

Planar Point Location For Large Data Sets: To Seek Or Not To Seek

Jan Vahrenhold and Klaus H. Hinrichs

FB 10, Institut für Informatik, Westfälische Wilhelms-Universität, Einsteinstr. 62,
48149 Münster, Germany. {jan,khh}@math.uni-muenster.de

Abstract. We present an algorithm for external memory planar point location that is both effective and easy to implement. The base algorithm is an external memory variant of the bucket method by Edahiro, Kokubo and Asano that is combined with Lee and Yang’s batched internal memory algorithm for planar point location. Although our algorithm is not optimal in terms of its worst-case behavior, we show its efficiency for both batched and single-shot queries by experiments with real-world data. The experiments show that the algorithm benefits from its mainly sequential disk access pattern and significantly outperforms the fastest algorithm for internal memory.

1 Introduction

The well-known problem of *planar point location* consists of determining the region of a planar subdivision that contains a given query point. We assume that a planar subdivision is given by N line segments, and that each segment is labeled with the names of the two regions it separates. In this setting, a point location query can also be answered by *vertical ray shooting*, i.e., by extending a vertical ray from the query point towards $y = +\infty$ and reporting the first segment hit by that ray. In this paper, we are interested in the problem of locating points in a very large planar subdivision that is stored on disk while reducing the overall time spent on I/O operations. Our model of computation is the standard two-level I/O model proposed by Aggarwal and Vitter [3]. In this model, N denotes the number of elements in the problem instance, M is the number of elements fitting in internal memory, and B is the number of elements per disk block, where $M < N$ and $2 \leq B \leq M/2$. An I/O is the operation of reading (or writing) a disk block from (into) external memory. Computations can only be done on elements present in internal memory. Our measures of performance are the number of I/Os used to solve a problem and the amount of space (disk blocks) used. Aggarwal and Vitter [3] considered sorting and related problems in the I/O model and proved that sorting requires $\Theta\left(\frac{N}{B} \log_{M/B} \frac{N}{B}\right)$ I/Os. When dealing with massive data sets, it is likely that there are K point location queries to be answered at a time—also called *batched point location*.

Most algorithms for planar point location exploit hierarchical decompositions which can be generalized to a so-called *trapezoidal search graph* [1]. Using balanced hierarchical decompositions, searching then can be done efficiently in

both the internal and external memory setting. As the query points and thus the search paths to be followed are not known in advance, external memory searching in such a graph will most likely result in unpredictable access patterns and random I/O operations. Disk technologies and operating systems, however, support sequential I/O operations much more efficiently than random I/O operations.

Experimental results published in the database literature [5, 17] reveal that external memory algorithms can benefit even from relatively small clusters of sequentially stored data. We will show how to obtain an algorithm for solving the external memory planar point location problem that fulfills this criterion.

Previous Results. The problem of internal memory planar point location has been studied extensively, and we refer the reader to general surveys on that topic [22, 24]. We now briefly review related work on external memory planar point location and introduce the notation $n = N/B$, $m = M/B$, and $k = K/B$.

Goodrich et al. [16] suggested a batched planar point location algorithm for monotone subdivisions using persistent B -trees that needs $\mathcal{O}((n + K/B) \log_m n)$ I/O operations for preprocessing N segments and locating K query points. Arge, Vengroff, and Vitter [7] perform batched planar point location of K points among N segments in $\mathcal{O}((n + K/B) \log_m n)$ I/O operations using the *extended external segment tree*. The algorithm relies on an external memory variant of fractional cascading [9], and thus the practical realization of their algorithm is rather complicated. Conceptually, the algorithm allows for single-shot queries.

Crauser et al. [11] merge the batch filtering technique of Goodrich et al. [16] into an external variant of Mulmuley’s randomized incremental construction of the trapezoidal decomposition [20] and obtain an $\mathcal{O}((n + K/B) \log_m n)$ batched randomized algorithm. Arge and Vahrenhold [6] presented an algorithm for single-shot point location in a dynamically changing general subdivision building upon an earlier result by Agarwal et al. [2]. The space requirement is linear, queries and insertions take $\mathcal{O}(\log_B^2 n)$ I/O operations, and deletions can be performed in $\mathcal{O}(\log_B n)$ I/O operations. The bound for queries is worst-case whereas the update complexity is amortized.

