A Parallel Plane Sweep Algorithm for Multi-Core Systems

Mark McKenney & Tynan McGuire
Department of Computer Science, Texas State University
{mckenney,tynan_mcguire}@txstate.edu

ABSTRACT

A parallel version of the plane sweep algorithm targeted towards the small number of processing cores available on commonly available multi-core systems is presented. Experimental results show that the proposed algorithm significantly out-performs the serial plane sweep on such systems.

Categories and Subject Descriptors

H.2.8 [**DATABASE MANAGEMENT**]: Database Applications—*Spatial databases and GIS*;
I.3.5 [**COMPUTER GRAPHICS**]: Computational Geometry and Object Modeling—*Geometric algorithms*

General Terms

Algorithms

1. INTRODUCTION

The emergence of commonplace and cheap multi-core computers provides significant computing resources in low-cost systems; however, many algorithms are designed for single core processing and do not effectively utilize the resources provided by multi-core architectures. In the areas of spatial databases and geographic information systems in particular, spatial and geometric operations used for processing data are known to be computationally intensive and have the potential to benefit from multi-core architectures if spatial algorithms can be re-designed to take advantage of them. Therefore, it is worthwhile to investigate the parallelization of geographic and spatial operations for multi-core CPU architectures commonly available in today's computers.

Multi-core processors tend to have a small number of cores (typical current numbers range from two to four) as opposed to high end computer clusters. This characteristic poses a challenge to parallelizing algorithms in that introducing too much parallelism can degrade performance due to parallelization overhead and competition for resources. Instead, parallelism on a smaller scale is more appropriate.

ACM GIS '09 November 4 6, 2009. Seattle, WA, USA

In this paper, we provide a method to parallelize the plane sweep algorithm [6, 1, 2]. We choose to focus on the plane sweep algorithm since it forms the foundation of many common spatial operations, such as topological predicates and the classic set operations of intersection, union, and difference. In this paper, we orient our discussion of the plane sweep algorithm towards its use in spatial operations between pairs of regions; however, our method of parallelization applies to the plane sweep in general.

Attempts to parallelize the plane sweep algorithm have traditionally used methods of segmenting the input to an operation, either statically or dynamically, such that the operation will be performed over segments in parallel. However, proposed solutions introduce synchronization points or operate only over input arranged as orthogonal line segments [5, 3]. In [4], the plane sweep paradigm is avoided to achieve a highly parallel solution to the line segment arrangement problem, but this approach does not achieve good theoretical performance on a small number of processing cores.

We make the observation that the high running time of geometric operations is due to the large size of inputs to those operations. Therefore, we take the approach of *dynamically* segmenting the input to spatial operations, and performing the operation on multiple portions of the input in parallel without the need for synchronization between input portions. However, we cannot simply segment the input without violating invariants required by the plane sweep algorithm; therefore, one contribution of this paper is that we provide mechanisms to segment the input to a geometric operation while maintaining these invariants. Furthermore, it is undesirable to introduce new line segments into data by splitting existing line segments, since the running time of the plane sweep is proportional to the number of input line segments. A second contribution of this paper is the development of a post-processing mechanism in which such split segments are efficiently combined into single segments. Finally, a third contribution of this paper is to determine the level of parallelism that is generally necessary for the proposed algorithm to achieve good performance on architectures with specific numbers of processor cores. The result provided in this paper is a parallel plane sweep algorithm that is transparent to the user and is easily scalable to run on any number of available processor cores.

2. ALGORITHM

Our proposed algorithm is a variation on the plane sweep technique; for a detailed description of the plane sweep algorithm, see [6, 1, 2]. For this paper, we assume that a region

is represented as a collection of straight line segments that represent the region's boundary. The plane sweep algorithm is used to implement operations, such as intersecting two regions, by sweeping an imaginary line across the plane that stops at *event points*, which are the left and right endpoints of each line segment. Intersecting line segments are found by examining only line segments that intersect the sweep line (these are stored in the *active list*), and only line segments that are adjacent in the order in which they intersect the sweep line. Because both the left and right end points of a segment are visited by the sweep line, line segments are represented as *halfsegments* such that the left halfsegment represents the line segment for the left end point to be visited, and its *brother*, the right halfsegment, represents the line segment for the right end point to be visited. Halfsegments are ordered based on their *dominating point* (i.e., the left or right end point that they represent).

