Greedy Triangulation via GPU

Randy Shoemaker
Department of Computer Science
College of William and Mary
Williamsburg, USA
rwshoemaker@email.wm.edu

***Abstract*—We present a GPU implementation of the greedy triangulation algorithm from [1]. The version from Shoemaker et al. utilized the Message Passing Interface (MPI) which parallelized the greedy triangulation algorithm to run on a cluster. We present a version utilizing CUDA to run on many streaming multiprocessors which achieves significant speed up over Shoemaker et al. Our GPU implementation and our experimental data can be found in our Github repository[1].**

I. Introduction

The greedy triangulation algorithm takes a set of points in the plane and greedily produces a triangulation. It builds the triangulation in three steps. In the first step all lines between each pair of points is generated, then these lines are sorted in increasing size. The third step is the triangulation step, during this step the smallest line is added to the triangulation and all lines that conflict with this line are removed, then the next smallest line is added and this step is repeated. The triangulation step is the most costly step of the algorithm. The first two steps combined take a very small fraction of the overall run time of the algorithm. Due to this fact we focused our efforts on parallelizing the triangulation step using CUDA.

The greedy triangulation algorithm is known to give a $\sqrt{n}-$approximation of the Minimum Weight Triangulation problem (MWT) [1]. This problem has been shown to be NP-hard [2]. Triangulation problems arise in many domains, such as computational geometry and geometry processing. Thus, quickly producing a triangulation is useful for many applications. We present a parallel version of the greedy triangulation algorithm which utilizes CUDA to run on many streaming multiprocessors. Previous work has produced a parallel version which runs on a cluster utilizing the Message Passing Interface (MPI) [1]. Our approach achieves significant speedup when compared to the cluster version. We present an empirical comparison to the previous work as well as a comparison to the serial version. We also present a theoretical analysis of our approach vs others. In the appendix we include images of some triangulations produced by our code.

In Section II we discuss the greedy triangulation algorithm and in Section III we discuss our implementation. In Section IV we present a theoretical analysis of our algorithm and in Section V we present our empirical results. In Section VI we provide a discussion of our results before concluding in Section VII. Lastly, we discuss potential future work.

II. Background

Before discussing the greedy triangulation algorithm we first present some notation. Let P be a point set we wish to triangulate using the greedy algorithm; for convenience we let $n = |P|$. Let $\mathcal{L}$ denote the set of all line segments formed by connecting every pair of points in P. Let P_i denote the i^{th} element of P. Given P the greedy triangulation algorithm produces the greedy triangulation $\mathcal{T}$. With this notation we present the greedy triangulation algorithm.

```
Input: P  // The point set
Output: T // The triangulation
L ← ∅
n ← |P|
idx ← 0
// Line generation step
for i ← 0 to n
    for j ← i + 1 to n
        L_idx ← the line from P_i to P_j
        idx ← idx +1
// Sorting step
Sort L in ascending order
// Triangulation step
T ← ∅
while L ≠ ∅
    I ← L_0
    remove I from L
    T ← T ∪ {I}
    for J ∈ L:
        if(not share_endpoint(I, J)
           && intersects(I, J)):
                remove J from L
return T
```

The function `share_endpoint` takes two lines and returns true iff the lines share an endpoint and `intersects` takes two lines and returns true iff they intersect. When implemented, the first two steps, the generation and sorting steps, take up a very small portion of the over all run time. Below, we have included some sample run times of a serial implementation of the algorithm. The run times are in seconds.

[1]The url for our repository: https://github.com/shoemarw/gpu_greedy_triangulation

Points	Generation	Sort	Run time (seconds)
724	0.0149	0.0532	7.9712
1024	0.0236	0.1027	24.996
1448	0.0724	0.2175	98.2896
2048	0.0847	0.4431	385.6797
2896	0.1786	0.9174	1407.0545

Since the triangulation step is the most computationally intensive, our work focuses on parallelizing this step.

In the following section we present our parallel version of the algorithm.

