Parallel Greedy Triangulation of a Point Set

Eliza Shoemaker and Randy Shoemaker

James Madison University
Parallel Greedy Triangulation of a Point Set
Eliza Shoemaker and Randy Shoemaker

Abstract
A greedy triangulation 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 is inexpensive computationally and gives an approximation of the minimum-weight triangulation problem, an NP-hard problem, which is computationally expensive. 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.

Index Terms - Computation theory, greedy algorithms, parallel algorithms, parallel programming
Introduction

A greedy triangulation algorithm takes a set of points on a 2D 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 approximate solution to the NP-hard minimum-weight triangulation (MWT) problem. As an NP-hard problem, the MWT is computationally expensive: it requires unworkably large amounts of "wall-time" and/or computer processors to arrive at an optimized solution. In contrast, a greedy triangulation algorithm is computationally inexpensive: it requires less actual elapsed time on a very precise clock on the wall—one that measures to .0001 of a second—and/or fewer linked computer processors in a world where most desktop computers have just four processors.

Triangulation is a classic CS problem and a greedy triangulation is one of the simplest and most natural algorithms for triangulation. A solution to the MWT problem is one of CS’s holy grails; more broadly, relatively optimized greedy triangulation approximations for the MWT have applications for graphics, data compression, and database systems. In this paper, 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.

1. Background

Before discussing the greedy triangulation algorithm and its computational aspects, we must first discuss some relevant concepts. An algorithm is a step-by-step procedure, terminating in a finite amount of time, which specifies how to solve instances of a particular problem. An algorithm has a worst-case complexity of $O(f(n))$ if the amount of computation needed to carry out the algorithm, in the worst case, grows on the order of $f(n)$ as the size of the input $n$ grows. Informally, a problem is considered NP-hard if it is at least as hard to solve as complex problems, such as the travelling salesman problem. In the travelling salesman problem, a list of $n$ cities and distances between them is given, and the goal is to find a way to visit each city exactly once such that the total distance travelled is minimized. As $n$ grows, it becomes intractable to find solutions to NP-hard problems. A $O(f(n))$-approximation to an NP-hard problem is a solution where the error between the approximation and the actual solution grows on the order of $f(n)$ as $n$ grows, which is relatively manageable or acceptable error.

A triangulation of a set of points is a collection of the points connected by edges such that the edges form triangles. Consider Fig. 1 and the points in Fig. 1a. The steps of the greedy triangulation algorithm can be seen in Fig. 1b through Fig. 1i, where the set of points in Fig. 1a is being triangulated. The greedy triangulation algorithm takes a set of points in the plane and returns a triangulation of the point set.

![Figure 1. Steps of the greedy triangulation algorithm.](image-url)
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.

The greedy triangulation has been an area of research for more than fifty years, in part because it gives an approximation of minimum-weight triangulation (MWT) [1]. 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. In 2008, Mulzer and Rote [2] proved the MWT problem to be NP-hard, which means that approximations for MWT are desirable. Earlier, Levcopolous and Krznanic [3] showed that the greedy triangulation gives a \( n \)-approximation of the MWT, where \( n \) is the number of points in the triangulation problem. This means that as the number of points grows, the difference between the greedy solution and the actual solution grows on the order of \( O(n) \) [3].

Dickerson et al. [1] developed an algorithm with an average case complexity of \( O(n) \) to compute the greedy triangulation. Their approach requires the point set to be uniformly distributed within a convex hull. The convex hull of a point set is a polygon formed by connecting the points with straight lines which contains the entire point set within its interior. Drysdale et al. [4] offered an improved \( O(n) \) algorithm that also requires the input set to be uniformly distributed in a convex hull. Levopoulos and Krznanic [5] showed that the greedy triangulation can be computed in linear time.

Parallel implementations of the greedy triangulation algorithm exist. Jansson [6] developed a parallel version which runs in \( O(n) \) on \( O(n^4) \) processors. This means that as the number of points \( n \) grows, the required number of computer processors grows on the order of \( n^4 \). For large point sets, Jansson's parallel version becomes impractical. For instance, if \( n = 1000 \), Jansson's version would require 1,000,000 processors; a typical desktop computer has four processors. 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 both the serial and parallel versions of the greedy triangulation algorithm.

2. Method

Fig. 2 presents a relatively reader-friendly pseudocode version of the serial algorithm we created using the C programming language. In this version, if there are \( n \) points in the point set, there are \( \frac{1}{2}n(n-1) \) lines between all points. Thus, it would seem that the serial 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 \( O(n^2) \) [4]. This is because, once a line is added to the triangulation, all lines which intersect it no longer need to be considered. Thus, as the algorithm progresses, lines are rapidly eliminated.

