the last section, which lead to the segment sorting and the red/blue line segment intersection algorithms, has several other almost immediate consequences [19]. If one takes a closer look at the algorithm for EPD one realizes that it works in general with K query points, which are not necessarily endpoints of the segments. Therefore the result leads to an $O((n+k) \log_m n)$ I/O solution to the batched planar point location problem, that is, the problem in which one are given a planar decomposition by N line segments and wants for each of K query points to locate the region in which it lies. Similarly the EPD algorithm leads to algorithms for a couple of region decomposition problems. First it leads to an algorithm for trapezoid decomposition of a set of N segments [69, 77], as the core of this problem precisely is to find for each segment endpoint the segment immediately above it. Using a slightly modified version of an internal-memory algorithm [48], the ability to compute a trapezoid decomposition of a simple polygon then leads to an $O(n \log_m n)$ polygon triangulation algorithm. Finally, using a complicated integration of all the ideas in the red/blue line segment intersection algorithm with the external priority queue discussed in Section 3.3 [12], one can obtain an $O((n+t) \log_m n)$ I/O algorithm for the general line segment intersection problem, where one is just given N segments in the plane and should report their pairwise intersections.

4.5 Summary

- Main paradigms for developing external computational geometry algorithms:
  - Distribution sweeping.
  - Batched dynamic data structures (external buffered one-dimensional range tree and segment tree).
  - Batched filtering.
  - External fractional cascading.
- Optimal algorithms developed for a large number of problems.
- In practice:
  - Experiments with orthogonal line segment intersection algorithms suggest that algorithms developed for the parallel disk model perform well in practice.
  - Much larger problems instances seem to be solvable with I/O algorithms than with main memory algorithms.

5 TPIE — A Transparent Parallel I/O Environment

In Section 4.1 we discussed the experiments with orthogonal line segment intersection algorithms carried out by Chiang [32, 33]. As discussed these experiments suggest that algorithms developed for the parallel disk model perform well in practice, and that they can very well lead to the solution of problem instances one would not be able to solve in practice with algorithms developed for main memory. Unfortunately, existing systems tend not to adequately support the functionality required in order to implement algorithms designed for the parallel disk model directly. Most operating systems basically lets the programmer program a virtual machine with (practically) unlimited main memory, and control I/O "behind the back" of the programmer. However, in order to implement algorithms developed for the parallel disk model one needs to be able to explicitly control I/O, and thus it seems that one has to try to bypass the operating system and write very low level code in order to implement the algorithms we have discussed. Doing so would be a very complicated task and would probably lead to very inflexible code, which would not be portable across different platforms.

On the other hand we have seen how a large number of problems can be solved using a relatively small number of paradigms, such as merging, distribution (and distribution sweeping), and buffered external data structures. The Transparent Parallel I/O Environment (TPIE) proposed by Vengroff [89, 91, 94] tries to take advantage of this. While Chiang [32, 33] performed experiments in order to compare the efficiency of algorithms designed for internal and external memory and to validate the I/O-model, TPIE is designed to assist programmers in the development of I/O-efficient (and easily portable) programs. TPIE implements a set of high-level paradigms (access methods) which lets the programmers specify the functional details of the computation they wish to perform within a given paradigm, without explicitly having to worry about doing I/O or managing internal memory. The paradigms supported by the current prototype of TPIE includes scanning, distribution, merging, sorting, permuting, and matrix arithmetic [91, 94].

In order to allow programmers to abstract away I/O, TPIE uses a stream approach. A computation is viewed as a continuous process in which a program is fed streams of data from an outside source and leave trails (in form of other streams of data) behind it. In this way programmers

only need to specify the functional details in the computation they wish to perform within a given paradigm. TPIE then choreograph an appropriate sequence of I/Os in order to keep the computation fed. To realize that the stream approach is indeed natural, just consider a simple version of merge sort. Here a stream of data is first read and divided into a number of (sorted) main memory sized streams, which are then continually read m at a time and merged into a longer stream. Having implemented basic stream handling routines, the programmers only need to specify how to compare objects in order to sort a given set of objects using the external merge sort paradigm—without having to worry about I/O. Note that the programmers do not even need to worry if the streams are stored on a single disk or if a number of parallel disks are used.