III. The Parallel Algorithm

We now present our parallel version. First we must establish some further notation. Let T be an array of boolean values which are initially all true. During the execution of the algorithm, T_i becomes false exactly when $\mathcal{L}_i$ is found to conflict with a line that is in $\mathcal{T}$. Once the algorithm has concluded T_i is true iff $\mathcal{L}_i$ belongs to the triangulation. Below is the pseudo code for our implementation. Since the triangulation step is the only step parallelized we only include pseudo code for it.

```
Input: L  // The sorted lines
       b  // Number of blocks
       t  // Number of threads
Output: T // The triangulation
// Triangulation step
T ← {True_i}_{i=1}^{|L|} // Include every line
s ← 0 // Index of smallest line
while s < |L|
    triangulate<b,t>(L, s, T)
    find_new_small<1,1>(s, T)
T ← {L_i ∈ L : T_i is True}
return T
```

The implementation of this part of the algorithm runs on the host. The implementations of the functions `triangulate` and `find_new_small` run on the device and are launched from the host. These kernel calls allow us to synchronize the blocks. `triangulate` takes a line and eliminates all lines which conflict with it by updating the corresponding elements of T to false. `find_new_small` updates the index s of the smallest line after a round of eliminations. These functions have no return values and manipulate T and s respectively on device memory to minimize data transfers between the host and the device. Below we have included pseudo code for `triangulate` and `find_new_small`. The number of blocks b and threads per block t are command line arguments for the implementation for experimental purposes.

```
triangulate(L, s, T)
    j ← blockID.x*blockDim.x + threadId.x
    while j < |L|
        if T_j is True and conflicts(L_s, L_j)
            T_j ← False
        j ← j + blockDim.x*gridDim.x
```

```
find_new_small(s, T)
    s ← s + 1
    while s < |L| and T_s is false
        s ← s + 1
```

`find_new_small` simply increments the index of the smallest line so that the eliminated lines are skipped over. `triangulate` eliminates lines that conflict with the smallest line during a round in parallel. This task occurs over b blocks, each containing t threads. After a round of eliminations we must increment s on one block while the others wait. In order to accomplish this kind of synchronization we use kernel calls from the host.

The device function `triangulate` is written so that the cache is used as efficiently as possible, we used Nvidia's blogs[2] as a guide. The data that each thread operates on was tuned to achieve maximum memory coalesce. `triangulate` calls the device function `conflicts` which tests two lines to see if they conflict with one another, it in turn calls other functions. We include pseudo code for these functions in the appendix.

In the following section we present a theoretical analysis of our implementation.

IV. Theoretical Analysis

In this section we present an analysis of the number of floating point operations performed by the triangulation step of the greedy triangulation algorithm. For a given number of points n we compute an approximation of the number of floating point operations required by using an average case analysis.

Let N_n be the total number of floating point operations needed to produce a triangulation during the triangulation step. Notice that N_n can be computed if we know the number of conflict tests are executed and the number of floating point operations per conflict test. So let C_n be the number of conflict tests needed for a point set with n elements, and let F_n be the number of floating point operations required for one conflict test if there are n points in the point set. C_n clearly depends on n, to see why F_n depends on n notice that `share_endpoint` is computed for every conflict test and this function makes several calls `is_equal`. C_n depends on n because the probability that `is_equal` is true for two given points depends on the number of lines which is a function of n. Since we are doing an average case analysis this probability is used to estimate the floating point operations needed. In the appendix we provide the pseudo code for every function analyzed here.

