Experiments on the Practical I/O Efficiency of Geometric Algorithms: Distribution Sweep vs. Plane Sweep

Yi-Jen Chiang

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-95-07
May 1995

Extended abstract to appear in Proc. Workshop on Algorithms and Data Structures (1995)

Experiments on the Practical I/O Efficiency of Geometric Algorithms: Distribution Sweep vs. Plane Sweep *

Yi-Jen Chiang
yjc@cs.brown.edu
Department of Computer Science
Brown University
Providence, R. I. 02912–1910

Abstract

We present an extensive experimental study comparing the performance of four algorithms for the following *orthogonal segment intersection problem*: given a set of horizontal and vertical line segments in the plane, report all intersecting horizontal-vertical pairs. The problem has important applications in VLSI layout and graphics, which are large-scale in nature. The algorithms under evaluation are distribution sweep and three variations of plane sweep. Distribution sweep is specifically designed for the situations in which the problem is too large to be solved in internal memory, and theoretically has optimal I/O cost. Plane sweep is a well-known and powerful technique in computational geometry, and is optimal for this particular problem in terms of internal computation. The three variations of plane sweep differ by the sorting methods (external vs. internal sorting) used in the preprocessing phase and the dynamic data structures (B tree vs. 2-3-4 tree) used in the sweeping phase. We generate the test data by three programs that use a random number generator while producing some interesting properties that are predicted by our theoretical analysis. The sizes of the test data range from 250 thousand segments to 2.5 million segments. The experiments provide detailed quantitative evaluation of the performance of the four algorithms, and the observed behavior of the algorithms is consistent with their theoretical properties. This is the first experimental work comparing the practical performance between external-memory algorithms and conventional algorithms with large-scale test data.

(April 1995)

*An extended abstract of this paper will be presented at the *4th Workshop on Algorithms and Data Structures*, Kingston, Ontario, Canada, August, 1995. Research supported in part by the National Science Foundation, by the U.S. Army Research Office, and by the Office of Naval Research and the Advanced Research Projects Agency.

1 Introduction

Input/Output (I/O) communication between fast internal memory and slower external memory is the major bottleneck in many large-scale applications. The significance of this bottleneck is increasing as internal computation gets faster, and especially as parallel computing gains popularity. Due to this important fact, more and more attention has been given to the development of I/O-efficient algorithms in recent years. Most of the developed algorithms, however, are shown to be efficient only *in theory*, and their performance *in practice* is yet to be evaluated. In particular, all such algorithms assume that the internal computation is free compared to the I/O cost, which also has to be justified in practice. In this paper, we establish the practical efficiency of one such algorithm by an extensive experimental study.

1.1 Previous Related Work

As mentioned above, most of the previous work on I/O-efficient computation is theoretical. Early work on algorithms for parallel disk systems concentrates largely on fundamental problems such as sorting, matrix multiplication, and FFT [1, 24, 31]. The main focus of this early work is therefore directed at problems that involve permutation at a basic level. Indeed, just the problem of implementing various classes of permutation has been a central theme in external-memory I/O research [1, 12, 13, 15, 31].

More recently, external-memory research has moved towards solving graph and geometric problems. Work on graph problems includes transitive closure computations [28], some graph traversal problems [19], and memory management problems for maintaining connectivity information and paths on graphs [16]. Recently, Chiang *et al.* [10] present a collection of new techniques for designing and analyzing I/O-efficient graph algorithms, and apply these techniques to a wide variety of specific problems. For geometric problems, Goodrich *et al.* [20] study a number of problems in computational geometry and develop several paradigms for I/O-optimal geometric computations. Further results in this area have been obtained in [17, 32]. Also, Kanellakis *et al.* [21] and Ramaswamy and Subramanian [26, 27] give efficient data structures for performing range searching in external memory. Very recently, a new data structure called *buffer tree* and its applications are given in [2, 3], and an external-memory version of the directed topology tree ([18]) called *topology B-tree* is given in [9].

For excellent examples of experimental work in computational geometry, see Bentley [5, 6, 7, 8]. As for experimental work on I/O-efficient computation, very recently Vengroff has built an environment called TPIE for programming external-memory algorithms as he proposed earlier in [29], and also Vengroff and Vitter [30] have reported some benchmarks of TPIE on sorting and matrix multiplication. This work, however, is mainly on providing a programming environment and not on performance comparisons between external-memory algorithms and conventional algorithms. Other than this, we do not know of any previous experimental work on I/O-efficient computation.

1.2 Our Results

We present an extensive experimental study comparing the performance of four algorithms for the following *orthogonal segment intersection problem*: given a set of horizontal and vertical line segments in the plane, report all intersecting horizontal-vertical pairs. The problem has important applications in VLSI layout and graphics, which are large-scale in nature. The algorithms under evaluation are distribution sweep of Goodrich *et al.* [20] and three variations of plane sweep [25]. Distribution sweep theoretically has optimal I/O cost [20]. Plane sweep is a well-known and powerful technique in computational geometry, and is optimal for this particular problem in terms

of internal computation [25]. The three variations of plane sweep differ by the sorting methods (external merge sort [1] vs. internal merge sort) used in the preprocessing phase and the dynamic data structures (B tree [4, 11, 14] vs. 2-3-4 tree [14]) used in the sweeping phase. We generate the test data by three programs that use a random number generator while producing some interesting properties that are predicted by our theoretical analysis. The sizes of the test data range from 250 thousand segments to 2.5 million segments. The experiments provide detailed quantitative evaluation of the performance of the four algorithms, and the observed behavior of the algorithms is consistent with their theoretical properties.

The contribution of this work can be summarized as follows:

- We have presented the first experimental work comparing the practical performance between external-memory algorithms and conventional algorithms with large-scale test data.
- We have generated test data with interesting properties that are predicted by our theoretical analysis. In particular, we give techniques for analyzing the expected number of intersections and the average number of vertical overlaps among vertical segments in the data sets generated, which may be of independent interest.
- We have implemented distribution sweep, three variations of plane sweep and external merge sort under a uniform experimental framework so that some resource usage can be parameterized and various statistic information related to performance can be obtained and analyzed. The implementations handle all degeneracies and are robust.
- We have presented the first experimental study on the four algorithms for the important orthogonal segment intersection problem with large-scale test data, and established the practical efficiency of distribution sweep.

1.3 Organization of the Paper

The rest of the paper is organized as follows. In Section 2 we overview the four algorithms under evaluation. Details on the experimental setting are given in Section 3. In Section 4, we summarize our experimental results in nine charts and give a comparative analysis of the performance of the four algorithms. Finally, we conclude the paper in Section 5.

2 The Algorithms Under Evaluation

The four algorithms considered in this paper are distribution sweep, denoted `Distribution`, and three variations of plane sweep, denoted `B-Tree`, `234-Tree`, and `234-Tree-Core`, described next. To discuss the time complexity, let N be the total number of segments in the given input, K the number of intersecting pairs that must be reported, and M and B the numbers of segments that can fit into the main memory and into a page, respectively. Each I/O operation transfers one page of data.

2.1 Three Variations of Plane Sweep

The well-known *plane sweep* paradigm [25] is a powerful technique in computational geometry, and is optimal for the orthogonal segment intersection problem in terms of internal computation. The method consists of preprocessing and sweeping phases. In the preprocessing phase, we sort all endpoints by the y-coordinates in non-decreasing order. In the sweeping phase, we (conceptually) use a horizontal sweep line to sweep the plane from bottom to top, and use a dynamic data structure, typically a search tree, to keep the objects currently intersecting the sweep line. Operations (object

insertions/deletions or queries) are performed only when some *events* are encountered by the sweep line. In our problem, the objects being maintained are vertical segments, and the events are the endpoints of all segments. When a bottom endpoint of a vertical segment is encountered, it is inserted to the data structure; the segment remains intersecting the sweep line until its top endpoint is encountered, at which time it is deleted from the data structure. When an endpoint of a horizontal segment s is encountered, we search the data structure to find and report all vertical segments intersecting s, i.e., those segments whose x-coordinates are contained in the x-interval spanned by s. The sequence of events during the sweeping process is given by sorting in the preprocessing phase. Using any dynamic balanced tree, plane sweep takes optimal $O(N \log N)$ time in terms of internal computation.