One of the most interesting open problems related to point location is to generalize any of the above algorithms to three or higher dimensions.

2 A Practical External Memory Algorithm

The I/O-efficient algorithms mentioned in the previous section employ rather complicated constructs that mimic internal memory data structures and are hard to realize. For practical applications, however, it is often desirable to trade asymptotically optimal performance for simpler structures if there is hope for comparable or even faster performance in practice.

Our algorithm for performing batched planar point location in an N -segment planar subdivision is a modification of the bucket method proposed by Edahiro, Kokubo, and Asano [13]. In this section, we describe the basic algorithm and show how to adapt the algorithm to the external memory setting such that both single-shot and batched point location queries can be processed effectively.

2.1 Description of the Basic Algorithm

The bucket method partitions the minimum bounding box of the subdivision into $\mathcal{O}(N)$ equal-sized buckets and assigns to each bucket only the part of the subdivision falling into it. If an original segment crosses one or more grid lines, it is split into a corresponding number of fragments. These splits lead to a replication of the data, and in the worst case we might end up with a super-linear number of fragments. A similar method has been developed independently by Franklin [15] who used a regular grid to compute intersections of line segments.

To answer a point location query, the query point is hashed into the bucket it is contained in by normalizing its coordinates using the *floor*-function. Then comparisons have to be performed only against the portion of the map contained in this bucket. Let x_d be the width and y_d be the height of the minimum bounding box of the map, and let $(x_i^{(1)}, y_i^{(1)})$ and $(x_i^{(2)}, y_i^{(2)})$ denote the endpoints of segment i . Then let $S_x := \left(\sum_{i=1}^N |x_i^{(2)} - x_i^{(1)}| \right) / x_d$ and let $S_y := \left(\sum_{i=1}^N |y_i^{(2)} - y_i^{(1)}| \right) / y_d$. Two constants α_x and α_y are chosen such that they satisfy $\alpha_x / \alpha_y = S_y / S_x$ and $\alpha_x \alpha_y = 1$. Then the numbers of rows and columns are chosen as $N_x = \lfloor \alpha_x \sqrt{N} \rfloor$ and $N_y = \lfloor \alpha_y \sqrt{N} \rfloor$ for a total of $\mathcal{O}(N)$ buckets. An example for partitioning a map is shown in Figure 1(a)—compare this to Franklin’s regular $\sqrt{N} \times \sqrt{N}$ grid shown in Figure 1(b).

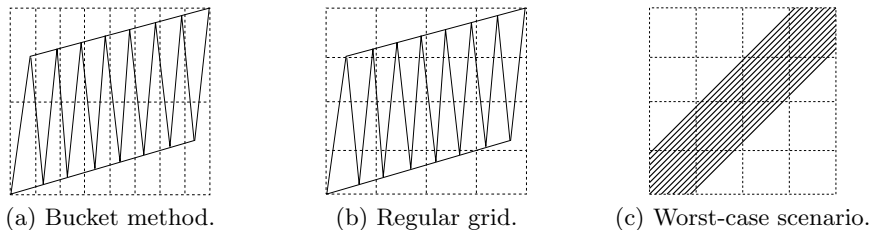


Fig. 1. Partitions induced by grid methods ([13], modified).

Each bucket of the partition stores information about the segments that fall within the bucket or intersect its border. The analysis of the algorithm shows a space requirement of $\mathcal{O}(L)$ where L is the total number of fragments into which the original segments are broken by the partitioning lines, i.e., the total number of fragments in the buckets. In the worst case, L can be on the order of $N\sqrt{N}$ [26] (see also Figure 1(c)) which leads up to $\mathcal{O}(N)$ query time, but for real-world data sets, we have $L \in \mathcal{O}(N)$ and expect $\mathcal{O}(1)$ query time as is confirmed by practical results [8, 13].

