2019

Parallelization of Plane Sweep Based Voronoi Construction with Compiler Directives

Anmol Paudel

Jie Yang

Satish Puri

Parallelization of Plane Sweep Based Voronoi Construction with Compiler Directives

Anmol Paudel
MSCS Department, Marquette University
Jie Yang
MSCS Department, Marquette University
Satish Puri
MSCS Department, Marquette University

SECTION I. Introduction

Voronoi diagrams are extensively used in computational geometry to partition a plane into multiple regions where each region corresponds to and contain a site, and that site will be the closest site to all points in that region. Figure 1 shows a Voronoi diagram with a unique region for each site. Here is a mathematical definition [1] of a Voronoi region:

Definition 1. *Let P*: = {p_1, p_2,…, p_n} *be a set of n distinct points in the plane; these points are the sites. We define the Voronoi diagram of P as the subdivision of the plane into n cells, one for each site in P, with the property that a point q lies in the cell corresponding to a site* p_i, *if and only if dist*(*q*, p_i) < *dist*(*q*, p_j) *for each* $p_j \in$ *with* $j \neq i$.

where $dist(p,q) := \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$

There are different algorithms to construct Voronoi diagram with *n* sites as input. A brute-force algorithm constructs one region at a time. Since each region is the intersection of *n*-1 half planes, it takes $O(n\log n)$ time per region, thereby resulting in an $O(n^2\log n)$ time algorithm. An optimal algorithm has $O(n\log n)$ lower bound [2]. The planesweep algorithm that we consider here for parallelization is an optimal algorithm.

We are exploiting parallelism in the planesweep algorithm on a per event basis, however, the order of event processing is still sequential. This is because there is interdependence between the static and dynamic events generated by concurrent event processing. We have discovered that there is enough computation in an event itself to warrant performance improvement in a shared memory environment. These computations include intersection of neighboring arcs (w.r.t. an event) that is required to generate new events. This is the first work to identify and report the performance enhancement possible while concurrently maintaining the spatial data structures (beachline) on a per-event basis.

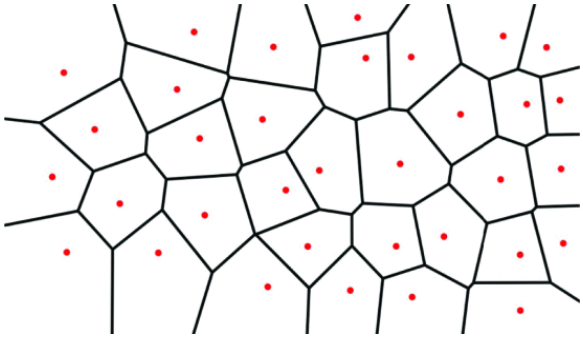

Fig. 1. Voronoi Diagram

[The dots in the figure are the sites and the lines are the edges of the a partitioned region. It can be observed that for any arbitrary point in the whole space, the closest site is the one inside the same region as it is.]

This paper is a part of our series of work focused on parallelizing existing spatial and computational geometry code using compiler directives. Our prior work was successful in the parallelization of the planesweep version of segment intersection and polygon intersection problems [3], [4], [5]. Existing literature focuses on theoretical work on parallel algorithms [6], [7]. There are other approaches of parallelization that use data decomposition [2], [8]. However, data decomposition algorithms require expensive merging steps ($O(n)$ time complexity) which are non-trivial to implement efficiently. Our work does not require explicit data decomposition.

This paper explores the concurrency available in processing each event in Voronoi diagram construction and uses directives to make an existing implementation of Fortune's algorithm faster with minimal efforts using compiler directives. OpenMP is an application programming interface which enables us to parallelize existing C, C++ or Fortran code by just adding compiler directives (#pragma) to it. The compiler takes the directives as hints for potential ways to inject parallelism in the sequential code. Directives based parallelization can be targeted at multicore CPUs, GPUs or a combination of both. Adding directives should not affect the correctness of the results produced, although the order in which results are produced might vary due to concurrency. Compiler directives based parallelization is more maintainable and performance portable to different multicore architectures and removes the hassle of having to change the parallelized code according to changes in

multicore architecture. Even though, OpenMP is good for regular parallelism, here we are trying to extract irregular and dynamic parallelism exposed by our modified Fortune's algorithm.