Our three variations of plane sweep differ by the sorting methods and the dynamic data structures used. The first variation, `B-Tree`, uses external merge sort [1] and a B tree [4, 11, 14]; this is a direct way to implement plane sweep in secondary memory. The number of I/O operations performed in the first phase is optimal $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ [1], and in the second phase is $O(N \log_B \frac{N}{B} + \frac{K}{B})$. The second variation, `234-Tree`, uses external merge sort and a 2-3-4 tree [14], viewing the internal memory as having an infinite size and letting the virtual memory feature of the OS handle page faults during the second (sweeping) phase. It has the same I/O cost in the first phase and $O(N \log N + \frac{K}{B})$ I/O cost in the second phase. Finally, the third variation, `234-Tree-Core`, uses internal merge sort and a 2-3-4 tree, letting the OS handle page faults all the time. The I/O costs in the first and second phases are $O(N \log N)$ and $O(N \log N + \frac{K}{B})$, respectively. Viewing the internal memory as virtually having an infinite size is conceptually the simplest, and is actually the most commonly used strategy today in practice.

2.2 Distribution Sweep

Distribution sweep [20] is an external-memory version of plane sweep based on the subdivision technique used in the "distribution sort" algorithms of [1, 23, 31]. When applied to the orthogonal segment intersection problem, it works as follows.

In the preprocessing phase, we sort the endpoints of all segments into two lists, one by x and the other by y. Again we use external merge sort. The list sorted by x is used to locate the medians which we will use to split the input into $\lfloor \frac{M}{B} \rfloor$ vertical strips. The list sorted by y is used to perform the sweep, moving bottom up. Associated with each strip γ_i is an *active list* A_i that maintains the vertical segments inside γ_i that intersect the horizontal sweep line.

During the sweep, if the bottom endpoint of a vertical segment is encountered, it is inserted into the active list A_i of the strip in which the segment lies, and later it is deleted from A_i when its top endpoint is encountered. When an endpoint of a horizontal segment s is met, we locate the two strips γ_i and γ_j containing the two endpoints of s, $i <= j$, and report all vertical segments currently in the active lists $A_{i+1}, ..., A_{j-1}$. Note that strips $\gamma_{i+1}, ..., \gamma_{j-1}$ are completely spanned horizontally by s. This reports all vertical segments intersecting s except for those lying in γ_i and γ_j, which will be processed in the next level of recursion. After the sweep is complete in the top level, we recursively perform the same procedure to each strip γ_i. The recursive process continues until the size of the subproblem falls below M, at which time we simply solve the problem in main memory.

Based on the above idea, distribution sweep actually uses a "lazy deletion" policy. Each active list A_i can be viewed as a stack with the topmost block maintained in the internal memory. Whenever the internal block is full, it is written to the external memory. Now, the top endpoints of vertical segments become "no action" events, and the deletions are performed during reporting: in the reporting process, instead of just reporting each vertical segment t in active list A_i, we check the

top endpoint of t to see if its deletion time has passed. If so we delete t, otherwise we report t and retain t in A_i. The total I/O cost is $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}+\frac{K}{B})$, which is optimal. Notice that distribution sweep needs two sortings as opposed to just one sorting in plane sweep.

3 Experimental Setting

3.1 Generation and Analysis of the Test Data

We use three programs to generate our test data; all of them use a random number generator that gives a uniform distribution. The programs randomly generate several attributes of a segment such as its length, position, and whether it is horizontal or vertical, and also maintain certain simple structures to make the test data interesting.

We must use the random number generator carefully to obtain interesting test data. Observe that if we just randomly generate segments with lengths uniformly distributed over $[0, N]$, place them randomly (and uniformly in each dimension) in a square with side length N, and make horizontal and vertical segments equally likely to occur, then the number K of intersections is $\Theta(N^2)$ (obtained by the analysis given below). In this case, any algorithm has $\Omega(\frac{N^2}{B})$ reporting I/O cost, which dominates the searching I/O costs in all four algorithms under evaluation. In fact, the following brute-force algorithm performs equally well: for each segment, check all the other $N-1$ segments for intersections; the I/O cost is $O(N\cdot\frac{N}{B})=O(\frac{N^2}{B})$. Certainly this kind of test data is undesirable.

Our three programs are denoted `gen-short`, `gen-long`, and `gen-rect`, and the data sets generated are correspondingly denoted `data-short`, `data-long` and `data-rect`. We try to generate test data with small number of intersections so that the searching I/O cost dominates the reporting cost. Also, it is conceivable that the number of vertical overlaps among vertical segments at a given time decides the tree size in that moment of plane sweep and also the total size of the active lists at that time of distribution sweep. Thus the number of vertical overlaps may affect the performance of the four algorithms. Our three programs generate test data with distinct structures regarding the number of intersections and the number of vertical overlaps. Also, all three programs decide whether the current segment being generated is horizontal or vertical by tossing a fair coin (simulated by using the random number generator).

Program `gen-short` generates short segments whose lengths are uniformly distributed over $[0, \sqrt{N}]$. The segments are randomly placed in an $N \times N$ square Q. More specifically, for the left endpoints of horizontal segments, the distances to the left and bottom sides of Q are uniformly distributed over $[0, N-\sqrt{N}]$ and over $[0, N]$, respectively. Similarly, for the bottom endpoints of vertical segments, the distances to the left and bottom sides of Q are uniformly distributed over $[0, N]$ and over $[0, N-\sqrt{N}]$, respectively. To simplify the discussion, we assume that the coordinate of the lower-left corner of Q is $(0, 0)$ (in the actual program this coordinate is $(-0.5N, -0.5N)$).

We now analyze the expected number of horizontal-vertical intersecting pairs in `data-short`. Let K be a random variable for the number of such pairs. We can express K by $K=\sum_{1\le i,j\le N,\ i\neq j} K_{ij}$, where for $1\le i,j\le N$, $i\neq j$, K_{ij} is a 0-1 random variable defined by

$$K_{ij}=\begin{cases}1 & \text{if segment } s_i \text{ is horizontal, segment } s_j \text{ is vertical, and } s_i\cap s_j\neq\emptyset\\ 0 & \text{otherwise.}\end{cases}$$

Clearly, we have

$$\Pr\{K_{ij}=1\}=\Pr\{s_i \text{ is horizontal and } s_j \text{ is vertical}\}\cdot\Pr\{\text{horizontal } s_i\cap\text{vertical } s_j\neq\emptyset\}.$$