This algorithm heavily relies on the *floor*-function which is not an integral component of the Real RAM model of computation. In our situation, however, we can simulate this function on the Real RAM by a binary search on the rows and columns, thereby increasing the query time to $\mathcal{O}(\log_2 \sqrt{N}) = \mathcal{O}(\log_2 N)$. This query time matches the worst-case complexity of several optimal algorithms for internal memory planar point location [14, 18, 23].

each relevant bucket, we load the corresponding portion of the map into main memory where we invoke an internal memory algorithm. To avoid uncontrolled I/O operations due to paging, the size of each bucket in question clearly should not exceed the available main memory size. As will be discussed in Section 3, it is most unlikely that—even for systems with small parameter values M —there will be a real-world data set which causes a single bucket to overflow. However, if there is an overfull bucket, we can detect it during preprocessing, and can further preprocess the bucket (or rather, the column containing it) with an optimal external memory point location algorithm, e.g., the algorithm of Arge et al. [7]. We omit details and only note that this additional preprocessing does not asymptotically increase the overall preprocessing time [26].

The costs for locating K points are composed of the I/O operations spent on sorting and on loading the buckets. The number of buckets to be loaded depends on the distribution of the actual query points. Therefore, we assume for our analysis a “malicious” setting in which the query points fall into $\Theta(\min\{n, K\})$ different buckets. As discussed above, the number of blocks occupied by a single bucket is $\mathcal{O}(1)$ in practice, and then we can load all relevant buckets in $\Theta(\min\{n, K\})$ I/O operations.

If a bucket with N^θ fragments fits into main memory, the I/O costs for locating K^θ query points in this bucket are linear in K^θ/B and N^θ/B , since we can process the query points blockwise as described in Section 2.2. However, if the bucket does not fit into main memory, we have to locate the K^θ query points using the extended external segment-tree which takes $\mathcal{O}(((K^\theta + N^\theta)/B) \log_m(N^\theta/B))$ I/O operations [7]. Since $\Theta(\sqrt{n})$ buckets might contain $\Theta(N)$ fragments each in the worst case, the overall I/O complexity for locating K query points within their containing buckets can be as bad as $\Theta(k \log_m n + \min\{\sqrt{n}, K\} \cdot n \log_m n)$.

Lemma 2. *The batched planar point location for K points in an N -segment planar subdivision can be effected in $\mathcal{O}(k \log_m k + \min\{n, K\})$ I/O operations if each bucket contains $\mathcal{O}(B)$ fragments. In the worst case, i.e., if the segments are broken into $\mathcal{O}(N\sqrt{n})$ fragments, the number of I/O operations is in $\mathcal{O}(k \log_m k + k \log_m n + \min\{\sqrt{n}, K\} \cdot n \log_m n)$.*

If there are no buckets that store an extended external segment-tree, the only external memory operations we use are linear scans and external sorting. Details about how to handle empty buckets and how to answer vertical ray shooting queries (as opposed to point location queries) are given in [26].

Single-Shot Point Location. Locating a single point can obviously be performed by transforming the coordinates of the query point into the coordinates of the corresponding data bucket by normalization. In practice, a single bucket contains only $\mathcal{O}(B)$ segments, so the portion of the map stored in this bucket is loaded into main memory in order to perform internal memory point location. In a worst-case scenario, the $\Omega(M)$ fragments in the bucket are organized by an extended external memory segment tree as described above. Locating a single point in that tree corresponds to a traversal of a root-to-leaf path. Attached to the nodes along this path are at most $\mathcal{O}(N)$ fragments, so the worst-case query time for locating a single point is bounded by $\mathcal{O}(n)$.

Lemma 3. *A single shot query can be performed in $\mathcal{O}(1)$ I/O operations if each bucket contains $\mathcal{O}(B)$ fragments. In the worst case, i.e., if a single bucket contains $\mathcal{O}(N)$ fragments, $\mathcal{O}(n)$ I/O operations may be necessary to answer a single shot query.*

Internal Memory Point-Location. For the internal memory part we use Lee and Yang’s algorithm [19]. This algorithm—a batched variant of Dobkin and Lipton’s *slab method* [12]—performs a plane-sweep with a vertical line V over the query points and the segments. While sweeping, we maintain all segments intersecting V sorted by the y -coordinate of the intersection point. Whenever the line stops at a query point p , a binary search among the segments intersecting V is performed to answer the point location query for p .