We have observed through testing that due to the upkeep required while segments are being added to and removed from the active list, the portion of the plane sweep algorithm involving the active list dominates execution time. This is especially true if numerical methods, such as the use of rational numbers instead of floating point point numbers, are used to eliminate numerical robustness problems. This observation leads to the hypothesis that a preprocessing and post-processing scan of the input data used to prepare it for parallelization will take very little time compared to a serial plane sweep algorithm; thus, the overhead introduced by these scans will be small compared to the gains they produce by allowing the segments involved in the active list to be processed in parallel.

HYPOTHESIS 1. *The parallelization gained by adding a preprocessing and post-processing step to the plane sweep algorithm will be sufficient enough to overcome the overhead introduced by the pre- and post-processing steps. This follows from the observation that most of the time spent in the plane sweep algorithm is in the portion of the algorithm involving active list management as compared to traversing the input.*

This hypothesis drives the overall design of our algorithm to follow three steps:

1. An initial scan slices the input data into p *strips*.
2. Plane sweep algorithms run over each strip in parallel.
3. The result of the plane sweeps are merged into a result.

2.1 Computing Strips

We require our algorithm to dynamically segment the input to a plane sweep algorithm into strips; thus, we do not enforce a static segmenting of space, such as a grid, but generate strips by examining properties of input regions. There are many possible strategies to segment the input that may take into account metrics such as density of halfsegments, total number of halfsegments in resulting strips, etc.; but attempting to utilize these metrics raised the time complexity of the overall algorithm, and these metrics are not necessarily indicative of underlying complexities such as the number of segment intersections that will occur in a given strip. Instead, we choose a simple method in which *split lines* are generated at roughly equal intervals across the portion of the input regions where the bounding rectangles of the regions intersect. These split lines are parallel to the sweep line and extend infinitely. If we are generating p strips, these split lines can be generated in $O(p)$ time. Once the split lines are known, the halfsegments forming the input to the operation are traversed and any halfsegment that crosses a split line is broken into two halfsegments such that no halfsegment will cross a split line. These breaks that are introduced into halfsegments are denoted *superficial breaks* since they are only required by the algorithm, are not present in the original inputs, and will be removed from the final result. Note that a slight complication will occur when removing superficial breaks if a halfsegment has an endpoint on a split line that is not a superficial break, or if a vertical segment lies on a split line. Therefore, split lines are slightly adjusted such that a split line does not intersect a halfsegment endpoint that is not part of a superficial break. This simple step can be carried out using a binary search on the input halfsegments and is key to maintaining a low time complexity for the algorithm. The result of this step is a list of halfsegments from the input and a list of halfsegments generated by superficial breaks that fall into each strip. Therefore, each strip can be used as input to a plane sweep algorithm.

The traditional plane sweep algorithm requires that the *plane sweep invariant* holds at all times. This invariant states that all segment intersections and event points are known in the area behind the sweep line. Obviously, this cannot hold when running multiple sweep lines over an input as we have proposed; however, the introduction of superficial splits along split lines effectively causes a sweep line's active list to clear at the split line. Therefore, no segment to the left of a split line can intersect with a segment to the right of a split line, and multiple sweep lines can start simultaneously at split lines. We summarize this as the *parallel sweep line invariant*:

DEFINITION 1. *The* parallel sweep line invariant *states that given a sweep line and its associated split, all intersections and event points are known between the sweep line and the beginning of the split.*

2.2 Performing the Plane Sweep

The plane sweep portion of the algorithm generates threads of execution that perform a plane sweep tailored to the specific geometric operation required. Each plane sweep is executed over a strip using the event schedules generated in the splitting phase of the algorithm. Note that because the input has been split such that each split is independent of the others, no communication is required between the plane sweeps operating on the various strips. Furthermore, because the basic plane sweep invariants hold on each strip, any version of the plane sweep algorithm may be used. This allows our technique to be easily integrated with existing algorithms and implementations.