The first term is $\frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$, so let us focus on computing the second term. In the following, s_i is horizontal and s_j is vertical. For s_i and s_j to intersect, the y-coordinate of s_i is contained in the y-interval spanned by s_j (denoted by $y(s_i) \in I_y(s_j)$), and similarly for the x-dimension. Define P_0 to be the conditional probability $\Pr\{y(s_i) \in I_y(s_j) \mid y(b(s_j)) = t, |s_j| = h\}$, where $y(b(s_j))$ is the y-coordinate of the bottom endpoint of s_j, and $|s_j|$ is the length of s_j. For any $t \in [0, N - \sqrt{N}]$ and $h \in [0, \sqrt{N}]$ we have $P_0 = \Pr\{y(s_i) \in [t, t+h] \mid y(b(s_j)) = t, |s_j| = h\} = \frac{h}{N}$, since $y(s_i)$ is uniformly distributed over $[0, N]$. Let $f(t)$ be the probability density function of $y(b(s_j))$. Since $P_0 = \frac{h}{N}$ is independent of t, we have $\Pr\{y(s_i) \in I_y(s_j) \mid |s_j| = h\} = \int_{-\infty}^{+\infty} P_0 \cdot f(t)\, dt = P_0 \cdot \int_{-\infty}^{+\infty} f(t)\, dt = \frac{h}{N}$. Moreover, $\Pr\{y(s_i) \in I_y(s_j) \mid |s_j| = h, |s_i| = w\} = \Pr\{y(s_i) \in I_y(s_j) \mid |s_j| = h\} = \frac{h}{N}$, since events $\{y(s_i) \in I_y(s_j)\}$ and $\{|s_i| = w\}$ are conditionally independent given $|s_j| = h$. By a similar argument, we have $\Pr\{x(s_j) \in I_x(s_i) \mid |s_j| = h, |s_i| = w\} = \frac{w}{N}$, and therefore $\Pr\{s_i \cap s_j \neq \emptyset \mid |s_j| = h, |s_i| = w\} = \frac{h}{N} \cdot \frac{w}{N}$. Now $|s_j|$ is uniformly distributed over $[0, \sqrt{N}]$, namely,

$$g(h) = \begin{cases} \frac{1}{\sqrt{N}} & \text{if } 0 \leq h \leq \sqrt{N} \\ 0 & \text{otherwise} \end{cases}$$

is the probability density function of $|s_j|$; similarly for $|s_i|$. Thus

$$\Pr\{\text{horizontal } s_i \cap \text{vertical } s_j \neq \emptyset\} = \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} \frac{h}{N} \cdot \frac{w}{N} \cdot g(h) \cdot g(w)\, dh\, dw = \frac{1}{4N}.$$

Therefore we have $\Pr\{K_{ij} = 1\} = \frac{1}{16N}$. It follows that $E[K] = N(N-1) \cdot E[K_{ij}] = \frac{1}{16}(N-1)$. Figure 1 shows the actual numbers of intersections with respect to data size N, for all three data sets generated. The observed K values are indeed $\frac{1}{16}N$ for `data-short`.

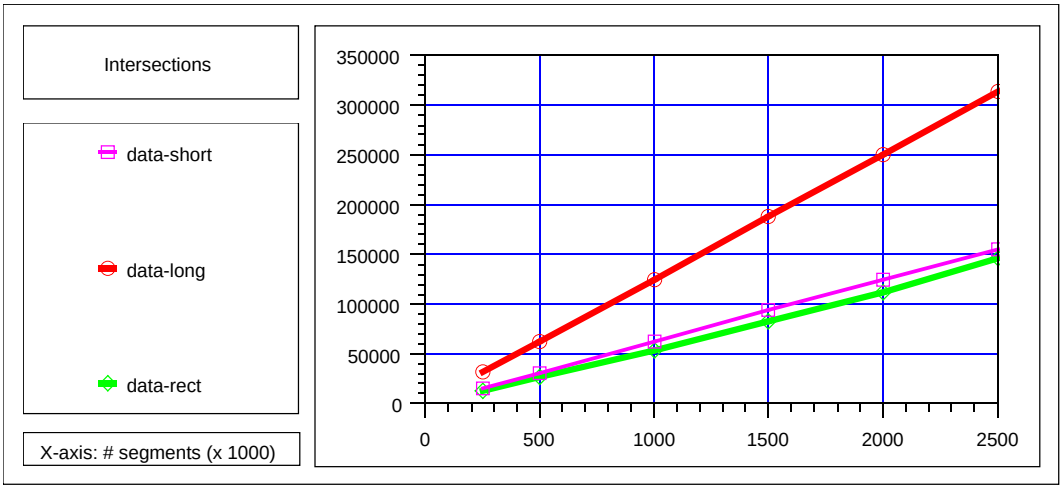

Figure 1: The actual numbers of intersections with respect to the number of segments in data sets `data-short`, `data-long` and `data-rect`.

Now we proceed to analyze for `data-short` the average number of vertical overlaps among vertical segments, that is, the average number of vertical segments "cut" by the horizontal sweep line l *when* l *is passing through an event*. The average is taken over all sweeping events. Notice that this average number is exactly the average number of items stored in the data structure when an update/query operation is performed during plane sweep. A related problem has been studied in [22]. Intuitively, we would estimate this average number to be proportional to the average length of vertical segments, which is $\Theta(\sqrt{N})$. A rigorous analysis is given next.

Let V be a random variable for the number of vertical segments cut by l for an event. Our goal is to compute $E[V]$. We can express V by $V = V_1 + V_2 + \cdots + V_N$, where for $1 \le j \le N$, V_j is a 0-1 random variable defined by

$$V_j = \begin{cases} 1 & \text{if segment } s_j \text{ is vertical and } s_j \cap l \neq \emptyset \\ 0 & \text{otherwise.} \end{cases}$$

Clearly, we have

$$\Pr\{V_j = 1\} = \Pr\{s_j \text{ is vertical}\} \cdot \Pr\{\text{vertical } s_j \cap l \neq \emptyset\}. \tag{1}$$

The first term is $\frac{1}{2}$, so let us compute the second term. In the following s_j is vertical. If $y(l)$ were uniformly distributed over $[0, N]$, then letting l play the role of a horizontal segment s_i and applying the previous method for $E[K]$, we would have $\Pr\{s_j \cap l \neq \emptyset \mid |s_j| = h\} = \Pr\{y(l) \in I_y(s_j) \mid |s_j| = h\} = \frac{h}{N}$, and $\Pr\{V_j = 1\}$ could be computed accordingly. But this is not the case.

Recall that the positions of l depend on the positions of the events. Let the probability density function of $y(l)$ be defined by

$$k(u) = \begin{cases} k_1(u) & \text{if } 0 \le u < \sqrt{N} \\ k_2(u) & \text{if } \sqrt{N} \le u \le N - \sqrt{N} \\ k_3(u) & \text{if } N - \sqrt{N} < u \le N \\ 0 & \text{else.} \end{cases}$$

If there were only horizontal segments, then we would have $k_1(u) = k_2(u) = k_3(u) = \frac{1}{N}$. Now consider including also the vertical segments. There is no bottom endpoint q with $y(q) \in (N - \sqrt{N}, N]$, so $k_3(u) < \frac{1}{N}$. Also, for a top endpoint q to have $y(q) = u$ for some $u \in [0, \sqrt{N})$ (e.g., $u = \frac{1}{2}\sqrt{N}$), the length of the corresponding vertical segment must not exceed u, while there is no such restriction on a top endpoint with y-coordinate in $[\sqrt{N}, N - \sqrt{N}]$. Therefore we have $k_1(u) < \frac{1}{N}$ and $k_2(u) \ge \frac{1}{N}$. To compute $\Pr\{V_j = 1\}$, we proceed in a different way.