We can assume that all K_i query points for bucket i are sorted by increasing x -coordinates since we otherwise can sort them during the initial sorting step at no additional costs. Similarly, we assume that all N_i segments of bucket i are presorted by their left endpoint’s x -coordinate. The space requirement is $\mathcal{O}(N_i)$ because we do not need to keep all query points in main memory but can read them blockwise from their stream. By using an I/O-optimal algorithm for the worst-case setting, we can guarantee to use no more than $\mathcal{O}(M)$ internal memory even if there are $\Omega(M)$ fragments in a single bucket.

Lemma 4. *The internal algorithm for locating K_i points within a bucket storing N_i segments requires $\mathcal{O}(N_i)$ internal space and $\mathcal{O}((N_i + K_i) \log_2 N_i)$ time.*

3 Experimental Results

To verify the results of the previous sections, we implemented our algorithm and performed an extensive experimental evaluation with real-world data sets. Due to the sophisticated structure of the other algorithms discussed in Section 1, there exists no implementation of these methods, and thus we are not able to compare our results with an I/O-efficient algorithm. We also do not compare against standard index structures used in spatial databases as these indexes are tree-like structures, and we expect random access patterns similar to those discussed in the introduction. Instead, we compare with the original bucket method of Edahiro et al. that has been identified as the most efficient internal memory point location algorithm for practical data sets with respect to both speed and space requirements [8, 13].

Our Implementation. We implemented the algorithms in C++ using TPIE [4, 27], a collection of templated functions and classes that allow for simple and efficient implementation of external memory algorithms. Since our algorithm basically consists of a sequence of sequential read and write scans, we chose the `stdio`-implementation of TPIE which uses standard UNIX™ streams and thus benefits from the operating system’s caching and prefetching mechanisms. We compiled all programs using the GNU C++ compiler (version 2.8.1), with `-O3` level of optimization. All internal memory data structures were realized using the *Standard Template Library* (STL) [21].

All experiments were performed on a SunTM SparcSTATIONTM 20 Model 51 running SolarisTM 2.6 with 32 MB RAM, a 50 MHz SuperSPARCTM STP1020A processor, and a SeagateTM ST 34520WC (4.2 GB capacity, 9.5 ms average read time, 40 MB/sec. peak throughput) local disk. Motivated by the empirical observation by Chiang [10] that I/O algorithms perform much better when the available main memory is not exhausted completely but only some fraction is used, the memory-management system of TPIE was configured to use no more than 12 MB of main memory at a time. The remaining free memory was available to the block caching mechanism of the operating system.

Our implementation of the internal memory variant was allowed to use all of the available virtual memory, i.e., as much as 480 MB of swap space. To reduce the space requirements of the internal memory variants as much as possible, we modified the original algorithm in two ways. First, we did not require the input data to be present in main memory at the beginning of the algorithm. Instead, we used the stream management of TPIE to scan the input stream in order to compute the grid’s granularity. Then, the segments were scanned, broken into fragments, and directly moved to the buckets’ lists. The second modification aimed at the data stored within each bucket. In the original variant, each bucket contains a list of segments and two additional lists of pointers to items in this list to guarantee linear running time of the point location algorithm within each bucket. Since the average number of segments per bucket was at most seven and the maximum size of a bucket did not exceed 85 segments (see Table 1), we decided not to maintain these additional lists which would have resulted in allocating additional (virtual) memory but answered the point location query by an $\mathcal{O}(N \log_2 N)$ plane-sweep algorithm similar to the one described in Section 2.2.