SECTION II. Fortune's Algorithm

Fortune's algorithm is a planesweep algorithm for computing Voronoi Diagram in $O(nlogn)$ time with $O(n)$ space [9]. Fortune presented a transformation that could be used to compute Voronoi diagrams with a sweepline technique. [9]

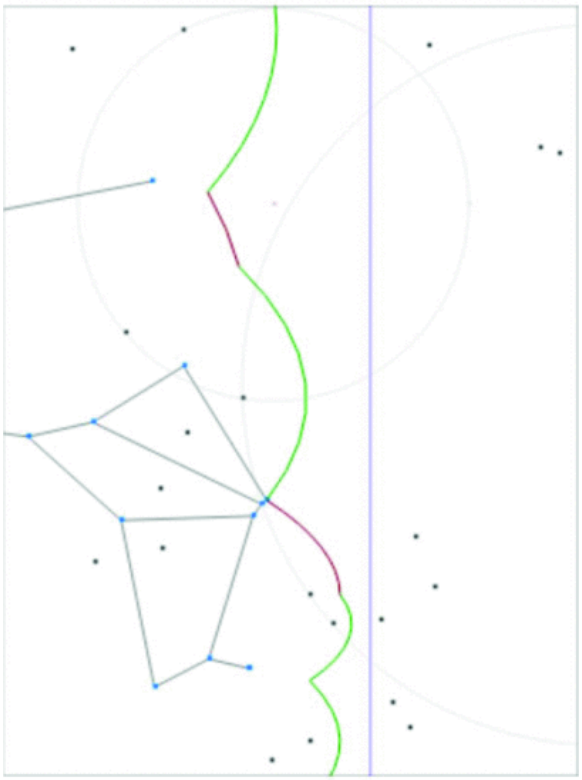

Fig. 2. A snapshot of the algorithm showing circle events, a vertical sweep line and beachline made up of arcs. (Best viewed in color)

In Figure 2, the dark grey dots are the site points. The blue dots are the Voronoi vertices and lines connecting the blue dots are the Voronoi edges. The vertical blue line is the sweepline. The green and red arcs form the beachline structure at the sweepline position. The light grey circles are the circle events. As the sweepline reaches a site point, an arc/parabola corresponding to it is created which will grow as the sweepline progresses and is clipped by neighbouring arcs or new arc ahead of it. The collection of active arcs is the beachline.

Algorithm 1 is a simplified algorithmic description of the implementation of Fortune's Algorithm. The focus of the description here is to show the flow of the algorithm so that the possibilities and limitations to a directive based approach can be explored. This algorithmic description here is necessary to understand the flow of execution and interdependencies among the variables that are key to any directive-based parallelization.

Algorithm 1 Fortune's Algorithm (Horizontal Sweep)

1: P ← load all points
2: Initialize a bounding box with offset
3: Initialize beachline B
    // B is of type arc

```
4: Initialize output O
        // O is a collection of edges of the partitioned regions
5: Initialize events priority queue
        // event with minimum x-coordinate is at the top
6: Sort P in ascending order by x-coordinate
7: for each p in P do
8:      while (events.top.x <= p.x) do
9:              ProcessEvent(events.deque())
10:     end while
11:     ProcessPoint(p)
12: end for
13: ProcessRemainingEvents()
14: FinishEdges()
```

In the event data structure, *x* is the maximum x-location a circle event can affect and it introduces a event processing there. So, x = p.x + radiusOfTheCircle.