Define P to be the conditional probability $\Pr\{s_j \cap l \neq \emptyset \mid y(l) = u, |s_j| = h\}$. By the fact that $y(b(s_j))$ is uniformly distributed over $[0, N - \sqrt{N}]$ and the analysis shown in Fig. 2, we have that $P = \frac{h}{N-\sqrt{N}}$ for $\sqrt{N} \le u \le N - \sqrt{N}$ (see Fig. 2(b)) and $P = \frac{\min\{u,h\}}{N-\sqrt{N}}$ for $0 \le u < \sqrt{N}$ (see Fig. 2(a)). As for $N - \sqrt{N} < u \le N$ (see Fig. 2(c)), note that $P = \frac{h-t}{N-\sqrt{N}}$ if $h \ge t$ and $P = 0$ if $h < t$, and thus $P = \frac{len}{N-\sqrt{N}}$ where $len = \max\{h - t, 0\} = \max\{h - u + N - \sqrt{N}, 0\}$. In summary, we have

$$P \stackrel{\text{def}}{=} \Pr\{s_j \cap l \neq \emptyset \mid y(l) = u, |s_j| = h\} = \begin{cases} \frac{\min\{u,h\}}{N-\sqrt{N}} \stackrel{\text{def}}{=} P_1 & \text{if } 0 \le u < \sqrt{N} \\ \frac{h}{N-\sqrt{N}} \stackrel{\text{def}}{=} P_2 & \text{if } \sqrt{N} \le u \le N - \sqrt{N} \\ \frac{\max\{h-u+N-\sqrt{N},0\}}{N-\sqrt{N}} \stackrel{\text{def}}{=} P_3 & \text{if } N - \sqrt{N} < u \le N \\ 0 & \text{else.} \end{cases}$$

Observe that $\min\{u, h\} \le h$ and that $\max\{h - u + N - \sqrt{N}, 0\} \le h$ for $u > N - \sqrt{N}$, so we have $P \le \frac{h}{N-\sqrt{N}}$ for $u \in [0, N]$. Recall that the probability density function of $|s_j|$ is $g(h)$ given before. Let $\widetilde{P}$ denote the probability $\Pr\{\text{vertical } s_j \cap l \neq \emptyset\}$. Then

$$\begin{aligned} \widetilde{P} &= \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} P \cdot k(u)\, g(h)\, du\, dh \\ &\le \int_0^{\sqrt{N}} \int_0^{N} \frac{h}{N - \sqrt{N}}\, k(u)\, \frac{1}{\sqrt{N}}\, du\, dh = \int_0^{\sqrt{N}} \frac{h}{N - \sqrt{N}}\, \frac{1}{\sqrt{N}} \left(\int_0^{N} k(u)\, du \right) dh = \frac{1}{2}\, \frac{1}{\sqrt{N} - 1}, \end{aligned}$$

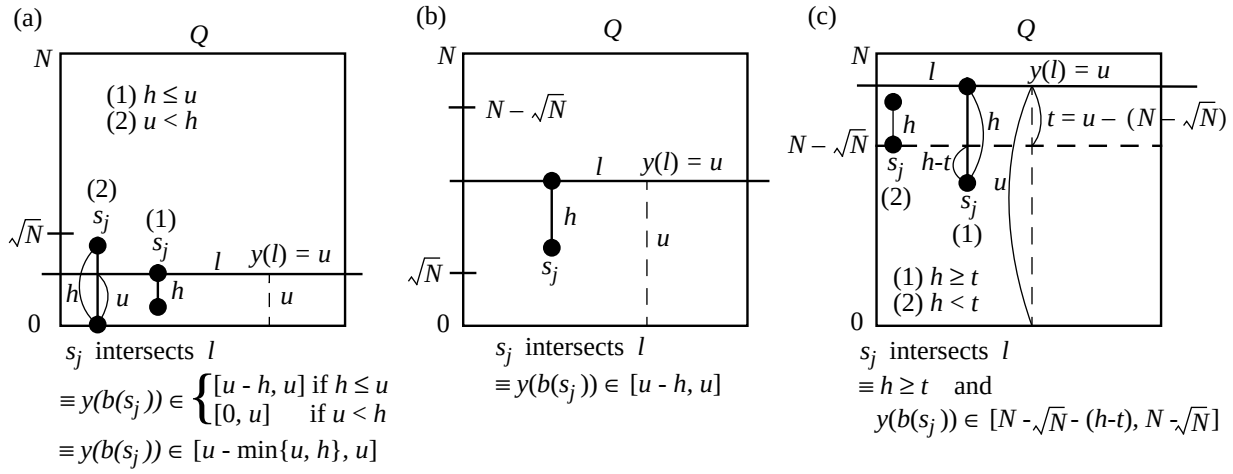

Figure 2: Computing $\Pr\{s_j \cap l \neq \emptyset \mid y(l) = u, |s_j| = h\}$ for `data-short`: (a) $0 \leq u < \sqrt{N}$; (b) $\sqrt{N} \leq u \leq N - \sqrt{N}$; (c) $N - \sqrt{N} < u \leq N$.

where the last equality follows from the fact that $\int_0^N k(u)du = 1$. Also,

$$\begin{aligned}
\widetilde{P} &= \int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} P \cdot k(u)\, g(h)\, dh\, du \\
&= \int_0^{\sqrt{N}}\int_{-\infty}^{+\infty} P_1 \cdot k_1(u)\, g(h)\, dh\, du + \int_{\sqrt{N}}^{N-\sqrt{N}}\int_{-\infty}^{+\infty} P_2 \cdot k_2(u)\, g(h)\, dh\, du \\
&\quad + \int_{N-\sqrt{N}}^{N}\int_{-\infty}^{+\infty} P_3 \cdot k_3(u)\, g(h)\, dh\, du \\
&\geq \int_{\sqrt{N}}^{N-\sqrt{N}}\int_0^{\sqrt{N}} \frac{h}{N-\sqrt{N}}\,\frac{1}{N}\,\frac{1}{\sqrt{N}}\, dh\, du = \frac{1}{2}\,\frac{1}{\sqrt{N}-1}\,\frac{\sqrt{N}-2}{\sqrt{N}},
\end{aligned}$$

where the inequality follows from that fact that the first and the third additive terms are both non-negative and that $k_2(u) \geq \frac{1}{N}$. Now we have lower and upper bounds for $\widetilde{P}$, and recall from equation (1) that $\Pr\{V_j = 1\} = \frac{1}{2}\cdot\widetilde{P}$. It follows that $E[V] = N \cdot E[V_j] \leq \frac{1}{4}\sqrt{N}(\frac{\sqrt{N}}{\sqrt{N}-1}) = \frac{1}{4}\sqrt{N} + \frac{1}{4} + \frac{1}{4(\sqrt{N}-1)}$ and also $E[V] \geq \frac{1}{4}\sqrt{N}(\frac{\sqrt{N}-2}{\sqrt{N}-1}) = \frac{1}{4}\sqrt{N} - \frac{1}{4} - \frac{1}{4(\sqrt{N}-1)}$, that is, $E[V] = \frac{1}{4}\sqrt{N} + O(1) \sim \frac{1}{4}\sqrt{N}$. It is interesting to see that $\frac{1}{4}\sqrt{N}$ can be interpreted as the product of the probability of a segment being vertical ($\frac{1}{2}$) and the average length of vertical segments ($\frac{1}{2}\sqrt{N}$). Figure 3 shows the actual values of the average number of vertical overlaps with respect to data size N, for all three data sets generated. We also compare for `data-short` the actual values and the analyzed values in Table 1, which shows that the observed values are indeed $\frac{1}{4}\sqrt{N}$.

N : # segments ($\times 10^3$)	250	500	1000	1500	2000	2500
Actual value	125.23	176.74	249.98	306.40	353.58	395.60
$\frac{1}{4}\sqrt{N}$ ($\sim E[V]$)	125	176.78	250	306.19	353.55	395.28

Table 1 The actual and analyzed values of the average number of vertical overlaps in data set `data-short`.

