Parallelizing the Greedy Triangulation

Eliza Shoemaker, Randy Shoemaker

April 2019

1 Abstract

A greedy algorithm takes a set of points in the plane and returns a triangulation of the point set. The triangulation is built by adding the smallest line segment between points that does not intersect any line previously in the triangulation. The greedy triangulation gives an approximation of the Minimum Weight Triangulation (MWT), an NP-Hard problem. We present serial and parallel implementations of the greedy triangulation using the following approach: once a line is added to the triangulation all intersecting lines are removed from consideration. This process is repeated until a triangulation is obtained. We present and analyze experimental wall-time data for the serial and parallel implementations. We show that the parallel version has strong and weak scaling properties and that this algorithm benefits greatly from parallelism.

2 Background

A greedy algorithm takes a set of points in the plane and returns a triangulation of the point set. The triangulation is built by adding the smallest line segment between points that does not intersect any line previously in the triangulation. We present serial and parallel implementations of greedy triangulation using the following approach: once a line is added to the triangulation all intersecting lines are removed from consideration. This process is repeated until a triangulation is obtained. Pseudo-code for the serial version of our implementation is shown on page 3. If there are n points in the point set there are $\binom{n}{2} = \frac{n(n-1)}{2}$ lines between all points. Thus it would seem that the above approach has a worst case complexity of $O(n^4)$ since we may have to check every line against all other lines. However, it is known that this method has a worst case complexity of $O(n^3)$[2].

The greedy triangulation has been an area of research for several decades. It has been studied because it is known to give an approximation of the Minimum Weight Triangulation (MWT) problem. The MWT seeks to produce the triangulation of a point set with minimum weight. In this context the weight of a triangulation is the sum of the lengths of the line segments comprising it.

MWT is known to be NP-Hard [6], so approximations for it are desirable. The greedy triangulation is known to give a $\sqrt{n}$-approximation of the MWT [4].

Dickerson et. al. developed an $O(n \log n)$ algorithm to compute the greedy triangulation [1]. This approach requires the point set to be uniformly distributed within a convex hull. Drysdale et. al. gave an improved $O(n)$ algorithm which also requires the input set to be uniformly distributed in a convex hull [2]. It has also been shown that given the Delaunay triangulation the greedy triangulation can be computed in linear time [5]. The approach in the algorithm on page 3 does not require the point set to be uniformly distributed nor does it require computing the Delaunay triangulation. This approach does not assume or exploit any structure of the point set. This allows the algorithm to accept more versatile inputs, it is for this reason that we chose to parallelize this algorithm.

Parallel implementations of the greedy algorithm exist. Jansson developed a parallel version which runs in $O(n)$ on $O(n^4)$ processors [3]. For large point sets this becomes impractical. The parallel version presented in this paper is suitable for larger point sets and does not require such a large number of processors. In the following section we present the both the serial and parallel versions of our greedy triangulation algorithm.

3 Methods

Our parallel version of the greedy triangulation was created by modifying the serial version in the algorithm on 3. The serial algorithm consists of three phases: generation, sort, and triangulate. During the generation phase the $\binom{n}{2}$ possible line segments are generated, where n is the number of points in the point set. During the sort phase the lines are sorted in ascending order. In our implementation we used `qsort` from the `c` library to carry out the sort. During the triangulate phase the triangulation is built by successively adding the smallest line and removing all lines that intersect with it. After all intersecting lines are removed the new smallest line is selected and the process repeats. After each line has either been removed or added to the triangulation the algorithm terminates and returns the triangulation.

The parallel algorithm on page 4 can also be divided into the same phases as the serial version. To achieve parallelization we had to make some modifications. The generation phase is the same for both the serial and parallel versions except in the parallel version the generated lines are distributed to all processes. A parallel version of this phase was created but experimentation showed that it was slower than the serial version. After the lines are generated they are distributed to each process. Each process then carries out the sort phase in parallel. Once each process has sorted their local array of lines the triangulation phase begins. During the triangulation each process finds its smallest line and global communication is used so that each process has a list of the smallest line from each process. Each process then selects the smallest line. The ROOT process adds the smallest to the triangulation and the process that has the smallest line

removes it from its list of lines. At this point each process removes the lines that intersect the smallest line from its list. As in the serial version this is repeated until each line either belongs to the triangulation or has been removed. The algorithm then returns the triangulation and terminates.