We now compute an approximation of C_n. Let A_n be the average number of lines in the greedy triangulation of a point set with n elements. A_n is important because, on average, it is the number of iterations of the while loop in `triangulate` averaged across all point sets with n points. We ran experiments using different point sets to arrive at a

[2]Please see https://devblogs.nvidia.com/how-access-global-memory-efficiently-cuda-c-kernels/

suitable approximation of A_n. For several values of n we generated five unique point sets. The values of n used were $4, 16, 64, 256, 512, 1024$, and 2048. For each n we found that the number of lines in the triangulation were always close. For each n we took the average of the number of lines in the triangulations[3]. The averages are included in the below table

Points	approximation of A_n
4	5
16	37.2
64	182.6
256	858.4
512	1963.4
1024	4874.8
2048	14056

We then fit a quadratic to the approximations of A_n, we use a quadratic because A_n appears to grow on the order of n^2. The resulting function is

$$A_n = 2.01931 * 10^{-3} n^2 + 2.7221n + 5.27957$$

Now that we have the number of iterations of the while loop A_n, to compute C_n we need to estimate the number of conflict tests during a typical iteration of the while loop. For the sake of analysis we assume that an equal number of lines are eliminated during each iteration of the while loop. We believe that this assumption is reasonable when averaged across all point sets for a given n. Under this assumption $\frac{|\mathcal{L}_n|-A_n}{A_n}$ lines are eliminated during an iteration of the while loop. It follows that at iteration i there are $|\mathcal{L}_n| - (i-1)\frac{|\mathcal{L}_n|-A_n}{A_n} - 1$ left that could potentially be part of the triangulation. That means that at iteration i each of these lines must be tested using `conflicts`. Therefore at iteration i there must be $|\mathcal{L}_n|-(i-1)\frac{|\mathcal{L}_n|-A_n}{A_n}-1$ conflict tests and across all iterations and there must be a total of

$$C_n = \sum_{i=1}^{A_n} |\mathcal{L}_n| - (i-1)\frac{|\mathcal{L}_n| - A_n}{A_n} - 1$$

conflict tests. This expression reduces to $C_n = \frac{1}{2}(A_n^2 + A_n(\mathcal{L}_n - 3) + \mathcal{L}_n)$. By plugging in the experimental approximation of A_n this becomes

$$C_n = 5.06866 * 10^{-4} n^4 + 0.685517n^3 + 4.60191n^2 + 8.71847n + 6.01757$$

We now derive an approximation of the number of floating point operations per call to `conflicts`, F_n. We do an average case analysis to find F_n. To compute F_n we note that when evaluating a compound boolean expression in CUDA C short-circuiting is used. So, for example, if `!share_endpoint`(I, J) evaluates to false `conflicts` returns false immediately and no further computation occurs. To compute F_n we first compute the cost of all functions called by `conflicts`. We start with `is_equal`.

[3]This data is included in the repository in the folder find_An. This folder also contains code to generate this experimental data.

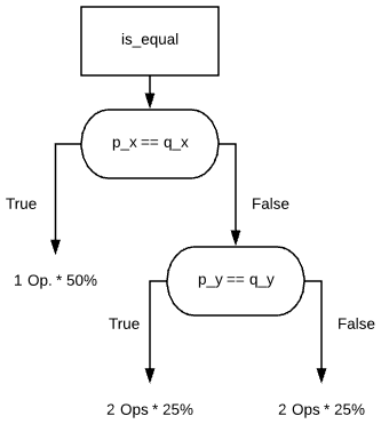

Fig. 1. The execution tree for `is_equal`.

Consider the execution tree of `is_equal` in Figure 1. If we assume that each branch is equally likely then the average cost in terms of operations is $1 * .5 + 2 * .25 + 2 * .25 = 1.5$ operations. With this we can compute the average cost of `!share_endpoint`(I, J), which is 1 plus the average cost of `share_endpoint`(I, J).

Before we consider the execution tree of `share_endpoint`(I, J) we must consider the probability that two points are equal. Notice that each point is paired with $n-1$ points and in total there are $\frac{n(n-1)}{2}$ lines. It follows that the probability that two randomly chosen points are equal is

$$\frac{n-1}{\frac{n(n-1)}{2}} = \frac{2}{n}$$

Thus the probability that `is_equal` returns true is $\frac{2}{n}$. Consider the execution tree of `share_endpoint` in Figure 2. To compute the average cost of `share_endpoint` we take the the sum of each branch in Figure 2, this sum reduces to $6 - \frac{18}{n} + \frac{24}{n^2} + \frac{12}{n^3}$.