Program `gen-long` generates short as well as long segments while keeping the number of intersections small. For a horizontal segment, the length is assigned $\sqrt{N}$ (short segment); for a vertical

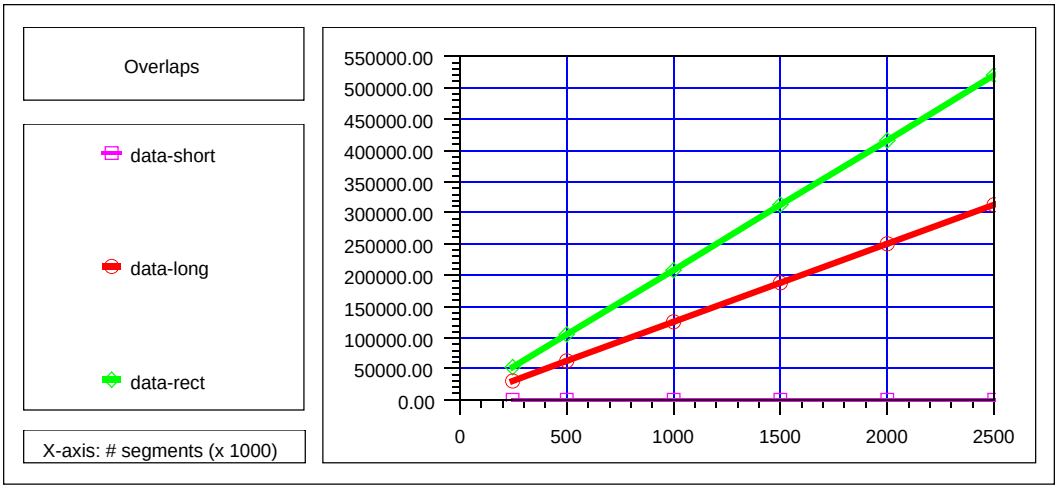

Figure 3: The actual values of the average number of vertical overlaps with respect to the number of segments in data sets `data-short`, `data-long` and `data-rect`.

segment, the program tosses a coin, giving length $\sqrt{N}$ (short segment) if the outcome is a head and N (long segment) otherwise. The horizontal and vertical short segments are randomly placed in the $N \times N$ sqaure Q in the same way as described in `gen-short`. As for vertical long segments, the bottom endpoints are placed randomly in an $N \times N$ square Q' whose lower-right corner coincides with the lower-left corner of Q, such that the distances from the bottom endpoints to the left and bottom sides of Q' are both uniformly distributed over $[0, N]$. Thus we have about $\frac{1}{2}N$ short horizontal segments, $\frac{1}{4}N$ short vertical segments, and $\frac{1}{4}N$ long vertical segments which cause no intersections. Using similar analysis methods as given before, we have that $E[K] = \frac{1}{8}(N-1)$ and $E[V] = \Theta(N)$; the latter bound shows that $E[V]$ is asymptotically proportional to the length of the *long* vertical segments. The observed K values of `data-long` are indeed $\frac{1}{8}N$ (see Fig. 1), and the observed values of the average number of vertical overlaps are also $\frac{1}{8}N$ (see Fig. 3).

In program `gen-rect`, we generate horizontal and vertical segments with lengths uniformly distributed over $[20, 60]$ and over $[0, 2N]$, respectively. The left endpoints of horizontal segments are randomly placed inside an $80N \times N$ rectangle R (with horizontal side length $80N$), such that the distances to the left and bottom sides of R are uniformly distributed over $[0, 80N]$ and over $[0, N]$, respectively. The vertical segments are placed as follows: in the x-direction, the distance between the left side of R and the *i-th vertical* segment is $(i-1) \times 160$, $i = 1, 2, \cdots$; in the y-direction, the distances from the bottom endpoints to the bottom side of R are uniformly distributed over $[0, N]$. It is easily seen that each horizontal segment can intersect at most one vertical segment, so that $K = O(N)$. Using similar analysis methods as given before, we have that $E[K] = \Theta(N)$ and also $E[V] = \Theta(N)$. Again the latter bound shows that $E[V]$ is asymptotically proportional to the average length of vertical segments. Figures 1 and 3 show that the actual K values are $\frac{1}{18}N$, and the actual values of the average number of vertical overlaps are $\frac{1}{4.8}N$.

3.2 Computing Environment and Performance Measures

We perform the experiments on a Sun Sparc-10 workstation, which is running under Solaris 2.4 and is a multi-user distributed system. The main memory size is 32Mb and one page is of size 4Kb. Our performance measures are running time, number of I/O operations performed (i.e., number of pages read and written by the process), and number of page faults occurred.

Notice that the running time is our ultimate concern. Unlike previous experimental work, the CPU time does not correctly reflect the performance of the algorithms we want to measure, since our main concern is the amount of time in which the CPU is sleeping waiting for the I/O or page faults. To overcome the difficulty, we perform all experiments by running the processes in the *real-time* class with the *highest priority* and measure the elapsed time. Also, the secondary memory used is the local disk so that the performance is not affected by the network file servers.

We are surprised to find that the system does not fully support performance statistic information. For example, it is claimed that user commands `time` and `timex` give CPU and elapsed times as well as numbers of I/O and page faults, etc., but it turns out that only the information regarding times are available. By using a system call in our program to retrieve the information in the `/proc` file system, we are able to obtain the number of page faults that require physical I/O's, yet the numbers of pages read and written by the process are still unavailable. Therefore, we also keep track of the numbers of times the `read` and `write` system calls are executed, where each time the size of the data being transferred is one page by our implementation.

We implement the algorithms so that the amount of main memory used can be parameterized. It is surprising that the main memory size available for use is typically much smaller than what we thought. When we run `Distribution` on `data-long` of 1.5×10^6 segments with various sizes of main memory used (see Fig. 4), in theory we would expect that using more main memory results in a better performance according to the I/O cost bound $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B} + \frac{K}{B})$, but the experiments show that using 4Mb gives the best performance (average running time 47.64 minutes), and using 20Mb gives a significantly worse performance (average running time 271.97 minutes)! Going from 1Mb to 4Mb, the number of I/O operations slightly decreases, and the number of page faults increases slightly yet negligibly; the net effect is to make the running time decrease slightly (see Fig. 4). Going from 4Mb to 20Mb, the number of I/O operations again decreases only slightly, but the number of page faults increases significantly, thus resulting in a much worse performance (see Fig. 4). This is actually a system issue. Using the `top` user command, we see that the "real" main memory size in the system configuration is only 26Mb rather than 32Mb and that processes (including their text, data, and stack portions) are never fully loaded into the main memory. The process loading behavior is decided by the OS and the user has no control over it. In the following, all the algorithms are running with the parameters of the main memory size set to 4Mb.

4 Analysis of the Experimental Results

Algorithms `Distribution`, `B-Tree`, `234-Tree`, and `234-Tree-Core` have been executed on data sets `data-short`, `data-long`, and `data-rect`, with data sizes ranging from 250 thousand segments to 2.5 million segments. While running times and numbers of page faults may differ between runs of the same example, the numbers of I/O operations are always the same. We run each example three times, and find that the variation among runs is at most 5%. More importantly, these differences among runs do not affect the performance ranking of the four algorithms. Figures 5–7 show the corresponding values of average running times, exact numbers of I/O operations, and average numbers of page faults of the four algorithms.