TPIE is implemented in C++ as a set of template classes and functions and a run-time library. The current implementation supports access to data stored on one or more disks attached to a workstation. In the future, it is the plan that TPIE will support multiprocessors and/or collections of workstations. TPIE is a modular system with three components; a block transfer engine (BTE), a memory manager (MM) and an access method interface (AMI). The BTE is responsible for moving blocks of data to and from disk, that is, it is intended to bridge the gap between the I/O hardware and the rest of the system. If the system consists of several processors, every processor has its own BTE. The MM running on top of one or more BTEs is responsible for managing main memory resources. All memory allocated by application programs or other parts of TPIE is handled by the MM. Finally, the AMI provides the high-level interface to the programmer and is the only component with which most programmers will need to interact directly. As mentioned the access methods supported by the AMI currently include scanning, distribution, merging, sorting, permuting, and matrix arithmetic. The interested reader is refereed to [89, 91, 94] for details. In [91] implementations of algorithms such as convex hull and list ranking are also discussed. Finally, it is discussed how to obtain the prototype version of TPIE. Currently, TPIE does not support external buffered data structures but we hope in the future to include such structures in the environment. Similarly, it is the plan to extend TPIE with support for more application controlled I/O in order to allow implementation of the in GIS commonly used indexing structures.

In [92] Vengroff and Vitter discuss applications of TPIE to problems in scientific computing, and report some performance results of programs written to solve certain benchmark problems. The TPIE paradigms used in these experiments are scanning, sorting, and matrix arithmetic. The main conclusions made are that TPIE is indeed practical and efficient, and that algorithms for the theoretical parallel disk model perform well in practice. Actually, Vengroff and Vitter show that using TPIE results in a small CPU overhead compared to entirely main memory implementation, but allows much larger data sets to be used. Also, for the implemented benchmarks, the time spent on I/O range from being negligible to being of the same order of magnitude as internal computation time, showing that using TPIE a large degree of overlap between computation and I/O can be accomplished. Very recently, Arge et al. [18] reported similar encouraging experiences with a TPIE implementation of an algorithms for the pairwise rectangle intersection problem.

6 Conclusions

As GIS systems frequently handle huge amounts of data it is getting increasingly important to design algorithms with good I/O performance for problems arising in such systems. Many important computational geometry problems are abstractions of important GIS operations, and in recent years a number of basic techniques for designing I/O-efficient algorithms for such problems have been developed. In this note we have surveyed these techniques and the algorithms developed using them. However, the young field of I/O-efficient computation is to a large extend still wide open. Even though the experimental results reported so far are encouraging, a major future goal is to investigate the practical merits of the developed I/O algorithms.

Acknowledgments

I would like to thank Yi-Jen Chiang and Pavan Kumar Desikan for reading earlier drafts of this note and Yi-Jen Chiang for providing Figure 12 and 13.

References

[1] P. K. Agarwal, L. Arge, T. M. Murali, K. Varadarajan, and J. S. Vitter. I/O-efficient algorithms for contour line extraction and planar graph blocking. In *Proc. ACM-SIAM Symp. on Discrete Algorithms (to appear)*, 1998.

[2] A. Aggarwal, B. Alpern, A. K. Chandra, and M. Snir. A model for hierarchical memory. In *Proc. ACM Symp. on Theory of Computation*, pages 305–314, 1987.

[3] A. Aggarwal and A. K. Chandra. Virtual memory algorithms. In *Proc. ACM Symp. on Theory of Computation*, pages 173–185, 1988.

[4] A. Aggarwal, A. K. Chandra, and M. Snir. Hierarchical memory with block transfer. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, pages 204–216, 1987.

[5] A. Aggarwal and G. Plaxton. Optimal parallel sorting in multi-level storage. *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 659–668, 1994.

[6] A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[7] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[8] B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12(2-3), 1994.

[9] R. J. Anderson and G. L. Miller. A simple randomized parallel algorithm for list-ranking. *Information Processing Letters*, 33:269–273, 1990.

[10] D. S. Andrews, J. Snoeyink, J. Boritz, T. Chan, G. Denham, J. Harrison, and C. Zhu. Further comparisons of algorithms for geometric intersection problems. In *Proc. 6th Int'l. Symp. on Spatial Data Handling*, 1994.

[11] ARC/INFO. *Understanding GIS—the ARC/INFO method*. ARC/INFO, 1993. Rev. 6 for workstations.

[12] L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Proc. Workshop on Algorithms and Data Structures, LNCS 955*, pages 334–345, 1995. A complete version appears as BRICS technical report RS-96-28, University of Aarhus.