Before analyzing `intersects` we analyze `orient` and `lies_on`. Each call to `orient` costs 9 floating point operations. For the cost of `lies_on` consider its execution tree in Figure 3. For simplicity we label each condition as condition 1, condition 2, etc. and assume that either truth value of each condition is equally likely. From the tree we can see that the average cost of `lies_on` is $\frac{15}{4}$.

We can now compute the average cost of `intersects` and `conflicts`. Since `conflicts` calls `share_endpoint` we must include the cost of it and the probability of taking its right most branch into account for each branch of the execution tree of `intersects`. We note that the execution tree of `intersects` is a sub tree of the execution tree of `conflicts` so we include them both in the same figure. Consider the execution tree in Figure 4. From the tree we can

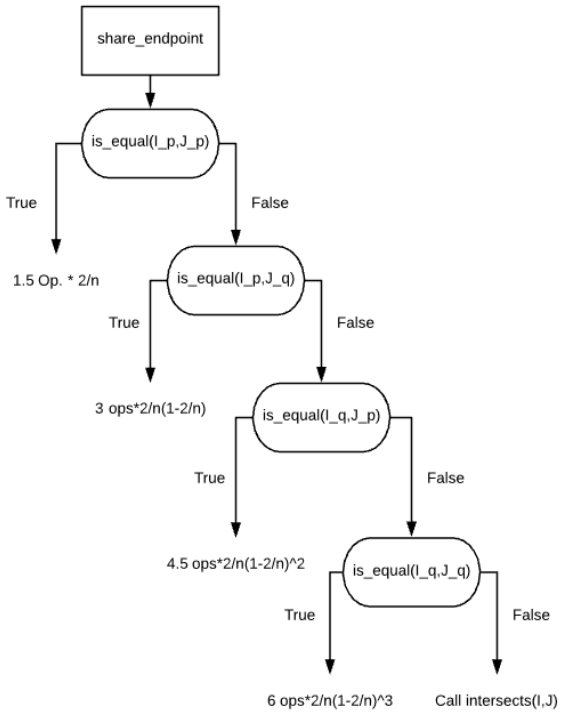

Fig. 2. The execution tree for `share_endpoint`.

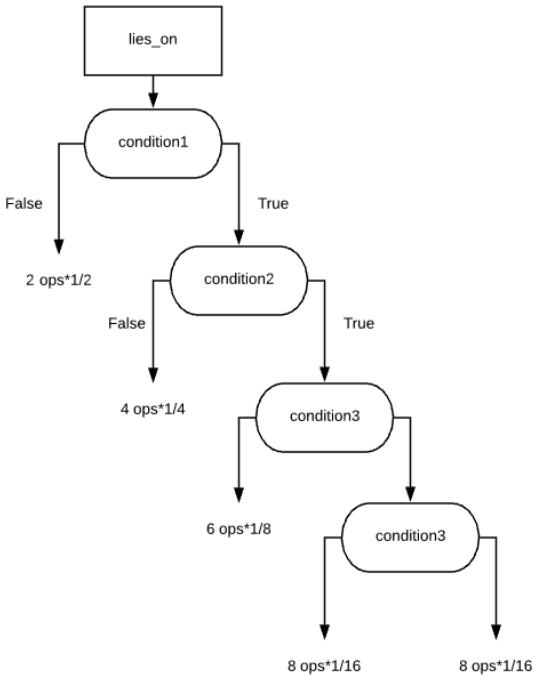

Fig. 3. The execution tree for `lies_on`.