Our experimental results show that while the performance of the three variations of plane sweep depends heavily on the average number of vertical overlaps, the performance of distribution sweep is both steady and efficient. Also, distribution sweep does not require a large amount of main memory to perform well: using 4Mb is enough. We make more detailed observations as follows:

- `234-Tree-Core` performs the best for small input ($N = 250 \times 10^3$) in all three data sets (see Figs. 5–7), but as input size grows, the performance becomes considerably worse, and up to

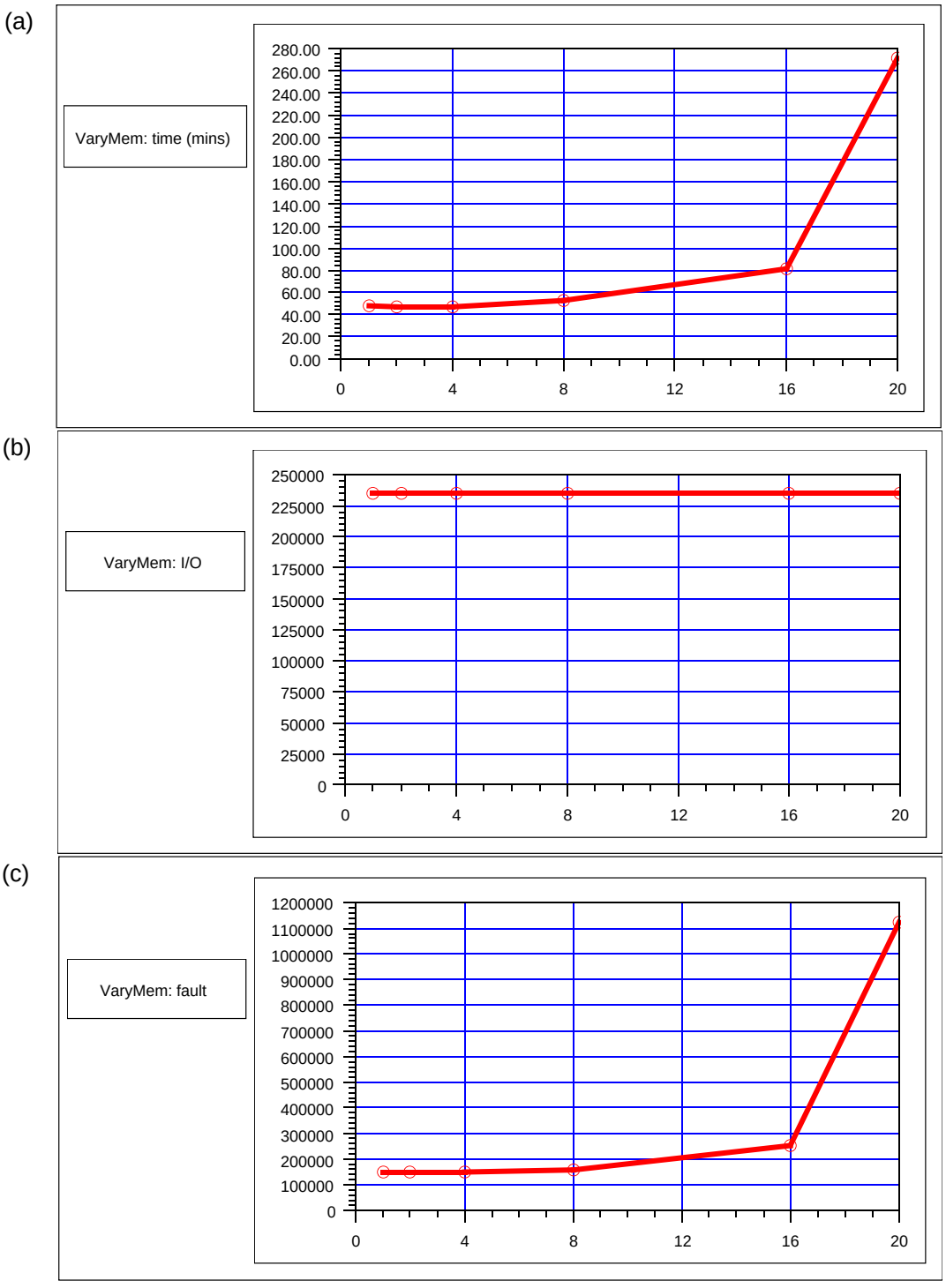

Figure 4: Running `Distribution` on data set `data-long` of 1.5×10^6 segments with various sizes of the main memory used: (a) average running times in minutes; (b) exact numbers of I/O operations; (c) average numbers of page faults.

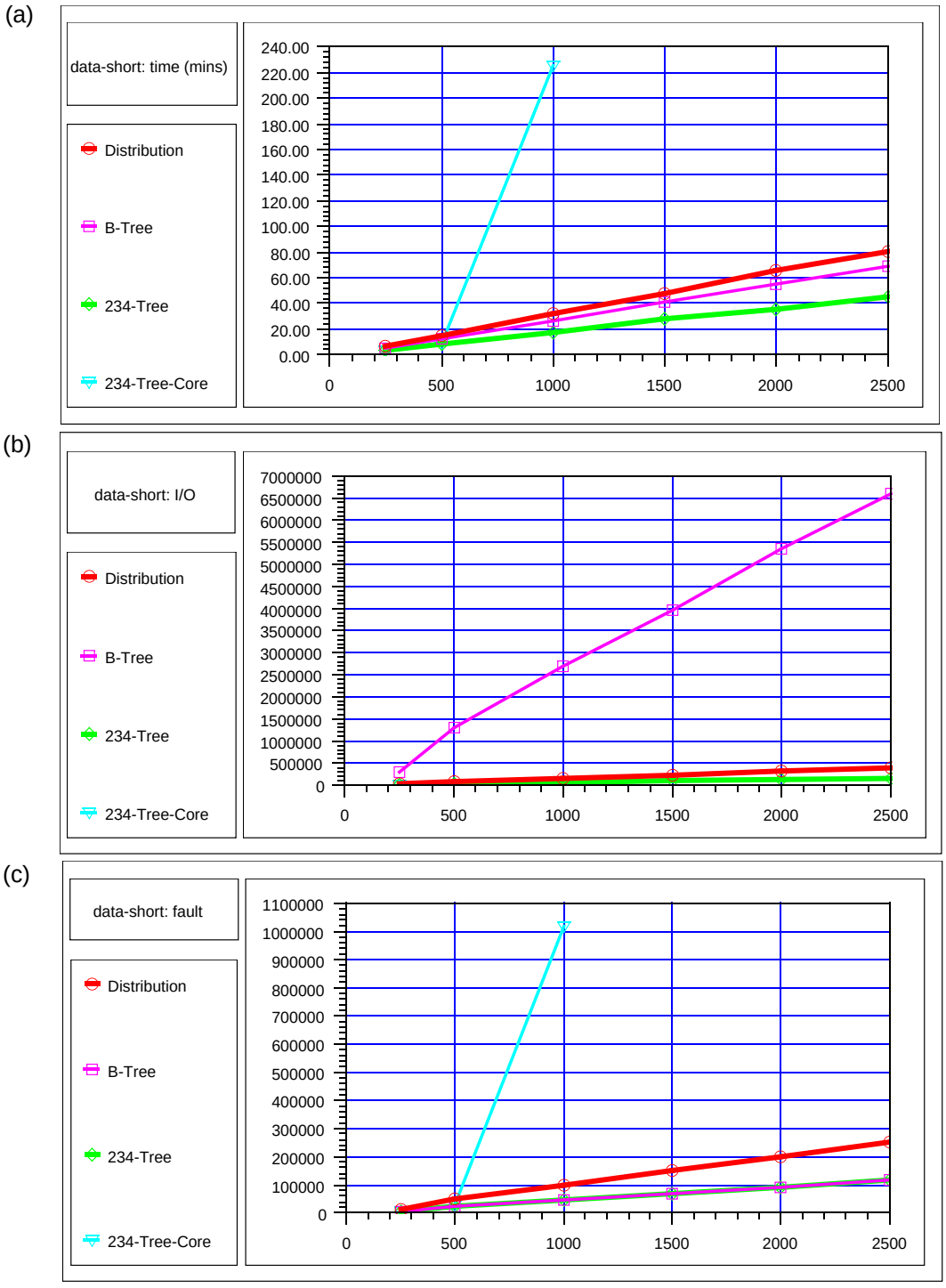

Figure 5: The results for the algorithms running on data set `data-short`: (a) average running times in minutes; (b) exact numbers of I/O operations; (c) average numbers of page faults. We run `234-Tree-Core` only up to $N = 10^6$ since at this point it already takes time much longer than the others even at $N = 2.5 \times 10^6$.