[13] L. Arge. The I/O-complexity of ordered binary-decision diagram manipulation. In *Proc. Int. Symp. on Algorithms and Computation, LNCS 1004*, pages 82–91, 1995. A complete version appears as BRICS technical report RS-96-29, University of Aarhus.

[14] L. Arge. *Efficient External-Memory Data Structures and Applications*. PhD thesis, University of Aarhus, February/August 1996.

[15] L. Arge, P. Ferragina, R. Grossi, and J. Vitter. On sorting strings in external memory. In *Proc. ACM Symp. on Theory of Computation*, pages 540–548, 1997.

[16] L. Arge, M. Knudsen, and K. Larsen. A general lower bound on the I/O-complexity of comparison-based algorithms. In *Proc. Workshop on Algorithms and Data Structures, LNCS 709*, pages 83–94, 1993.

[17] L. Arge and P. B. Miltersen. On showing lower bounds for external-memory computational geometry. In preparation.

[18] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Theory and practice of I/O-efficient algorithms for multidimensional batched searching problems. In *Proc. ACM-SIAM Symp. on Discrete Algorithms (to appear)*, 1998.

[19] L. Arge, D. E. Vengroff, and J. S. Vitter. External-memory algorithms for processing line segments in geographic information systems. *Algorithmica*, to appear. special issues on Geographical Information Systems.

[20] L. Arge and J. S. Vitter. Optimal dynamic interval management in external memory. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, pages 560–569, 1996.

[21] R. D. Barve, E. F. Grove, and J. S. Vitter. Simple randomized mergesort on parallel disks. In *Proc. ACM Symp. on Parallel Algorithms and Architectures*, 1996.

[22] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[23] J. L. Bentley. Algorithms for klee's rectangle problems. Dept. of Computer Science, Carnegie Mellon Univ., unpublished notes, 1977.

[24] J. L. Bentley and D. Wood. An optimal worst case algorithm for reporting intersections of rectangles. *IEEE Transactions on Computers*, 29:571–577, 1980.

[25] G. Blankenagel and R. H. Güting. XP-trees—External priority search trees. Technical report, FernUniversität Hagen, Informatik-Bericht Nr. 92, 1990.

[26] M. Blum, R. W. Floyd, V. Pratt, R. L. Rievest, and R. E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7:448–461, 1973.

[27] P. Callahan, M. T. Goodrich, and K. Ramaiyer. Topology B-trees and their applications. In *Proc. Workshop on Algorithms and Data Structures, LNCS 955*, pages 381–392, 1995.

[28] T. M. Chan. A simple trapezoid sweep algorithm for reporting red/blue segment intersections. In *Proc. of 6th Canadian Conference on Computational Geometry*, 1994.

[29] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. *Journal of the ACM*, 39:1–54, 1992.

[30] B. Chazelle, H. Edelsbrunner, L. J. Guibas, and M. Sharir. Algorithms for bichromatic line-segment problems and polyhedral terrains. *Algorithmica*, 11:116–132, 1994.

[31] B. Chazelle and L. J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1:133–162, 1986.

[32] Y.-J. Chiang. *Dynamic and I/O-Efficient Algorithms for Computational Geometry and Graph Problems: Theoretical and Experimental Results.* PhD thesis, Brown University, August 1995.

[33] Y.-J. Chiang. Experiments on the practical I/O efficiency of geometric algorithms: Distribution sweep vs. plane sweep. In *Proc. Workshop on Algorithms and Data Structures, LNCS 955*, pages 346–357, 1995.

[34] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 139–149, 1995.

[35] D. R. Clark and J. I. Munro. Efficient suffix trees on secondary storage. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 383–391, 1996.

[36] A. Cockcroft. *Sun Performance and Tuning. SPARC & Solaris.* Sun Microsystems Inc., 1995.

[37] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[38] T. H. Cormen. *Virtual Memory for Data Parallel Computing.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1992.

[39] T. H. Cormen. Fast permuting in disk arrays. *Journal of Parallel and Distributed Computing*, 17(1-2):41–57, 1993.

[40] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, Mass., 1990.

[41] T. H. Cormen and L. F. Wisniewski. Asymptotically tight bounds for performing BMMC permutations on parallel disk systems. In *Proc. ACM Symp. on Parallel Algorithms and Architectures*, pages 130–139, 1993.