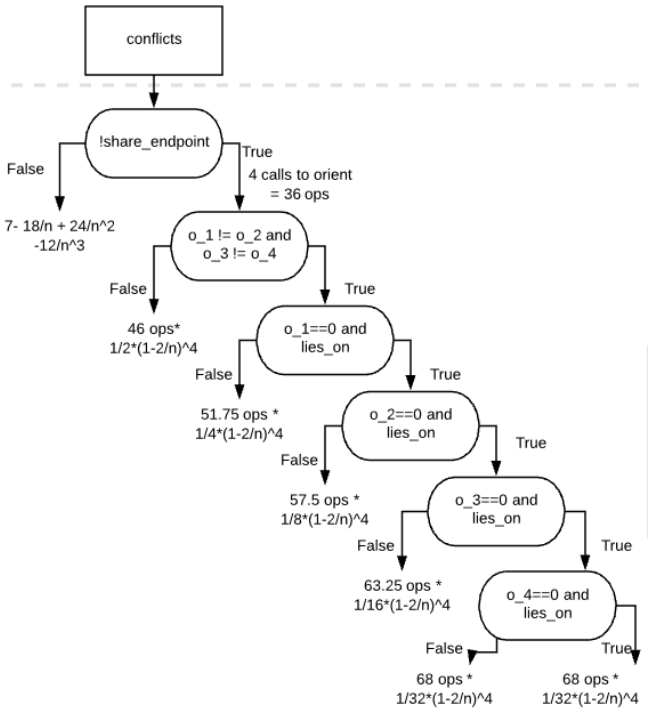

Fig. 4. The execution tree for `conflicts`.

see that the total cost of `conflicts` is

$$\begin{aligned} F_n &= 7 - \frac{18}{n} + \frac{24}{n^2} - \frac{12}{n^3} + (1 - \frac{2}{n})^4(\frac{46}{2} + \frac{51.75}{4} + \frac{57.5}{8} \\ &\quad + \frac{63.25}{16} + \frac{68}{16}) \\ &= 58.3281 + \frac{821.25}{n^4} - \frac{1654.5}{n^3} + \frac{1255.88}{n^2} - \frac{428.625}{n} \end{aligned}$$

We emphasize that F_n is the cost of an average call to `conflicts`, so summing F_n over many calls to `conflicts` yields more accurate estimates.

Since we have approximations for F_n and C_n we can now compute N_n, the total number of operations needed to complete the triangulation step for n points. Notice that N_n is the product of C_n and F_n plus the total number of operations needed to keep track of the index of the smallest line. The latter cost is the total cost of all calls to `find_new_small`, which is $3|\mathcal{L}_n| - 2A_n$. Therefore the total cost of the triangulation step is

$$\begin{aligned} N_n &= C_n F_n + 3|\mathcal{L}_n| - 2A_n \\ &= 0.0295645n^4 + 39.7676n^3 - 24.7725n^2 - 603.873n \\ &\quad + 1259.71 + \frac{1319.19}{n} - \frac{3088.04}{n^2} - \frac{2796.03}{n^3} + \frac{4941.93}{n^4} \\ &\quad + 3|\mathcal{L}_n| - 2A_n \\ &= 0.0295645n^4 + 39.7676n^3 - 23.2765n^2 - 610.817n \\ &\quad + 1249.15 + \frac{1319.19}{n} - \frac{3088.04}{n^2} - \frac{2796.03}{n^3} + \frac{4941.93}{n^4} \end{aligned}$$

With this result, given the time taken to complete the triangulation step, we can estimate the FLOPS of our parallel

version. We note that the same computation takes place in the serial version so we may also estimate the FLOPS of the serial version using N_n as a baseline for comparison. In the following section we present experimental results of our parallel version and in Section VI we give a discussion of our results.

V. Experiments

We now present our experimental results.The following data was gathered by executing the serial version of the algorithm on the bg9 computer in the Computer Science department of William and Mary. The values for N^n are reported in giga-operations and the FLOPS are reported as GFLOPS. The function N^n is the approximation of floating point operations required by the algorithm. It was derived in the previous section. This data is used as a baseline for our parallel implementation.