Listing 1. Data Structure for Event

```
struct event {
        var x ;
        point p ;
        arc * a;
}
```

Algorithm 2 ProcessEvent(event e)

```
1: Input event e
2: if (e.valid) then
3:      Begin a new Segment s at e.x
4:      Remove e.a from beachline B
5:      Complete segments e.a.s0 and e.a.s1
6:      // Check circle events
                CheckCircleEvent(e.a.prev, e.x)
                CheckCircleEvent(e.a.next, e.x)
7:  end if
```

A. Parallelizing Fortune's Algorithm

We start by trying to find opportunities in the algorithm where compiler directives can be inserted for parallelization. The most obvious choice would be to parallelize the loops. Loop parallelization using directives is the easiest way to parallelize and usually has very less overheads. Furthermore, internal loops inside nested loops can also be parallelized.

In Algorithm 1, the for-loops and while loops cannot be directly parallelized due to interdependencies and memory side-effects. Algorithm 3 and Algorithm 2 can not run concurrently due to the interdependence of site events and the circle events.

In Algorithm 2, since entirety of its execution is based on a conditional, we need to determine the possibility of parallelizing this portion if it gets executed. Here, line 4 is dependent on line 3 because we need the segment *s*

to remove *e.a* from beachline B. However, excluding this, the two operations in line 5 and the two operations in line 6 can be parallelized to run concurrently. Completing the two segments in line 5 does not affect any other operations that could happen here concurrently. However the two circle events check in line 6 can lead to new events being added, but since these events are just added and not used elsewhere, we can put adding events part of the code inside critical sections and still parallelize line 6. So, in overall we can have five sections that run in parallel here - one section would comprise of lines 3 and 4, another two sections would comprise of completing each segment in line 5 and the other two sections would comprise of the two circle events checks in line 6.

Algorithm 3 ProcessPoint(point p)

```
1: Input point p
2: for arc i in beachline B do
3:         if intersects(p,i) then
4:                 Add new arc at p.x to beachline B
5:                 Connect new arc to prev and next segments of i
                           CheckCircleEvent(i, p.x)
                           CheckCircleEvent(i.prev, p.x)
                           CheckCircleEvent(i.next, p.x)
6:                 return
7:         end if
8: end for
9: arc I ← last arc in B
10: Insert segment between p and i
```

Algorithm 3 is even more complicated to parallelize because it has loops, conditionals inside loop and early exits inside those conditionals. An event is rendered invalid if the arc associated with that event is no longer in the beachline.

The outermost loop is searching for an arc corresponding to the new event. This is done by performing an exhaustive search looking for a single instance for which the search criterion is fulfilled. Then a series of operations is performed on the resultant instance if it was found. If a resultant instance was found then not only the loop is returned but the whole procedure is exited. We can start by separating the search and the execution of the result of the search. So, we parallelize the loop in step 2 to find an arc *i* which satisfies the if-condition and remove the execution part below. One problem here is that if such an arc is found by sequential iteration early on, parallelizing it might just give us unessential overhead. To remedy this, we will convert this search loop into a chunked iterative exhaustive search loop by providing hints to the compiler that there might be a loop cancellation before each chunked iteration. This transformation makes it suitable for utilizing OpenMP parallel loop cancellation feature as shown line 5 and line 6 of Algorithm 4.

Concurrent Processing of Circle Events

Another problem with Algorithm 3 is that, a sequential search would have terminated after finding the first instance for which the search criteria would have been satisfied but during a concurrent chunked iteration, there might be multiple instances for which the search criteria has been satisfied. For correctness with regards to the sequential code, we can use a minimum reduction to make certain that the first instance is reported. At this point we will either have an arc *i* that satisfies the conditional or not and the loop will be exited but the procedure will not have been terminated. We can put this conditional of whether an arc *i* has been found in an if-statement with its else-part as lines 10-11. If an arc *i* has been found then we can execute the lines 4-6 with *i* and if not then we execute lines 10-11. This removes any early procedure terminating conditions from Algorithm