The serial algorithm consists of three phases: generate, sort, and triangulate. During the generate phase, the \( \frac{1}{2}n(n-1) \) 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 according to their length. In our implementation, we used the qsort algorithm from the C programming language's standard library of functions 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 approach of the algorithm in Fig. 2 can benefit from parallelism. It is for this reason that we chose to parallelize this algorithm.

Our parallel version of the greedy triangulation algorithm was created by modifying the serial version in the algorithm in Fig. 2. The serial version is a relatively generic algorithm, commonly referred to as a naive solution, as it was simple to come up with and seemed like the most natural solution. The parallel algorithm in Fig. 3 can also be divided into the same phases as the serial version. To achieve parallelization, we made some modifications. A parallel version of the generate phase was created, but experimentation showed that it was slower than the serial version. The generate phase is the same for both the serial and parallel versions, except that the lines generated in the parallel version are distributed to all processes. After the lines are generated, they are divided into subsets of equal size, and each subset is distributed to a process. Each process then carries out the sort phase in parallel. Once each process has sorted its local array of lines, the triangulate 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, which coordinates the other processes, adds the smallest line to the triangulation, and the process that
Figure 2. Serial greedy triangulation algorithm.

**Input**: Planar Point Set $P$

**Output**: Triangulation $T$

1. $lines \leftarrow \emptyset$
2. $n \leftarrow |P|$
3. $// \text{Phase 1: generate all lines}$
4. $lines \leftarrow \text{List of lines from } P[i] \text{ to } P[j] \text{ for all pairs } (i, j) \text{ where } i < j$
5. $// \text{Phase 2: sort the lines}$
6. Sort the contents of $lines$ in ascending order
7. $T \leftarrow \emptyset$
8. $unknowns \leftarrow |lines| \text{ // Number of lines with unknown status}$
9. $\text{while } unknown > 0 \text{ do}$
10. $\quad l^* \leftarrow \text{lines}[0]$
11. $\quad T \leftarrow T \cup \{l^*\} \text{ // Add smallest line to triangulation}$
12. $\quad lines \leftarrow lines - l^* \text{ // Remove the smallest line from lines}$
13. $\quad unknowns \leftarrow unknowns - 1$
14. $\quad \text{// Remove all lines that intersect with } l^*$
15. $\quad \text{for } l \in lines \text{ do}$
16. $\quad \quad \text{if } l \text{ intersects } l^* \text{ then}$
17. $\quad \quad \quad \quad lines \leftarrow lines - l \text{ // Remove the intersecting line}$
18. $\quad \quad \quad unknowns \leftarrow unknowns - 1$
19. $\text{return } T$

Figure 3. Parallel greedy triangulation algorithm.

**Input**: Planar Point Set $P$

**Output**: Triangulation $T$

1. $lines \leftarrow \emptyset$
2. $n \leftarrow |P|$
3. $\text{line.index} \leftarrow 0$
4. $\text{// Phase 1: generate all lines}$
5. $\text{if process is ROOT then}$
6. $\quad \text{// Only generate lines on the ROOT process}$
7. $\quad \text{for } i \leftarrow 0 \text{ to } n \text{ do}$
8. $\quad \quad \text{for } j \leftarrow i + 1 \text{ to } n \text{ do}$
9. $\quad \quad \quad \text{lines[index]} \leftarrow \text{the line from } P[i] \text{ to } P[j]$
10. $\quad \quad \text{line.index} \leftarrow \text{line.index} + 1$
11. $\text{Distribute equal sized subsets of lines to each process, initialize local.lines}$
12. $\text{// Phase 2: sort the lines (in parallel)}$
13. $\text{Sort the contents of local.lines on each process in ascending order}$
14. $\text{// Phase 3: greedily build the triangulation (in parallel)}$
15. $T \leftarrow \emptyset$
16. $unknowns.local \leftarrow |local.lines| \text{ // Number of lines with unknown status}$
17. $\text{while unknowns.local > 0 do}$
18. $\quad local^* \leftarrow \text{local.lines}[0]$
19. $\quad small.lines \leftarrow \text{AllGather}(local^*) \text{ // Share smallest line}$
20. $\quad l^* \leftarrow \text{smallest line in small.lines}$
21. $\text{if process is ROOT then}$
22. $\quad T \leftarrow T \cup \{l^*\} \text{ // Add smallest line to triangulation on ROOT}$
23. $\text{if } l^* \text{ in local.lines then}$
24. $\quad \text{// Remove the smallest line from the process’s local.lines and update its unknowns.local}$
25. $\quad local.lines \leftarrow local.lines - l^*$
26. $\quad unknowns.local \leftarrow unknowns.local - 1$
27. $\text{// Remove all lines that intersect with } l^*$
28. $\text{for } l \text{ in local.lines do}$
29. $\quad \text{if } l \text{ intersects } l^* \text{ then}$
30. $\quad \quad \text{local.lines} \leftarrow local.lines - l \text{ // Remove the intersecting line}$
31. $\quad unknowns.local \leftarrow unknowns.local - 1$
32. $\text{return } T$
has the smallest line removes this line 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.