Points	N^n	Run time (seconds)	GFLOPS
724	23.2	7.9712	2.91
1024	75.1816	24.996	3.01
1448	250.656	98.2896	2.55
2048	861.606	385.6797	2.23
2896	3045.22	1407.0545	2.16

We now present the statistics for our parallel CUDA-C implementation. The following data was gathered by executing our parallel version on the GPU in the bg9 computer in the Computer Science department of William and Mary. We only present a subset of the data using the best combination of the number of blocks (B) and number of threads per block (T) for each point set. To see all of the data generated by our tests please refer to the Github repository[4]. As with the values for the serial version N^n is reported in giga-operations.

n	B	T	N^n	Time (s)	GFLOPS
724	256	512	23.2	0.2794	83.0
1024	256	1024	75.18	0.4629	162.4
1448	256	1024	250.7	0.9115	275.0
2048	256	1024	861.61	2.2405	384.6
2896	256	1024	3045.2	6.2929	484.0
4096	512	1024	11054	19.5484	565.5
5792	512	1024	40999	66.6196	615.4
8192	512	1024	155008	240.101	645.6

We were able to triangulate point sets with much larger values due to the massive speedup. In the serial version we did not triangulate point sets this large due to the large amount of time it would take.

In the following section we give a discussion of our results.

VI. Results

We now discuss our results. In the previous section we presented data from our experiments. We can see that the serial version scales poorly in terms of GFLOPS, in fact as the number of points in the point set grows, the theoretical GFLOPS actually decreases. We believe that this is due to the effects of moving the processed lines in and out of the cache. While this occurs the CPU is idle, thus lowering the amount of FLOPS. As the number of lines grows thrashing in the cache grows too. We believe that this explains the diminishing performance of the serial version as the number of points grows.

Now consider our parallel version. As the number of lines in the point set increases we can see that the theoretical GFLOPS actually increases, though it appears increase asymptotically. We believe that this increase in GFLOPS is due to the effects of memory coalescing. The way the lines are accessed by threads across blocks in the parallel version is designed so that cache evictions are minimized. Each block of threads only accesses lines that are near each other in the line array, so the lines stay in the cache of the streaming multiprocessors longer, thus minimizing evictions. We believe that this explains the increasing performance of the parallel version.

The parallel version enables larger point sets to be triangulated in a practical amount of time. Consider the serial version, as the number of points increases by a factor of $\sqrt{2}$ the run time roughly increases by a factor of 4. This means that triangulating a point set with $8192 = 2896 * \sqrt{2} * \sqrt{2} * \sqrt{2}$ points should take roughly $1407 * 4 * 4 * 4 = 90,048$ seconds, which is about 25 hours. The parallel version with 512 blocks and 1024 blocks per thread only takes about 240 seconds, or 4 minutes.

We now compare our results to the previous MPI version in [1]. We do not perform a comparison using our theoretical analysis because the MPI implementation is very different than this version, so N^n is not an accurate approximation of the number of floating point operations needed to perform the triangulation step. Here we only compare the run times of the triangulation step of the two versions. Below is a table containing the amount of time needed to accomplish the triangulation step using the MPI version. The stats are from [1] and were run on the cluster at the Computer Science department at James Madison University in April 2019.

Points	Time on 64 P in (s)
707	1.2657
1000	2.3979
1414	5.3011
2000	12.742
2828	46.203
4000	177.73

We can see that with point sets of comparable size, the GPU version is faster, thus enabling larger point sets to be triangulated. For example, our version is able to compute a point set with 4096 elements in about 20 seconds, where as the MPI version takes about 3 minutes. This is a speedup by a factor of 9.

We can conculde that this implementation allows larger point sets to be triangulated more quickly and efficiently than

[4]Specifically the file log.csv contains all experiments performed using our best parallel version. log.csv was generated using experiments.sh which is also included in the repository.

before. In the following section we provide our conclusions and afterwords we discuss future work.

VII. Conclusion

The CUDA-C implementation presented here scales nicely in terms of both run time and theoretical GFLOPS. Our results and theoretical analysis show that our implementation can triangulate large point sets quickly and efficiently. When compared to the serial version our results show that our implementation is significantly faster, allowing a point set of 8192 points to be triangulated in 4 minutes instead of 25 hours. When compared to the previous work in [1] our implementation is also quicker, being able to triangulate point sets up to nine times faster. We can conclude that them implementation presented here is robust and enables greedy triangulations of large point sets to be carried out in a reasonable amount of time.