Characteristics of the Test Data. We used Digital Line Graph data sets in 1:100,000 scale provided by the U.S. Geological Survey [25]. The characteristics of the data are described in Table 1. Each line segment needs four double-precision floating-point numbers for the coordinates of its endpoints, two integers for identifying the regions on both sides and one integer for the number of the bucket it is assigned to. With a page size of 4,096 bytes, the number B_S of segments that fit into a disk block is 73. Since each point needs two coordinates and one label, the number B_P of points that fit into one block is 170.

Data Set	Class	Size	Objects	Buckets		Max. Objects	
				internal	external	int.	ext.
G1 ^a	Hydro	25 MB	550,891	552,123	6,571	37	549
G2 ^a	Roads	27 MB	597,099	598,437	7,189	45	1,199
G3 ^b	Hydro	15 MB	336,555	337,820	4,030	74	747
G4 ^b	Roads	56 MB	1,224,606	1,226,732	14,521	47	908
G5 ^{a,b}	Roads	83 MB	1,821,705	1,824,589	21,481	85	1,881

Table 1. Data sets for building the grid (sizes given for raw data).

^a Data quadrangles for Albany, Amsterdam, Auburn, Binghampton, Elmira, Glenn Falls, Gloversville, Norwich, Pepacton, Pittsfield, Syracuse, and Utica.

^b Data quadrangles for Block Island, Bridgeport, Long Branch, Long Island, Middletown, New Haven, Newark, and Trenton.

Grid Construction. The first set of experiments was dedicated to the preprocessing phase of the point location algorithms. We performed several runs of the preprocessing phase and present the statistics and averaged timings in Figure 2. As can be seen from Figure 2(a), the number of fragments produced by the internal memory algorithm is at most 50% more than the number of original segments. This and the internal memory data structures for the grid and the buckets increase the main memory space requirement by a factor of up to three—note that the algorithm constructs as many buckets as original segments. The penalties for using virtual memory are immense: the algorithm spends more than 2/3 of the overall running time waiting for I/O operations caused by page faults (labeled “I/O time” in Figure 2(b)). In contrast, the space requirement of the external memory variant is very moderate: the number of fragments does not exceed the number of segments by more than 10%. This is true for data sets which produce almost no empty buckets (G1 and G2) and for data sets which produce a grid with up to 50% empty buckets (G3, G4, and G5). Given $B_S = 73$, the average size of non-empty buckets is more than three blocks. With 12 MB internal memory available and 56 bytes required to store a segment, $M = 12 \cdot 1,024^2 / 56 \approx 224,500$ segments can be packed into one single bucket—compare this to a maximum load of around 1,900 fragments per bucket for our real-world data sets.

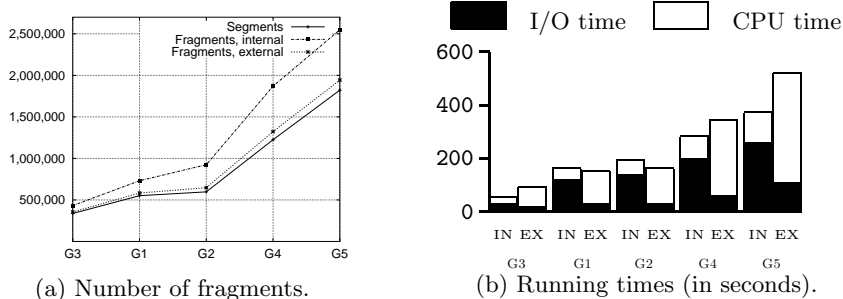


Fig. 2. Constructing the grid in internal (IN) and external (EX) memory.

The external memory algorithm is accessing the disk for reading, writing, and sorting the data, and the “I/O time” given in Figure 2(b) reflects the time spent waiting for these operations to complete. In contrast to the internal memory algorithm, the external memory variant is sorting the fragments, and the internal memory component of this process contributes a significant amount to the overall running time. What is of much more practical relevance than preprocessing time, is the performance of answering point location queries which we discuss next.