2.3 Combining the Results

Once the plane sweeps for all of the splits have completed, the results of each plane sweep must be merged into a single result. Note that at this point the result of the required spatial operation has been computed, but it contains superficial breaks. In the case of operations such as topological predicates, this may be sufficient and no combination step is required. However, if the result of the spatial operation is to be stored or used in a chain of nested operations, then the superficial segment breaks must be removed.

Recall that in the splitting stage of the algorithm, we choose to split the input such that no existing line segments have an end point on a split line; however, this does not mean

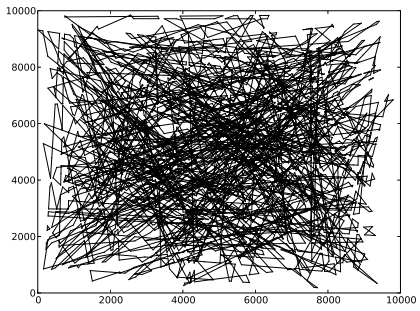

Figure 1: A synthetic region from pair 1 in Table 1.

	Description	Input 1 Half-segments	Input 2 Half-segments	Result Half-segments
1	Synthetic	92,436	98,874	350,086
2	Synthetic	96,496	97,238	355,558
3	Synthetic	102,298	92,492	357,534
4	Synthetic	378,218	411,068	1,455,014
5	Synthetic	379,096	394,998	1,428,486
6	Synthetic	412,738	404,388	1,513,822
7	Texas boundary	118,610	118,610	237,244
8	Florida boundary	40,652	40,652	81,312
9	California boundary	22,896	22,896	45,800

Table 1: Attributes of the pairs of regions used to test our algorithm. The regions in the first six pairs are randomly generated and have a high degree of overlap. The final three pairs consist of state outlines taken from U.S. Census TIGER data, translated in space, and then overlaid with themselves.

that no two line segments will intersect on a split boundary. The recombination of superficial breaks must consider such a situation and only recombine segments that are separated by superficial breaks. Note that a segment from the original input may have many superficial breaks introduced depending on the number of splits introduced.

In order to combine split segments, we make the observation that a segment that has a superficial split at point p will be the only segment with endpoint p in both adjacent strips around the split boundary. If two segments intersect at point p, then they must intersect such that two line segments sharing point p will exist in both strips surrounding the split boundary. This follows directly from the constraint that split lines are introduced only where they do not intersect any segment endpoints. Therefore, the identification and combination of superficial splits consists of finding a halfsegment h and its brother h_b containing a point p on a split line that is shared by no other segment, finding the halfsegment s and its brother s_b in the next adjacent strip that have endpoint p, and combining h and s and their brothers to form a single halfsegment t and its brother t_b without a superficial break. t and t_b are then included in the final output, and h, h_b, s, and s_b are discarded. We assume that the output sequence of halfsegments returned from the plane sweeps are sorted; therefore, given a halfsegment h, we can determine if its endpoint q lies on a split line because the split lines are known, and we can find all other halfsegments with endpoint q in both strips adjacent to the split line using binary searches. Therefore, this portion of the algorithm can be implemented in $O(pn \lg pn)$ time complexity for n line segments and p strips. In practice, we find that the number of segments involved in the $(\lg pn)$ term is small compared to the input size.