Algorithm 1: Serial Greedily Triangulation Algorithm

```
   Input: Planar Point Set P
   Output: Triangulation T
 1 lines ← ∅
 2 n ← |P|
 3 line_index ← 0
   // Phase 1: generate all lines
 4 for i ← 0 to n do
 5     for j ← i + 1 to n do
 6         lines[index] ← the line from P[i] to p[j]
 7         line_index ← line_index + 1
   // Phase 2: sort the lines
 8 Sort the contents of lines in ascending order
   // Phase 3: greedily build the triangulation
 9 T ← ∅
10 unknowns ← |lines| // Number of lines with unknown status
11
12 while unknown > 0 do
13     l* ← lines[0]
14     T ← T ∪ {l*} // Add smallest line to triangulation
15
16     lines ← lines − l* // Remove the smallest line from lines
17
18     unknowns ← unknowns − 1
       // Remove all lines that intersect with l*
19     for l in lines do
20         if l intersects l* then
21             lines ← lines − l // Remove the intersecting line
22
23             unknowns ← unknowns − 1
24 return T
```

Algorithm 2: Parallel Greedily Triangulation Algorithm

Input: Planar Point Set P
Output: Triangulation T

$lines \leftarrow \emptyset$
$n \leftarrow |P|$
$line\_index \leftarrow 0$
// Phase 1: generate all lines
if *process is ROOT* **then**
    // Only generate *lines* on the *ROOT* process
    for $i \leftarrow 0$ ***to*** n **do**
        for $j \leftarrow i + 1$ ***to*** n **do**
            $lines[index] \leftarrow$ the line from $P[i]$ to $p[j]$
            $line\_index \leftarrow line\_index + 1$
Distribute *lines* to each process, initialize *local_lines*
// Phase 2: sort the lines (in parallel)
Sort the contents of *local_lines* on each process in ascending order
// Phase 3: greedily build the triangulation (in parallel)
$T \leftarrow \emptyset$
$unknowns\_local \leftarrow |lines\_local|$ // Number of lines with unknown status

while $unknowns\_local > 0$ **do**
    $l\_local^* \leftarrow lines\_local[0]$
    $small\_lines \leftarrow AllGather(l\_local^*)$ // Share smallest line

    $l^* \leftarrow$ smallest line in *small_lines*
    if *process is ROOT* **then**
        $T \leftarrow T \cup \{l^*\}$ // Add smallest line to triangulation on *ROOT*

    if l^* *in lines_local* **then**
        // Remove the smallest line from the process's *lines_local* and update its *unknowns_local*
        $lines\_local \leftarrow lines\_local - l^*$
        $unknowns\_local \leftarrow unknowns\_local - 1$
    // Remove all lines that intersect with l^*
    for l ***in*** *lines_local* **do**
        if l *intersects* l^* **then**
            $lines\_local \leftarrow lines\_local - l$ // Remove the intersecting line

            $unknowns\_local \leftarrow unknowns\_local - 1$
return T

The algorithms were implemented in the programming language `C` and the Message Passing Interface (MPI) was used to to implement the parallel version. The sorting phase was implemented in both versions by using `qsort` function from the `C` library.

4 Experiments

Experimental results for the serial and cluster were conducted on the cluster at James Madison University. This cluster has 16 nodes with each containing 8 processors. We tested both versions using varying sizes of point sets. Because there are on the order of $O(n^2)$ lines for n points each point set is 707 points times some multiple of $\sqrt{2}$. This is because multiplying the number of points by $\sqrt{2}$ doubles the input size, which is the number of lines. 707 points was used as a base line because smaller numbers of points yields wall-times which are small enough to be significantly affected by noise on the cluster. The number of points used for our experiments are as follows: 707, 1000, 1414, 2000, 2828, and 4000.