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

3. Experiments
Experimental results for the serial and parallel versions were conducted on the computer cluster at James Madison University. A computer cluster is a group of interconnected computers that can carry out computations in parallel. The JMU cluster has 16 nodes, each containing 8 processors. We tested both versions using varying sizes of point sets. We present the time taken to carry out all computations, known as the wall-time, for each point set in Tables I, II, and III. 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. We used 707 points as a baseline because smaller numbers of points yield timings that are small enough to be significantly affected by noise on the cluster. Noise occurs because the same program can run on the cluster many times and take a different amount of time in each instance. While fluctuations are typically on the order of .0001 seconds or less, they create much more noise when experiments use smaller numbers of points. Our experiments used the following numbers of points: 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. We used powers of 2 so we could analyze the behavior to detect both strong and weak scaling, two important concerns that we discuss more fully in our Results section below.

4. Results
Tables I, II, and III present the wall-times of the three phases. Each data point is the smallest value observed for that particular entry across 6 trials. The values for the generate phase include the time it took to distribute the points from the ROOT to all other processes.

Before discussing our results, we present some necessary terminology about the significance of scaling. When parallelizing an application, the speedup is measured in two ways: weak scaling and strong scaling. A parallel solution exhibits weak scaling if an increase in the number of processors while holding the problem size constant reduces the run time. A parallel solution exhibits strong scaling if the run time remains constant while the number of processors and problem size increase at the same rate. It is worth noting that weak scaling and strong scaling are independent [7]. Based on the data presented in Tables I, II, and III, we can make the following observations about the three phases of the parallel version:

1. The cost of distributing the lines adds overhead to the generate phase;
2. In general, the sort phase scales both strongly and weakly;
3. The triangulate phase scales strongly and the speedup is significant.

Since the lines must be distributed during the generate phase and the generate phase takes place serially on the ROOT, the wall-time of the parallel version’s generate 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 triangulate phase. In Table I, the wall-times for the generate phase for a given input 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 generate phase in terms of elapsed time are offset by the benefits of parallelism for the sort and triangulate phases.

The sort phase scales strongly and weakly. Notice in Table II that, in general, if the number of points in a 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 weakly. Recall that in order to double the input size we must scale the number of points by $\sqrt{2}$. In Table II, the wall-time remains roughly constant as the input size is doubled. This means that the sort phase scales strongly. 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 triangulate 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 triangulate phase, shown in Table III. Just like the sort phase, the triangulate phase scales strongly. 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 takes less than 3 minutes with 64 processes.

5. Discussion
The parallel version of the greedy triangulation algorithm outperforms the serial version. The triangulate phase in particular reaps the most benefits due to parallelism because of the significant reduction in wall-time.

If the scaling trends continue for larger numbers of processes, then it is apparent that large point sets can be triangulated quickly on larger clusters. The speedup analysis shows that this algorithm for the greedy triangulation benefits greatly from parallelism.

6. Conclusion
The greedy triangulation algorithm in Fig. 1 benefits greatly from parallelization. The serial and parallel versions consist of three phases: generating the lines, sorting the lines, and producing the triangulation. The parallel version in Fig. 2 has nice scaling properties. While the generate phase is slower due to communication between the ROOT process and the other processes, the sort and triangulate phases are faster. The sorting phase scales both strongly and weakly. The triangulate phase of the serial and parallel versions is the most cost-