3. EXPERIMENTAL RESULTS

We have implemented the proposed algorithm in C++ as part of a software library providing spatial data types and operations. Both the serial and parallel versions of the plane sweep were implemented in the Bently-Ottmann style since we require our output halfsegments to be sorted for the post-processing phase of the algorithm. Our hardware testbed consists of a laptop containing a single 2-core processor, a desktop with a single 4-core processor, and a server with two 4-core processors for a total of 8 cores. Our experiments consist of running the serial and parallel plane sweep algorithms on each machine using a variety of input data and recording the time it takes to run each algorithm over a given input. Time is recorded as wall clock time provided through system library calls that are accurate to the millisecond. We test our algorithm by using it to calculate an overlay of two regions, since the overlay forms the basis of the classic geometric set operations and provides good insight into the general performance of all related operations.

The experiments use synthetic data, in the form of regions that are randomly generated, and geographic data, in the form of state boundaries. The synthetic regions contain a large number of halfsegments and large numbers of segments overlap when these regions are overlaid. State boundaries are translated in space, then overlaid with themselves; this results in situations where a small number of segments intersect in relation to the number of segments in the state boundaries. Table 1 summarizes the pairs of regions used in the experiments, along with the numbers of halfsegments in each input geometry and the resulting output geometry. Figure 1 depicts the generated region corresponding to the first region in the first pair in Table 1.

The results of our experiments are shown in Figure 2. The x-axis represents the number of strips that were used for the particular run of the experiment. Note that the time indicated for a single strip represents the time for the serial algorithm. For synthetic data, two columns of data are plotted for each number of strips. The columns respectively indicate the average running time for calculating the map overlay of the first three region pairs and the second three region pairs from Table 1. For the geographic data, each triple of columns associated with a number of strips respectively represents the running time of our algorithm with the California, Florida, and Texas data. The columns in the graphs are broken into three contributing factors for algorithm run time: (i) the preprocessing time required to create strips, (ii) the post-processing time required to re-combine the strips into a single output, and (iii) the overlay time.

The experimental results clearly show that for all numbers of processor cores, our parallel algorithm performs better than the serial version for all data sets. In every case, performance increases as more strips are used up to a certain point when performance begins to degrade. This degradation is caused by competition for resources on a limited number of processor cores, plus the greater overhead of pre-

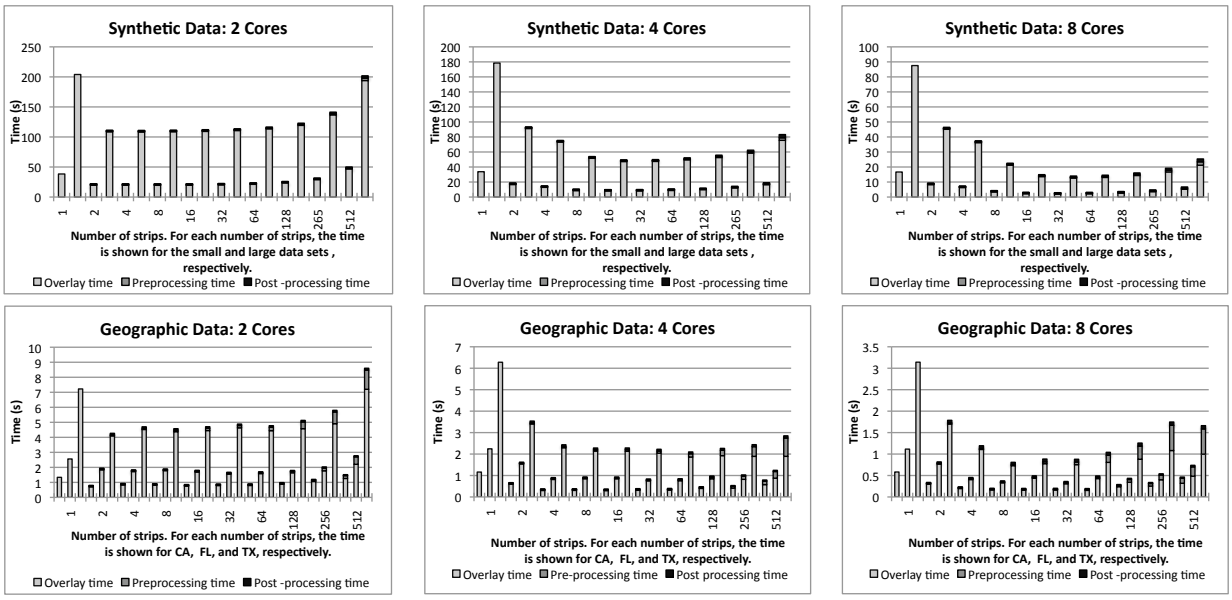

Figure 2: Running times of our algorithm over synthetic and geographic data on 2, 4, and 8 core architectures.