3. Then lines 4-6 that has been moved out of the loop and put inside this new conditional statement can now be explored for further parallelism. Lines 4-5 need to be executed sequentially because line 5 is dependent on the arc created in line 4. However, as shown in Algorithm 4, the three parts of line 6 can be parallelized to run concurrently even along with lines 4-5. Again, here the circle events check can lead to new events being added, but since these events are just added and not used elsewhere, we can put adding events part of the code inside critical sections and still parallelize. However, we will not be able to parallelize lines 10-11 of Algorithm 3 because its execution needs to be sequential. So, in this portion we are able to parallelize the search phase and lines 4-6 after they have been moved outside. As shown by Algorithm 4, Lines 4-6 from Algorithm 3 will have four sections - first section would comprise of lines 4-5 and the other three sections would comprise each of the three parts of line 6.

SECTION III. Results

An OpenMP implementation of code was created using the analysis in section II-A and executed on data with varying number of sites. The skeleton for the sequential C++ code used was inspired by the work of Matt Brubeck [10]. The machine used to run the OpenMP code has the Intel Xeon E5-2695 multi-core CPU with 45MB cache and base frequency of 2.10GHz.

TABLE I Timings of Running the Code in Sequential and With OpenMP

Sites	Sequential	OpenMP	SpeedUp
2k	0.456s	0.165s	2.761
4k	0.758s	0.419s	1.809
8k	2.06s	0.995s	2.070
16k	6.496s	2.748s	2.364
32k	13.748s	5.162s	2.663
64k	38.847s	18.029s	2.155
128k	84.396s	39.305s	2.147

Figure 3 shows the execution time for different number of site events. Even with the overhead of parallelization, the OpenMP version beats its sequential counterpart. We can see from Table I that we get almost above 2x speedup using upto four threads. The distribution of points affects the runtime of our algorithm and we have observed that having some types of distribution of points improves the performance of our algorithm. The speedup varies for different number of sites because the time taken to search for an arc corresponding to the event being processed is variable. In Algorithm 4, there are two code blocks which have been parallelized using OpenMP. There is a sequential dependency between block 1 (lines 2-7) and block 2 (lines 8-18). Even though the for-loop is highly parallelizable, the second block with OpenMP sections can only use few threads. In the worst case scenario, the execution time for block 1 depends on the number of active arcs in the beachline but in average case, the intersection test (line 3) can finish much earlier. We have found that beyond four threads there is a degradation in efficiency.

Algorithm 4 ProcessPoint(point p) with directives

1: Input point *p*, initialize bool doesIntersect = False

#pragma omp parallel for num_threads(threadCount)
2: **for** arc *i* in beachline B **do**
J ← index of arc *i* in beachline B
3: **if** intersects(*p*,*i*) **then**

```
4:                  ind = j
5:                  doesIntersect = True
                    #pragma omp cancel for
6:          end if
            #pragma omp cancellation point for
7: end for

8: if (doesIntersect == True) then
9:          arc l ← B[ind]
            #pragma omp parallel sections
            {
                    #pragma omp section
                    {
10:                 Add new arc at p.x to beachline B
11:                 Connect new arc to prev and next segments of i
                    }
                    #pragma omp section
12:                 CheckCircleEvent(i, p.x)
                    #pragma omp section
13:                 CheckCircleEvent(i.prev, p.x)
                    #pragma omp section
14:                 CheckCircleEvent(i.next, p.x)
            }
15: else
16:         arc l ← last arc in B
17:         Insert segment between p and i
18: end if
```

SECTION IV. Conclusion

Our experiments and design space exploration in directives-based parallelization of Fortune's algorithm has yielded a shared memory implementation that gives around 2x speedup compared to the sequential version. A four threaded parallelization is extremely useful for applications that run on personal devices with quad-core processors or on cloud instances where the most common instance of compute nodes usually has four cores. We have experimentally demonstrated a novel way of extracting irregular and dynamic parallelism inherent at each event of the algorithm. Moreover, our new method decreases the run-time of the algorithm on data sets of different size.