To ensure our analysis was robust we experimented with varying numbers of processes when testing the parallel version. Our experiments used the following numbers of processes: 1, 2, 4, 8, 16, 32, and 64. Powers of 2 were used so we could analyze the behavior to detect both strong and weak scaling. We have included tables specifying our results. These tables include the wall-times of the three phases where each data point is the smallest value observed for that particular entry. The values for the generation phase include the time it takes to distribute the points from the *ROOT* to all other processes. In the following section we analyze the performance of the parallel version and the serial version.

Experimental results for the generation phase.

Points	Serial	1 P	2 P	4 P	8 P	16 P	32 P	64 P
707	0.0113	0.0522	0.0312	0.0316	0.0193	0.0561	0.0762	0.1097
1000	0.0218	0.1014	0.0602	0.0518	0.0382	0.1109	0.1492	0.1784
1414	0.0429	0.2025	0.1176	0.1136	0.0730	0.2200	0.2955	0.3391
2000	0.0843	0.4433	0.2325	0.2007	0.1938	0.4715	0.5908	0.6798
2828	0.1673	0.7973	0.5070	0.4265	0.3774	0.8694	1.1742	1.3334
4000	0.3330	1.6651	1.1968	0.7969	0.6144	1.7375	2.3557	2.6662

Experimental results for the sort phase.

Points	Serial	1 P	2 P	4 P	8 P	16 P	32 P	64 P
707	0.0527	0.0538	0.0266	0.0200	0.0110	0.0053	0.0026	0.0014
1000	0.1087	0.1114	0.0542	0.0415	0.0229	0.0110	0.0054	0.0028
1414	0.2254	0.2668	0.1116	0.0606	0.0329	0.0232	0.0111	0.0054
2000	0.4675	0.5109	0.2329	0.1786	0.0962	0.0473	0.0230	0.0110
2828	0.9667	0.9868	0.4929	0.2660	0.2080	0.0997	0.0479	0.0232
4000	2.0025	2.0816	1.2922	0.6399	0.4286	0.2075	0.0898	0.0483

Experimental results for the triangulation phase.

Points	Serial	1 P	2 P	4 P	8 P	16 P	32 P	64 P
707	8.6674	9.3801	4.6939	2.8247	1.7278	1.2628	1.0404	1.2657
1000	26.493	34.767	18.436	10.754	6.1115	3.1537	2.3057	2.3979
1414	93.683	139.72	73.512	40.841	24.744	10.667	5.8319	5.3011
2000	345.38	511.96	259.15	150.19	116.75	43.747	21.820	12.742
2828	1187.1	1841.7	956.47	541.69	386.83	179.37	92.878	46.203
4000	4311.5	6878.5	4347.3	2192.0	1301.9	661.94	333.28	177.73

5 Results

With the included data we can make the following observations about the thee phases of the parallel version:

The cost of distributing the lines adds overhead to the generation phase.

In general the sort phase scales both strongly and weakly.

The triangulation phase scales strongly and the speed up is significant.

Since the lines must be distributed during the generation phase and the generation phase takes place serially on the *ROOT* the wall-time of the parallel version's generation phase is slower. This is because the *ROOT* must communicate with all other processes. When compared to the wall-time of the entire program this increase is dwarfed by the benefits of parallelizing the triangulation phase. In the table of values for the generation phase it can be seen that for a given input the wall-times generally decrease as the number of processes is increased from 1 to 2 and from 2 to 4. This is the case because the processes are all running on the same node when the number of processes is less than 8. When the number of processes increases to 8 the wall-time increases because the processes are running on more than one node and thus communication is more costly. The overhead costs of the generation phase are offset by the benefits of parallelism for the sort and triangulation phases.