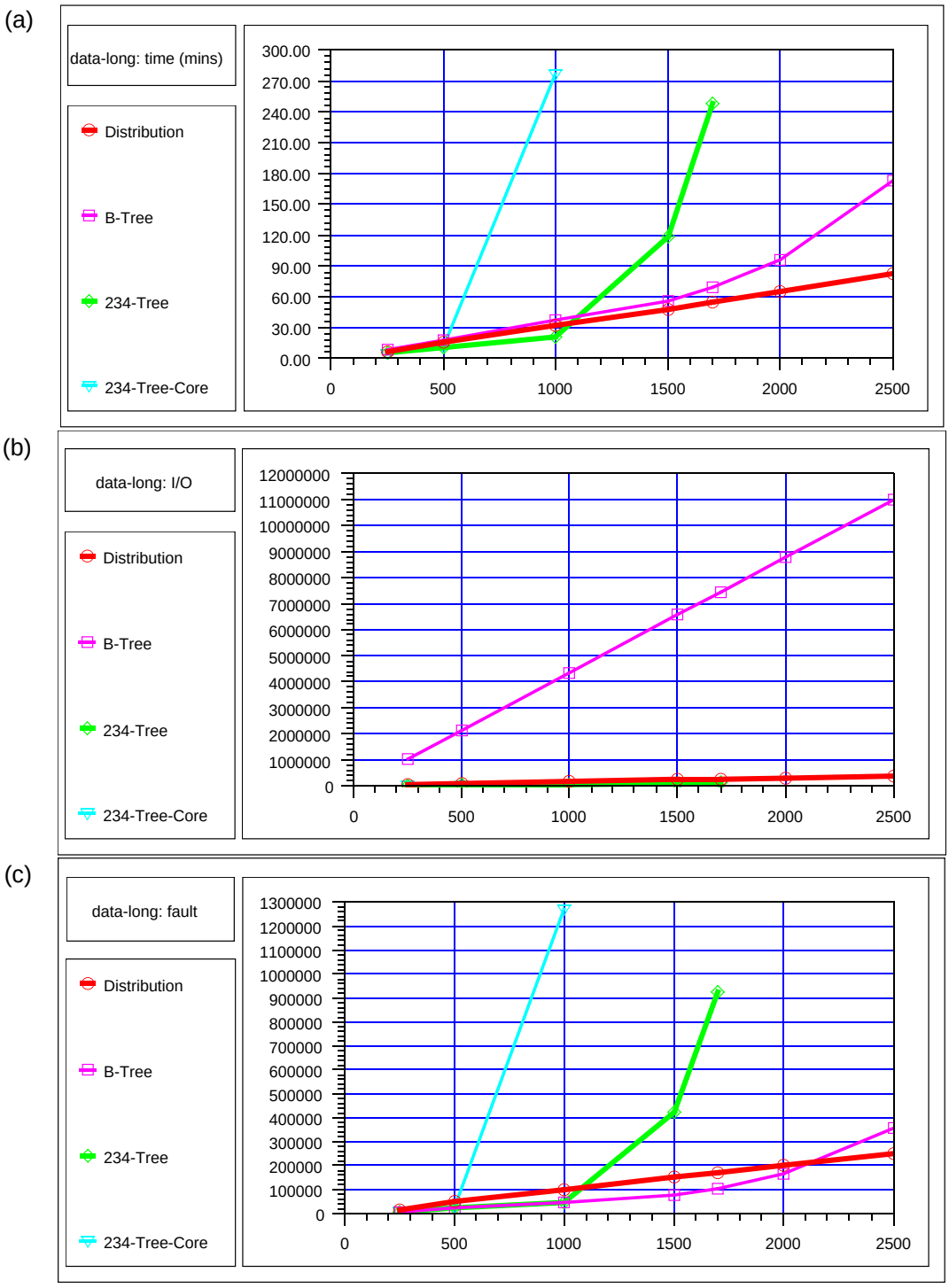

Figure 6: The results for the algorithms running on data set `data-long`: (a) average running times in minutes; (b) exact numbers of I/O operations; (c) average numbers of page faults. We run `234-Tree-Core` only up to $N = 10^6$ and `234-Tree` only up to $N = 1.7 \times 10^6$ since at these points they already take times much longer than the others even at $N = 2.5 \times 10^6$.

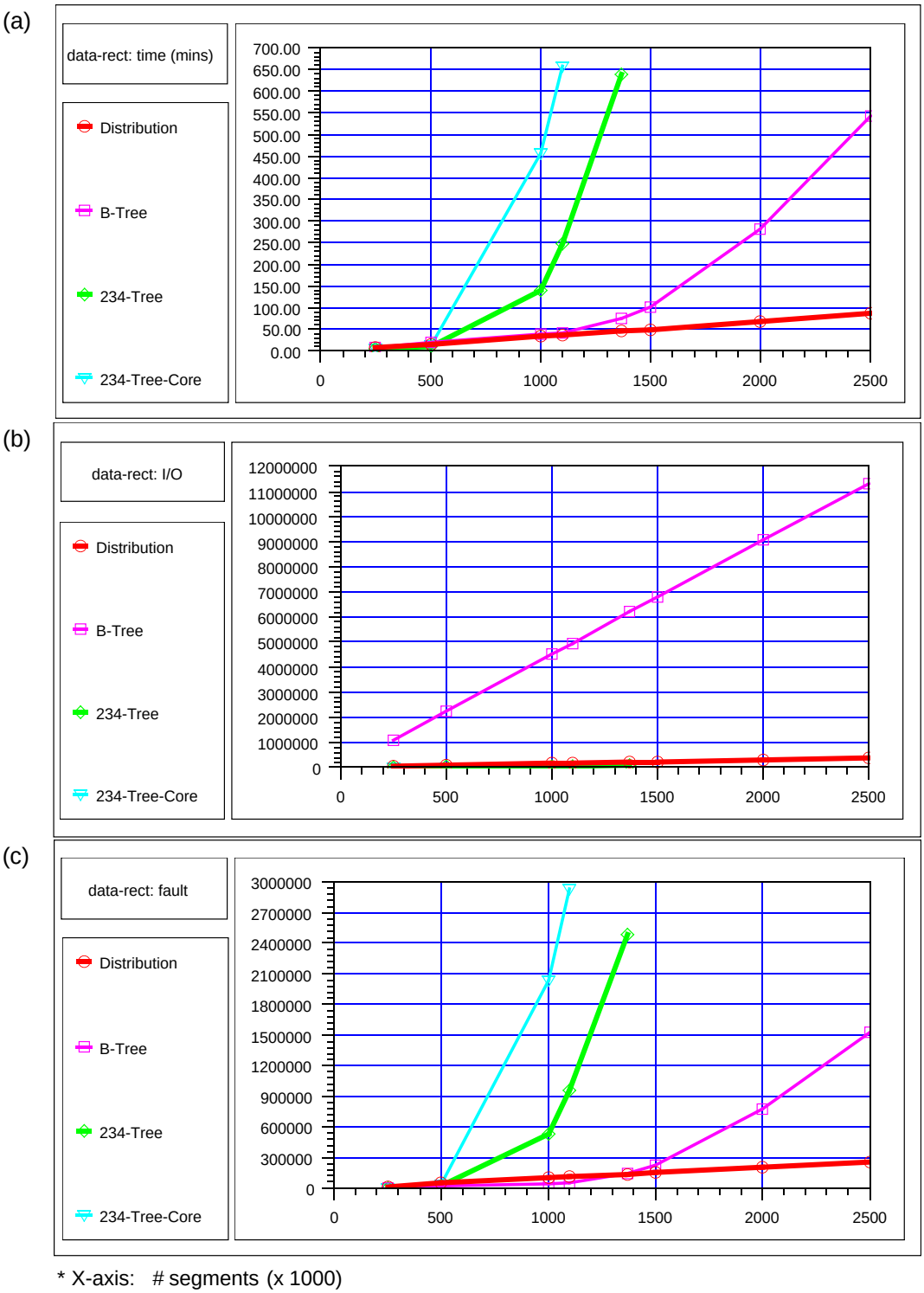

Figure 7: The results for the algorithms running on data set `data-rect`: (a) average running times in minutes; (b) exact numbers of I/O operations; (c) average numbers of page faults. We run `234-Tree-Core` only up to $N = 1.1 \times 10^6$ and `234-Tree` only up to $N = 1.37 \times 10^6$ since at these points they already take times much longer than the others even at $N = 2.5 \times 10^6$.