Single-Shot and Batched Queries. We performed two sets of point location experiments for each grid where we located the same query points—the vertices of the railroads and pipelines line data for the corresponding area—in single-shot and batched mode. To speed up the single-shot queries, we first use a linear number of tests to determine all segments of the current bucket whose projections onto the x -axis overlap the x -coordinate of the query point. If this

set is empty, we invoke the standard plane-sweep algorithm, making sure that we do not advance past the query point. If there are segments overlapping the query point, we locate the query point in a y -table containing all these segments.

In our first set of experiments, we used the vertices extracted from the railroads and pipelines data sets as query points. The results presented in Figure 3 show that our proposed batched algorithm outperforms the internal memory algorithm by a factor of up to seven. But even if we compare the running times of single shot queries, we see that the external memory variant is faster by at least a factor of two. The only exception is the data set G3, where the internal memory algorithm is up to 30% faster than the external memory single-shot variant. This comes as no surprise since the data set G3 is by far the smallest data set (see Table 1) and thus the least amount of swap space is needed.

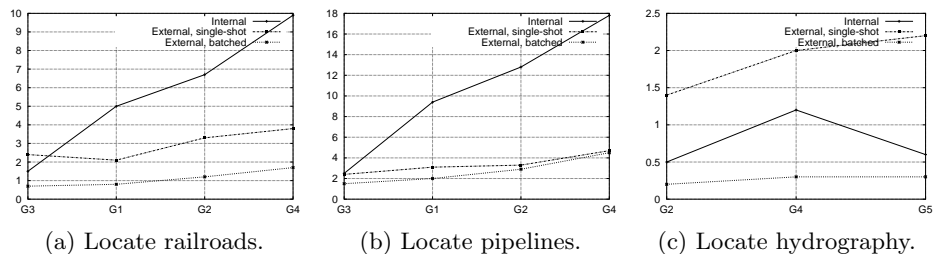


Fig. 3. Average query time per point (time in milliseconds).

The internal memory algorithm spends more than 95% of the running time on waiting for I/O operations due to paging. Since the hardware we are using has a relatively slow CPU and a relatively fast disk, we expect an even worse performance ratio for systems with faster processors and/or slower disks.

To investigate the behavior of the three algorithms for larger sets of query points, we performed another set of experiments, where we located the vertices extracted from the hydrography data sets in the grids constructed for the roads data sets. The results presented in Figure 3(c) show indeed that the performance of the external memory single-shot algorithm degrades the larger the set of query points is. This can be explained by the fact that now there are up to 80 query points per bucket. The single-shot algorithm needs to perform at least 80 linear scans per bucket, and each bucket contains up to 213 fragments on the average. In contrast, the batched algorithm only performs one plane-sweep per bucket after the initial sorting of the points. As we can see from Figure 3(c), the batched external memory variant is up to eight times faster than the single-shot external memory variant. Since the external memory single-shot variant spends more than 95% of its running time on internal memory operations, it additionally suffers from the relatively slow processor. In contrast, the internal memory algorithm benefits from the relatively fast disk and is able to outperform the external memory single-shot algorithm by a factor of up to four. The batched variant also spends up to 90% of its running time on internal memory operations, but is still able to perform more than three times faster than the internal memory algorithm.

Effects of Locality in the Query Pattern. The data sets used for our experiments store the segment data in form of short polygonal chains, i.e., in sequences of adjacent points. Since we constructed each data set by concatenating several quadrangles of line data, the data files (and hence the extracted point data as well) were stored as a sequence of clusters. In the experiments described above, the query points were processed in exactly the same order. To investigate effects of locality in the query pattern, we repeated all experiments with the query points randomly shuffled (using STL’s `random_shuffle` function).

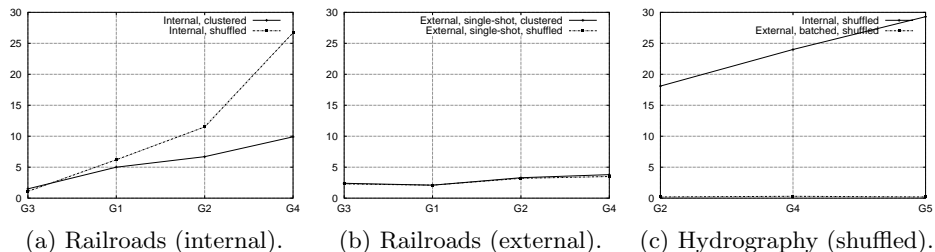


Fig. 4. Average query time per point (time in milliseconds).