The sort phase scales strongly and weakly. Notice that, in general, if the number of points in point set is fixed and the number of processes is doubled the wall-time is halved. This is why we are justified in asserting that the sorting phase scales strongly. Recall that in order to double the input size we must scale the number of points by $\sqrt{2}$. Notice that as the input size is doubled and the processes are doubled too that the wall-time remains roughly constant. This means that the sort phase scales weakly. Even though the sorting phase has nice scaling properties, the benefits to the algorithm as a whole are small because the proportion of the wall-time occupied by the sorting phase is small. It is the triangulation phase that is the most costly and where parallelism has the greatest benefit.

Most of the benefits to the total wall-time of the parallel algorithm come from the triangulation phase. Just like the sort phase, the triangulation phase scales strongly too. The parallel version with one process is slower than the

serial version due to overhead but for a higher number of processes the wall-times are much faster. This phase does not scale weakly, however, the benefits of parallelism are clear. When the point set contains 4000 points the wall-time for the serial version is about 70 minutes. The parallel version only takes less than 3 minutes with 64 processes. This significant reduction in wall-time illustrates the benefits of parallelism for our greedy triangulation algorithm.

6 Discussion

The parallel version of the greedy triangulation algorithm out performs the serial version. The triangulation phase in particular reaps the most benefits due to parallelism. If the scaling trends continue for larger number of processes then it is apparent that large point sets can be triangulated quickly on larger clusters. The speed up analysis shows that this algorithm for the greedy triangulation benefits greatly from parallelism. We conclude that parallelizing the greedy algorithm presented on page 3 was beneficial.

7 Conclusion

The greedy triangulation algorithm presented on page 3 benefits greatly from parallelization. The serial and parallel versions consist of three phases: generating the lines, sorting the lines, and producing the triangulation. The implementation of the parallel version on page 4 has nice scaling properties. While the line generation phase is slower due to communication, the sorting and triangulation phases are faster. The sorting phase scales both strongly and weakly. The triangulation phase of the serial and parallel versions is the most costly phase of the algorithm. The parallel version scales strongly and allows a triangulation to be computed in a fraction of the time. The speed up for the triangulation phase far outweighs the fact that the generation phase is slower. We conclude that parallelizing the greedy algorithm presented on page 3 was beneficial.

8 Future Work

Experiments on larger clusters should be conducted to further illustrate the scaling properties of the parallel implementation presented here. It would be useful to know if these trends continue. It is the author's contention that the speedup can be improved. One way to improve the performance of the program would be to use multi-threading on each process during the generation phases and the triangulation phases. Since the triangulation phase takes the most time multi-threading should be introduced there first. The portion of phase three which is most amenable to multi-threading is removing lines that intersect the line most recently added to the triangulation. Since the intersection of any two lines are independent of any other two lines this can be carried out efficiently on multiple threads. The line generation phase should benefit from multi-threading

for the same reason. It would also be beneficial to implement other triangulation algorithms and see how their wall-times compare to the results presented here. Additional future work could include creating implementations of the parallel algorithm to run on a GPU.

References

[1] Matthew T Dickerson et al. "Fast greedy triangulation algorithms". In: *Computational Geometry* 8.2 (1997), pp. 67–86 (cit. on p. 2).

[2] Robert L Scot Drysdale, Günter Rote, and Oswin Aichholzer. *A simple linear time greedy triangulation algorithm for uniformly distributed points.* Institutes for Information Processing Graz, 1995 (cit. on pp. 1, 2).

[3] Jesper Jansson. "Planar minimum-weight triangulations". In: *Master's Thesis, Department of Computer Science, Lund University, Sweden* (1995) (cit. on p. 2).

[4] Christos Levcopoulos and Drago Krznaric. "Quasi-Greedy Triangulations Approximating the Minimum Weight Triangulation." In: *SODA*. 1996, pp. 392–401 (cit. on p. 2).

[5] Christos Levcopoulos and Drago Krznaric. "The greedy triangulation can be computed from the Delaunay triangulation in linear time". In: *Computational Geometry* 14.4 (1999), pp. 197–220 (cit. on p. 2).

[6] Wolfgang Mulzer and Günter Rote. "Minimum-weight triangulation is NP-hard". In: *Journal of the ACM (JACM)* 55.2 (2008), p. 11 (cit. on p. 2).