and post-processing for the larger number of strips. However, it should be noted that the pre- and post-processing times remain small for moderate numbers of processors.

One of the goals of this paper is to find an appropriate number of strips for an input that will yield good performance for a specific number of processing cores. Clearly, according to the resulting data, a large range of numbers of strips for a particular number of processor cores yields approximately the same results. This is a significant finding since it implies that our algorithm will not suffer greatly if the optimal number of strips for a particular architecture is not chosen. Furthermore, since the test cases contain large numbers of halfsegments, small numbers of halfsegments, high levels of segment intersections, and low levels of segment intersections, we see that the range of strips required for good performance is robust with respect to various input configurations. According to the experimental results, a general rule of 32 to 64 strips is a good choice in general.

On our eight core machine, the average of the best speedups for each input pair is 6.4 for the synthetic data and 3.3 for the geographic data. On the four core machine, the average of the best speedups for each input pair is 3.6 for the synthetic data and 2.9 for the geographic case. The average of the best speedups for the two core machine is 1.8 for the synthetic data and 1.7 for the geographic data. Figure 2 clearly shows that the efficiency of our parallelization degrades rapidly based on the number of strips introduced; however, our algorithm is relatively simple and can be easily used in conjunction with existing serial plane sweep algorithms. Thus, we achieve a significant speedup, especially on eight cores, using algorithms that are widely implemented.

Hypothesis 1 is clearly confirmed by our experiments. The performance of the preprocessing step degrades as more strips are introduced, but it does so slowly until around 256 strips.

4. CONCLUSIONS

In this paper, we have provided a preprocessing and post-processing technique to segment the input to a plane sweep algorithm allowing multiple plane sweep algorithms to run in parallel over portions of the input data. This technique does not require the special treatment of vertical or horizontal lines, and in practice, introduces very little overhead. Furthermore, our technique can be used in conjunction with any existing variant of the plane sweep algorithm. The pre- and post-processing time complexity of the algorithm is $O(pn \lg pn)$ for p strips and n halfsegments and the space complexity is $O(n)$. This is paired with the time complexity of the plane sweep implementation chosen. Although this is not a theoretical improvement in running time, we see speedups of up to 6.4 on eight core architectures in practice. Furthermore, we have found that a range of thirty two to sixty four strips will perform well on a variety of numbers of cores and input configurations.

5. REFERENCES

[1] J. Bentley and T. Ottmann. Algorithms for Reporting and Counting Geometric Intersections. *IEEE Trans. on Computers*, C-28:643–647, 1979.

[2] B. Chazelle and H. Edelsbrunner. An Optimal Algorithm for Intersecting Line Segments in the Plane. *J. ACM*, 39(1):1–54, 1992.

[3] M. Goodrich, J. Tsay, D. Vengroff, and J. Vitter. External-Memory Computational Geometry. In *Foundations of Computer Science*, pages 714–723, 1993.

[4] M. T. Goodrich. Intersecting Line Segments in Parallel with an Output-Sensitive Number of Processors. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 127–137, 1989.

[5] H.-P. Kriegel, T. Brinkhoff, and R. Schneider. Combination of Spatial Access Methods and Computational Geometry in Geographic Database Systems. In *International Symposium on Advances in Spatial Databases*, pages 5–21, 1991.

[6] M. Shamos and D. Hoey. Geometric Intersection Problems. In *Foundations of Computer Science*, 1976.