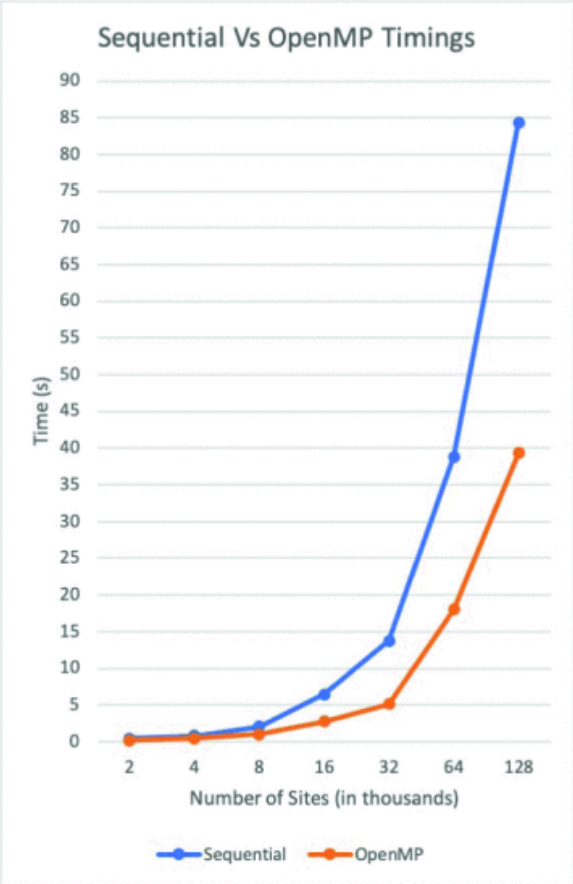

Fig. 3. Sequential vs OpenMP timings

ACKNOWLEDGEMENTS

This work is partly supported by the National Science Foundation Grant No. 1756000.

References

1. K. Wong and H. A. Muller, "An efficient implementation of fortune's plane-sweep algorithm for voronoi diagrams", 1991.
2. F. P. Preparata and M. I. Shamos, Computational geometry: an introduction, Springer Science & Business Media, 2012.
3. A. Paudel and S. Puri, "Openacc based gpu parallelization of plane sweep algorithm for geometric intersection", *Fifth Workshop on Accelerator Programming Using Directives co-located with the International Conference for High Performance Computing Networking Storage and Analysis (SC18)*, 2018.
4. S. Puri and S. K. Prasad, "Output-sensitive parallel algorithm for polygon clipping", *Parallel Processing (ICPP) 2014 43rd International Conference on*, pp. 241-250, 2014.
5. S. Puri, A. Paudel and S. K. Prasad, "MPI-vector-IO: Parallel I/O and Partitioning for Geospatial Vector Data", *Proceedings of the 47th International Conference on Parallel Processing ICPP*, pp. 13, 2018.
6. S. G. Akl and K. A. Lyons, Parallel computational geometry, Prentice-Hall, Inc., 1993.
7. M. T. Goodrich, M. R. Ghouse and J. Bright, "Sweep methods for parallel computational geometry", *Algorithmica*, vol. 15, no. 2, pp. 126-153, 1996.
8. M. d. Berg, O. Cheong, M. v. Kreveld and M. Overmars, Computational geometry: algorithms and applications, Springer-Verlag TELOS, 2008.

9. S. Fortune, "A sweepline algorithm for voronoi diagrams", *Algorithmica*, vol. 2, pp. 153-174, Nov 1987.
10. 2002, [online] Available: https://www.cs.hmc.edu/mbrubeck-/voronoi.html.