Figures 4(a) and 4(b) contrast the query performance of both single-shot algorithms for clustered and shuffled query points. Since the batched external algorithm sorts the query points according to the buckets they fall into and then processes the sorted sequence, the query performance is not affected by the order of the query points. Hence, we do not repeat the results for the batched variant.

Figure 4(a) shows that the performance of the internal memory algorithm degrades if the locality of the input data is destroyed (again, the smallest data set is an exception). For clustered data, it is likely that the page containing the bucket in question is still in main memory because of a previous request, and in the best case, the cache mechanism of the memory management unit helps speeding up accesses. If the query points—and hence the buckets—are randomly shuffled, the page requests produce not only translation-look-aside buffer (TLB) misses, but in the worst case will cause page faults as well. As a consequence the TLB does not speed up but slow down the algorithm, and we face an increased number of (uncontrolled) I/O operations. Figure 4(a) shows that the performance gets slowed down by a factor of almost three. On the other hand, the I/O-wait is still 97% of the running time. Since the computation done by the internal memory single-shot algorithm is independent from the order of queries, this is a clear indication for a large number of TLB misses.

The performance of the external memory single-shot algorithm, on the other hand, does not seem to depend on the access pattern, and for some situations there is even a minor improvement—see Figure 4(b). This is an indication for I/O operations being overlapped with internal memory computation by the system. The average query time per point is less than 4 ms—this corresponds to less than twice the average single-track seek time of the disk drive. For the internal memory variant, we have up to 26 ms average query time which is more than the full seek time of the disk drive.

Finally, we present a comparison of the internal memory algorithm with the batched external memory variant for locating large sets of query points in shuffled order—see Figure 4(c). This setting reflects the scenario for which our algorithm was designed: locating a large, not necessarily clustered, set of points in a large collection of segments. Again, the internal memory algorithm spends more than 95% of the overall running time waiting for I/O operations, and the average query time per point varies between 18 and 27 ms. Our proposed batched algorithm, on the other hand, answers all point location queries very fast: the average query time per point is at most 0.3 ms. In the best situation, our algorithm is able to perform more than 130 times faster than the internal memory single-shot point location. It should be noted that our algorithm does only temporarily occupy a fixed amount of internal memory. This means that—without the need for paging or swapping—the algorithm can be used on a workstation where other software, e.g., subsystems of a Geographic Information System, needs to reside in main memory.

4 Conclusions

We have presented an algorithm for external memory point location along with experimental results of a comparison with the fastest and most space-efficient internal memory method. To our knowledge, this is the first experimental evaluation of an algorithm for external memory point location. The comparison with the fastest and most space-efficient internal memory method shows a clear superiority of the batched algorithm for all settings. The main reason for this superiority is the mainly sequential disk access pattern of our batched algorithm that benefits from operating systems and disk controllers optimized for sequential I/O operations. Due to the mainly sequential disk access pattern, our batched algorithm will be a strong competitor of any I/O-optimal algorithm with respect to performance for real-world data sets. It remains open, though, to verify this conjecture by an experimental evaluation. Finally, note that the design of the algorithm allows for an almost straightforward extension to utilize multiple disks and processors—an important aspect when dealing with large data sets.