<table>
<thead>
<tr>
<th>Points</th>
<th>Serial</th>
<th>1 P</th>
<th>2 P</th>
<th>4 P</th>
<th>8 P</th>
<th>16 P</th>
<th>32 P</th>
<th>64 P</th>
</tr>
</thead>
<tbody>
<tr>
<td>707</td>
<td>0.0113</td>
<td>0.0522</td>
<td>0.0312</td>
<td>0.0316</td>
<td>0.0193</td>
<td>0.0561</td>
<td>0.0762</td>
<td>0.1097</td>
</tr>
<tr>
<td>1000</td>
<td>0.0218</td>
<td>0.1014</td>
<td>0.0602</td>
<td>0.0518</td>
<td>0.0382</td>
<td>0.1109</td>
<td>0.1492</td>
<td>0.1784</td>
</tr>
<tr>
<td>1414</td>
<td>0.0429</td>
<td>0.2025</td>
<td>0.1176</td>
<td>0.1196</td>
<td>0.0730</td>
<td>0.2200</td>
<td>0.2955</td>
<td>0.3391</td>
</tr>
<tr>
<td>2000</td>
<td>0.0843</td>
<td>0.4433</td>
<td>0.2325</td>
<td>0.2007</td>
<td>0.1938</td>
<td>0.4715</td>
<td>0.5908</td>
<td>0.6798</td>
</tr>
<tr>
<td>2828</td>
<td>0.1673</td>
<td>0.7973</td>
<td>0.5070</td>
<td>0.4265</td>
<td>0.3774</td>
<td>0.8694</td>
<td>1.1742</td>
<td>1.3334</td>
</tr>
<tr>
<td>4000</td>
<td>0.0330</td>
<td>1.6651</td>
<td>1.1968</td>
<td>0.7969</td>
<td>0.6144</td>
<td>1.7375</td>
<td>2.3577</td>
<td>2.6662</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Points</th>
<th>Serial</th>
<th>1 P</th>
<th>2 P</th>
<th>4 P</th>
<th>8 P</th>
<th>16 P</th>
<th>32 P</th>
<th>64 P</th>
</tr>
</thead>
<tbody>
<tr>
<td>707</td>
<td>8.6674</td>
<td>9.3801</td>
<td>4.6939</td>
<td>4.6939</td>
<td>2.8247</td>
<td>1.7278</td>
<td>1.2628</td>
<td>1.0404</td>
</tr>
<tr>
<td>1414</td>
<td>93.683</td>
<td>139.72</td>
<td>73.512</td>
<td>73.512</td>
<td>40.841</td>
<td>24.744</td>
<td>10.667</td>
<td>5.3011</td>
</tr>
<tr>
<td>2000</td>
<td>345.38</td>
<td>511.96</td>
<td>259.15</td>
<td>259.15</td>
<td>150.19</td>
<td>116.75</td>
<td>43.747</td>
<td>21.820</td>
</tr>
<tr>
<td>2828</td>
<td>1187.1</td>
<td>1841.7</td>
<td>956.47</td>
<td>956.47</td>
<td>541.69</td>
<td>386.83</td>
<td>179.37</td>
<td>92.878</td>
</tr>
<tr>
<td>4000</td>
<td>4311.5</td>
<td>6878.5</td>
<td>43473</td>
<td>43473</td>
<td>2192.0</td>
<td>1301.9</td>
<td>661.94</td>
<td>333.28</td>
</tr>
</tbody>
</table>
ly phase of the algorithm. The parallel version scales strongly and allows a triangulation to be computed in a fraction of the time. The speedup for the triangulate phase far outweighs the fact that the generate phase is slower. We conclude that parallelizing the greedy triangulation algorithm as in Fig. 2 is beneficial.

7. 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 authors’ contention that the speedup can be improved. One way to improve the performance of the program would be to use multithreading on each process during the generate phases and the triangulate phases. Multithreading involves many processes, sharing a common memory which execute on the same processor. Since the triangulate phase takes the most time, multithreading 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 multithreading 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. Other efforts could include a theoretical analysis of the parallel algorithm and creating implementations of the parallel algorithm to run on recent graphics processing units designed to carry out the same operation on many pieces of data at the same time.

Author’s Note

Eliza Shoemaker
Eliza Shoemaker (’19) graduated with a bachelor’s degree in Computer Science with a minor in Logic and Reasoning. She is currently working as a software engineer in Eastern Virginia. Eliza enjoys solving puzzles and drinking tea.

Randy Shoemaker
Randy Shoemaker (’19) graduated with a bachelor’s degree in Computer Science and Mathematics with a minor in Logic and Reasoning. He is currently a Ph.D. student at the College of William and Mary and conducts research in the area of Geometry Processing. He enjoys playing D&D with his son.

References