[42] R. F. Cromp. An intellegent information fusion system for handling the archiving and querying of terabyte-sized spatial databases. In *S. R. Tate ed., Report on the Workshop on Data and Image Compression Needs and Uses in the Scientific Community, CESDIS Technical Report Series, TR–93–99*, pages 75–84, 1993.

[43] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry – Algorithms and Applications.* Springer Verlag, Berlin, 1997.

[44] H. Edelsbrunner and M. Overmars. Batched dynamic solutions to decomposable searching problems. *Journal of Algorithms*, 6:515–542, 1985.

[45] P. Ferragina and R. Grossi. A fully-dynamic data structure for external substring search. In *Proc. ACM Symp. on Theory of Computation*, pages 693–702, 1995.

[46] P. Ferragina and R. Grossi. Fast string searching in secondary storage: Theoretical developments and experimental results. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 373–382, 1996.

[47] R. W. Floyd. Permuting information in idealized two-level storage. In *Complexity of Computer Calculations*, pages 105–109, 1972. R. Miller and J. Thatcher, Eds. Plenum, New York.

[48] A. Fournier and D. Y. Montuno. Triangulating simple polygons and equivalent problems. *ACM Trans. on Graphics*, 3(2):153–174, 1984.

[49] P. G. Franciosa and M. Talamo. Orders, implicit k-sets representation and fast halfplane searching. In *Proc. Workshop on Orders, Algorithms and Applications (ORDAL'94)*, pages 117–127, 1994.

[50] G. R. Ganger, B. L. Worthington, R. Y. Hou, and Y. N. Patt. Disk arrays. High-performance, high-reliability storage subsystems. *IEEE Computer*, 27(3):30–46, 1994.

[51] D. Gifford and A. Spector. The TWA reservation system. *Communications of the ACM*, 27:650–665, 1984.

[52] M. T. Goodrich, M. H. Nodine, and J. S. Vitter. Blocking for external graph searching. *Algorithmica*, 16(2):181–214, 1996.

[53] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, pages 714–723, 1993.

[54] R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1:132–133, 1972.

[55] L. J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of voronoi diagrams. *ACM Trans. on Graphics*, 4:74–123, 1985.

[56] L. M. Haas and W. F. Cody. Exploiting extensible dbms in integrated geographic information systems. In *Proc. of Advances in Spatial Databases, LNCS 525*, 1991.

[57] J. W. Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proc. ACM Symp. on Theory of Computation*, pages 326–333, 1981.

[58] S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.

[59] C. Icking, R. Klein, and T. Ottmann. Priority search trees in secondary memory. In *Proc. Graph-Theoretic Concepts in Computer Science, LNCS 314*, pages 84–93, 1987.

[60] B. H. H. Juurlink and H. A. G. Wijshoff. The parallel hierarchical memory model. In *Proc. Scandinavian Workshop on Algorithms Theory, LNCS 824*, pages 240–251, 1993.

[61] P. C. Kanellakis, S. Ramaswamy, D. E. Vengroff, and J. S. Vitter. Indexing for data models with constraints and classes. *Journal of Computer and System Sciences*, 52(3), 1996.

[62] D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? *SIAM Journal of Computing*, 15:287–299, 1986.

[63] M. Knudsen and K. Larsen. I/O-complexity of comparison and permutation problems. Master's thesis, University of Aarhus, November 1992.

[64] M. Knudsen and K. Larsen. Simulating I/O-algorithms. Master student project, University of Aarhus, August 1993.

[65] D. Knuth. *The Art of Computer Programming, Vol. 3 Sorting and Searching.* Addison-Wesley, 1973.

[66] V. Kumar and E. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proc. IEEE Symp. on Parallel and Distributed Processing*, 1996.

[67] R. Laurini and A. D. Thompson. *Fundamentals of Spatial Information Systems.* A.P.I.C. Series, Academic Press, New York, NY, 1992.

[68] H. G. Mairson and J. Stolfi. Reporting and counting intersections between two sets of line segments. In *R. Earnshaw (ed.), Theoretical Foundation of Computer Graphics and CAD, NATO ASI Series, Vol. F40*, pages 307–326, 1988.

[69] K. Mulmuley. *Computational Geometry. An introduction through randomized algorithms.* Prentice-Hall, 1994.

[70] J. Nievergelt and P. Widmayer. Spatial data structures: Concepts and design choices. In M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, editors, *Algorithmic Foundations of GIS*. Springer-Verlag, 1997. Lecture Notes in Computer Science (subseries: tutorials).