$N = 10^6$ its running times are already out of comparison.

- Consider data set `data-short` (see Fig. 5). Excluding `234-Tree-Core`, `234-Tree` always runs the fastest and `Distribution` always runs the slowliest. This can be explained by the small numbers of vertical overlaps which results in small tree sizes that still fit into the main memory. Also, `Distribution` performs two sortings, while all the others perform only one sorting.
- For data set `data-long` (see Fig. 6), `Distribution` runs much faster than all the others for $N \geq 1.5 \times 10^6$. `234-Tree`, following `234-Tree-Core` after $N \geq 10^6$, is out of comparison for its running times and numbers of page faults after $N \geq 1.7 \times 10^6$. The running times and numbers of I/O operations of `B-Tree` are still more or less linear, and are always worse than those of `Distribution`.
- For data set `data-rec` (see Fig. 7) with $N \geq 10^6$, `Distribution` performs the fastest, and the running times of the four algorithms differ significantly. For example, for $N = 1.37 \times 10^6$ and on average, `Distribution` runs for 45.29 minutes, `B-Tree` runs for 74.54 minutes, but `234-Tree` runs for more than 10.5 hours. Also, for $N = 2.5 \times 10^6$, `Distribution` always runs for less than 1.5 hours, but `B-Tree` always runs for more than 8.5 hours.
- For all three data sets, `234-Tree` and `234-Tree-Core` always have small numbers of I/O operations (see Figs. 5(b), 6(b), and 7(b)). This is because in the sweeping phase both of them only perform `read` for reading the input once and `write` for writing the output once, and all other I/O activities are page faults caused by the assumption of an infinite-size virtual memory. Also, `Distribution` always has much less I/O operations than `B-Tree` (see Figs. 5(b), 6(b), and 7(b)). Recall that the I/O cost bounds for `Distribution` and `B-Tree` are $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B} + \frac{K}{B})$ and $O(N \log_B \frac{N}{B} + \frac{K}{B})$, respectively. With the parameters of the main memory size set to 4Mb, the two logarithmic terms in these bounds are almost the same, and it is the $\frac{1}{B}$ term that makes the difference significant.
- Page faults seem to be more time-consuming than I/O operations (see Figs. 5–7). This is because the number of I/O operations is obtained by counting the number of times the `read` and `write` system calls are executed, and thus some of them might be executed for pages that are still residing in main memory (i.e., in the system buffer cache), while the number of page faults only counts for those that actually require physical I/O's. We hope that the actual number of I/O operations can be available by a better system support in the future.

5 Conclusion

We have presented an experimental study comparing the performance of four algorithms for the orthogonal segment intersection problem. The observed behavior of the algorithms shows that the performance of distribution sweep is both steady and efficient. Also, it does not require a large amount of main memory to perform well. In addition, it is shown that the performance resulting from the strategy of letting the OS handle page faults and not explicitly considering the I/O cost is very undesirable. We conclude that one should make a full control of the I/O behavior of the program to obtain a good performance guarantee.

Acknowledgement

I would like to thank Roberto Tamassia for a lot of stimulating discussions, and Tom Doeppner and Peter Galvin for providing me useful information about our computing systems.

References

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

[2] L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Proc. Workshop on Algorithms and Data Structures (to appear)*, 1995.

[3] L. Arge, D. E. Vengroff, and J. S. Vitter. External-memory algorithms for processing line segments in geographic information systems. Manuscript, 1995.

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

[5] J. L. Bentley. Experiments on traveling salesman heuristics. In *Proc. 1st ACM-SIAM Sympos. Discrete Algorithms*, pages 91–99, 1990.

[6] J. L. Bentley. K-d trees for semidynamic point sets. In *Proc. 6th Annu. ACM Sympos. Comput. Geom.*, pages 187–197, 1990.

[7] J. L. Bentley. Tools for experiments on algorithms. In *Proc. CMU 25th Anniversary Symp.*, 1990.

[8] J. L. Bentley. Fast algorithms for geometric traveling salesman problems. *ORSA J. Comput.*, 4(4):387–411, 1992.

[9] P. Callahan, M. T. Goodrich, and K. Ramaiyer. Topology B-trees and their applications. In *Proc. Workshop on Algorithms and Data Structures (to appear)*, 1995.

[10] 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.

[11] D. Comer. The ubiquitous B-tree. *ACM Comput. Surv.*, 11:121–137, 1979.

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

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

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

[15] T. H. Cormen, T. Sundquist, and L. F. Wisniewski. Asymptotically tight bounds for performing BMMC permutations on parallel disk systems. Technical Report PCS-TR94-223, Dartmouth College Dept. of Computer Science, July 1994.

[16] E. Feuerstein and A. Marchetti-Spaccamela. Memory paging for connectivity and path problems in graphs. In *Proc. Int. Symp. on Algorithms and Comp.*, 1993.

[17] 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.

[18] G. N. Frederickson. A data structure for dynamically maintaining rooted trees. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 175–184, 1993.

[19] M. T. Goodrich, M. H. Nodine, and J. S. Vitter. Blocking for external graph searching. In *Proc. ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Sys.*, pages 222–232, 1993.

[20] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational

geometry. In *IEEE Foundations of Comp. Sci.*, pages 714–723, 1993.

[21] P. C. Kanellakis, S. Ramaswamy, D. E. Vengroff, and J. S. Vitter. Indexing for data models with constraints and classes. In *Proc. ACM Symp. on Principles of Database Sys.*, pages 233–243, 1993.

[22] C. M. Kenyon-Mathieu and J. S. Vitter. The maximum size of dynamic data structures. *SIAM J. Comput.*, 20:807–823, 1991.

[23] M. H. Nodine and J. S. Vitter. Deterministic distribution sort in shared and distributed memory multiprocessors. In *Proc. 5th ACM Symp. on Parallel Algorithms and Architectures*, June 1993.

[24] 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*, January 1993.

[25] F. P. Preparata and M. I. Shamos. *Computational Geometry: an Introduction*. Springer-Verlag, New York, NY, 1985.

[26] S. Ramaswamy and S. Subramanian. Path caching: A technique for optimal external searching. In *Proc. ACM Symp. on Principles of Database Sys.*, pages 25–35, 1994.

[27] 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.

[28] J. D. Ullman and M. Yannakakis. The input/output complexity of transitive closure. *Annals of Mathematics and Artificial Intellegence*, 3:331–360, 1991.

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

[30] D. E. Vengroff and J. S. Vitter. I/O-efficient scientific computation using TPIE. Manuscript, 1995.

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

[32] B. Zhu. Further computational geometry in secondary memory. In *Proc. Int. Symp. on Algorithms and Computation*, 1994.