References

1. U. Adamy and R. Seidel. On the exact worst case query complexity of planar point location. *Proc. 9th Annual ACM-SIAM Symp. Discrete Algorithms*, 609–618. 1998.
2. P. Agarwal, L. Arge, G. Brodal, and J. Vitter. I/O-efficient dynamic point location in monotone planar subdivisions. *Proc. 10th Annual ACM-SIAM Symp. Discrete Algorithms*, 11–20. 1999.
3. A. Aggarwal and J. Vitter. The input/output complexity of sorting and related problems. *Comm. ACM*, 31(9):1116–1127, 1988.
4. L. Arge, R. Barve, O. Procopiuc, L. Toma, D. Vengroff, and R. Wickremesinghe. TPIE user manual and reference, edition 0.9.01a. Duke University, North Carolina, <<http://www.cs.duke.edu/TPIE/>>, 1999. (accessed 12 Jul. 1999).
5. L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, J. Vahrenhold, and J. Vitter. A unified approach for indexed and non-indexed spatial join. *Proc. 7th Intl. Conf. Extending Databases Technology*, LNCS 1777, 413–429. 2000.

6. L. Arge and J. Vahrenhold. I/O-efficient dynamic planar point location. *Proc. 16th Annual ACM Symp. Computational Geometry*, 191–200. 2000.
7. L. Arge, D. Vengroff, and J. Vitter. External-memory algorithms for processing line segments in geographic information systems. *Proc. 3rd Annual European Symp. Algorithms*, LNCS 979, 295–310. 1995.
8. K. Baumann. Implementation and comparison of five algorithms for point-location in trapezoidal decompositions. Master’s thesis, Fachbereich Mathematik und Informatik, Westfälische Wilhelms-Universität Münster, Germany, 1996. (in German).
9. B. Chazelle and L. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1:133–162, 1986.
10. Y.-J. Chiang. Experiments on the practical I/O efficiency of geometric algorithms: Distribution sweep vs. plane sweep. *Computational Geometry: Theory and Applications*, 9(4):211–236, March 1998.
11. A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E. Ramos. Randomized external-memory algorithms for some geometric problems. *Proc. 14th Annual ACM Symp. Computational Geometry*, 259–268. 1998.
12. D. Dobkin and R. Lipton. Multidimensional searching problems. *SIAM J. Comput.*, 5:181–186, 1976.
13. M. Edahiro, I. Kokubo, and T. Asano. A new point-location algorithm and its practical efficiency: Comparison with existing algorithms. *ACM Trans. Graphics*, 3(2):86–109, 1984.
14. H. Edelsbrunner, L. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM J. Comput.*, 15(2):317–340, 1986.
15. W. R. Franklin. Adaptive grids for geometric operations. *Proc. 6th Intl. Symp. Automated Cartography (Auto-Carto Six)*, vol. 2, 230–239. 1983.
16. M. Goodrich, J.-J. Tsay, D. Vengroff, and J. Vitter. External-memory computational geometry. *Proc. 34th Annual IEEE Symp. Found. Computer Science*, 714–723. 1993.
17. K. Kim and S. Cha. Sibling clustering of tree-based spatial indexes for efficient spatial query processing. *Proc. 1998 ACM CIKM Intl. Conf. Information and Knowledge Management*, 398–405. 1998.
18. D. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Comput.*, 12(1):28–35, 1983.
19. D.-T. Lee and C. Yang. Location of multiple points in a planar subdivision. *Inf. Proc. Letters*, 9(4):190–193, 1979.
20. K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms*. Prentice Hall, 1994.
21. D. Musser and A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley, 1996.
22. F. Preparata and M. Shamos. *Computational Geometry: An Introduction*. Springer, 2nd edition, 1988.
23. N. Sarnak and R. Tarjan. Planar point location using persistent search trees. *Comm. ACM*, 29:669–679, 1986.
24. J. Snoeyink. Point location. *Handbook of Discrete and Computational Geometry*, Discrete Mathematics and its Applications, chapter 30, 559–574. CRC Press, 1997.
25. U.S. Geological Survey. 1:100,000-scale digital line graphs (DLG). <http://edcwww.cr.usgs.gov/doc/edchome/ndcddb/ndcdb.html> (accessed 26 May 1999).
26. J. Vahrenhold. *External Memory Algorithms for Geographic Information Systems*. PhD thesis, Fachbereich Mathematik und Informatik, Westfälische Wilhelms-Universität Münster, Germany, 1999.
27. D. Vengroff. A transparent parallel I/O environment. In *Proc. DAGS Symp.*, 1994.