[71] M. H. Nodine and J. S. Vitter. Deterministic distribution sort in shared and distributed memory multiprocessors. In *Proc. ACM Symp. on Parallel Algorithms and Architectures*, pages 120–129, 1993.

[72] M. H. Nodine and J. S. Vitter. Paradigms for optimal sorting with multiple disks. In *Proc. of the 26th Hawaii Int. Conf. on Systems Sciences*, 1993.

[73] M. H. Nodine and J. S. Vitter. Greed sort: An optimal sorting algorithm for multiple disks. *Journal of the ACM*, pages 919–933, 1995.

[74] M. Overmars, M. Smid, M. de Berg, and M. van Kreveld. Maintaining range trees in secundary memory. Part I: Partitions. *Acta Informatica*, 27:423–452, 1990.

[75] L. Palazzi and J. Snoeyink. Counting and reporting red/blue segment intersections. In *Proc. Workshop on Algorithms and Data Structures, LNCS 709*, pages 530–540, 1993.

[76] Y. N. Patt. The I/O subsystem—a candidate for improvement. *Guest Editor's Introduction in IEEE Computer*, 27(3):15–16, 1994.

[77] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, 1985.

[78] S. Ramaswamy and S. Subramanian. Path caching: A technique for optimal external searching. In *Proc. ACM Symp. Principles of Database Systems*, 1994.

[79] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, 1994.

[80] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS.* Addison Wesley, MA, 1989.

[81] J. E. Savage. Space-time tradeoffs in memory hierarchies. Technical Report CS-93-08, Brown University, 1993.

[82] M. Smid. *Dynamic Data Structures on Multiple Storage Media.* PhD thesis, University of Amsterdam, 1989.

[83] M. Smid and M. Overmars. Maintaining range trees in secundary memory. Part II: Lower bounds. *Acta Informatica*, 27:453–480, 1990.

[84] S. Subramanian and S. Ramaswamy. The p-range tree: A new data structure for range searching in secondary memory. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 378–387, 1995.

[85] R. E. Tarjan. Amortized computational complexity. *SIAM J. Alg. Disc. Meth.*, 6(2):306–318, 1985.

[86] V. K. Vaishnavi and D. Wood. Rectilinear line segment intersection, layered segment trees, and dynamization. *Journal of Algorithms*, 3:160–176, 1982.

[87] M. van Kreveld. Geographic information systems. Utrecht University, INF/DOC–95–01, 1995.

[88] J. van Leeuwen. *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity.* Elsevier, 1990.

[89] D. E. Vengroff. A transparent parallel I/O environment. In *Proc. 1994 DAGS Symposium on Parallel Computation*, 1994.

[90] D. E. Vengroff. Private communication, 1995.

[91] D. E. Vengroff. *TPIE User Manual and Reference.* Duke University, 1995. Available via WWW at `http://www.cs.duke.edu/TPIE`.

[92] D. E. Vengroff and J. S. Vitter. Supporting I/O-efficient scientific computation in TPIE. In *Proc. IEEE Symp. on Parallel and Distributed Computing*, 1995. Appears also as Duke University Dept. of Computer Science technical report CS-1995-18.

[93] D. E. Vengroff and J. S. Vitter. Efficient 3-d range searching in external memory. In *Proc. of the 28th Annual ACM Symposium on Theory of Computing (STOC '96)*, Philadelphia, PA, May 1996.

[94] D. E. Vengroff and J. S. Vitter. I/O-efficient scientific computation using TPIE. In *Proceedings of the Goddard Conference on Mass Storage Systems and Technologies*, NASA Conference Publication 3340, Volume II, pages 553–570, College Park, MD, September 1996.

[95] J. S. Vitter. Efficient memory access in large-scale computation (invited paper). In *Symposium on Theoretical Aspects of Computer Science, LNCS 480*, pages 26–41, 1991.

[96] J. S. Vitter and M. H. Nodine. Large-scale sorting in uniform memory hierarchies. *Journal of Parallel and Distributed Computing*, 17:107–114, 1993.

[97] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory, I: Two-level memories. *Algorithmica*, 12(2–3):110–147, 1994.

[98] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory, II: Hierarchical multilevel memories. *Algorithmica*, 12(2–3):148–169, 1994.

[99] B. Zhu. Further computational geometry in secondary memory. In *Proc. Int. Symp. on Algorithms and Computation*, pages 514–522, 1994.