VIII. Future Work

We believe that this implementation can be further improved. One way it can be improved is by inlining all the functions discussed in the appendix so the code for them does not have to be moved in and out of the cache. We also believe that this algorithm can be improved by producing a hybrid of this version and the MPI version. This future hybrid implementation should run on multiple nodes, each containing several streaming multiprocessors. The inter-node communications should be handled as in the MPI version and the intra-node processing should occur as in this work.

We also believe that the theoretical approximation of floating point operations can be improved to be more accurate. We believe that it would be beneficial to do a study to test the assumption that an equal number of lines are eliminated during each round of eliminations. If this assumption in false, which is likely, a function $g_{n,r}$ should be derived using experimental data that approximates the number of lines eliminated at round r for a point set of size n. We also think that experimental data should be used to compute the probability that certain paths are taken in the execution trees. This can be used to derive a more accurate N_n.

References

[1] E. Shoemaker and R. Shoemaker, "Parallelizing the greedy triangulation." submitted to James Madison University Research Journal (JMURJ).

[2] W. Mulzer and G. Rote, "Minimum-weight triangulation is np-hard," *Journal of the ACM (JACM)*, vol. 55, no. 2, p. 11, 2008.

IX. Appendix

Here we include the pseudo code for all device functions needed to check if two lines conflict with one another. We note that in these functions zero corresponds to false and a non-zero number corresponds to True. Please refer to the Github repository for implementations of these functions.

The function `conflicts` takes two lines and returns true iff the two lines conflict with one another.

```
conflicts(I, J)
    return !share_endpoint(I, J)
            and intersects(I, J)
```

The function `share_endpoint` takes two lines and returns true iff the two lines share an endpoint. This function is important because if two lines share and endpoint they intersect but they do not conflict with one another. We let I_p and I_q denote the endpoints of the line I.

```
share_endpoint(I,J)
    return is_equal(I_p, J_p)
        or is_equal(I_p, J_q)
        or is_equal(I_q, J_p)
        or is_equal(I_q, J_q)
```

The function `is_equal` takes two points and returns true if they are the same point. We let p_x and p_y denote the x and y coordinates respectively of the point p.

```
is_equal(p, q)
    return p_x == q_x and p_y == q_y
```

The function `intersects` takes two lines and tells us if they intersect.

```
intersects(I, J)
    o_1 ← orient(I_p, I_q, J_p)
    o_2 ← orient(I_p, I_q, J_q)
    o_3 ← orient(J_p, J_q, I_p)
    o_4 ← orient(J_p, J_q, I_q)
    return (o_1 != o_2 and o_3 != o_4)
        or (o_1 == 0 and lies_on(I_p, J_p, I_q)
        or (o_2 == 0 and lies_on(I_p, J_q, I_q)
        or (o_3 == 0 and lies_on(J_p, I_p, I_q)
        or (o_4 == 0 and lies_on(J_p, I_q, J_q)
```

The function `orient` takes three points p, q, r and the return value depends on the orientation of the traversal from p to q to r. If the traversal is clockwise 1 is returned, if the traversal is counter-clockwise -1 is returned, if the points are colinear then 0 is returned. The function `sign` simply returns the sign of its argument.

```
orient(p, q, r)
    return sign((q_y − p_y)(r_x − q_x)
              − (q_x − p_x)(r_y − q_y))
```

The function `lies_on` takes three points p, q, r and returns true iff q lies on the line segment formed by p and r. The functions `max` and `min` return the max and min respectively of their arguments.

```
lies_on(p, q, r)
    return q_x ≤ max(p_x, r_x) and q_x ≥ min(p_x, r_x)
       and q_y ≤ max(p_y, r_y) and q_y ≥ min(p_y, r_y)
```

Below are some images of various triangulations produced by our parallel version, we include them because the are the visually appealing product of this work.

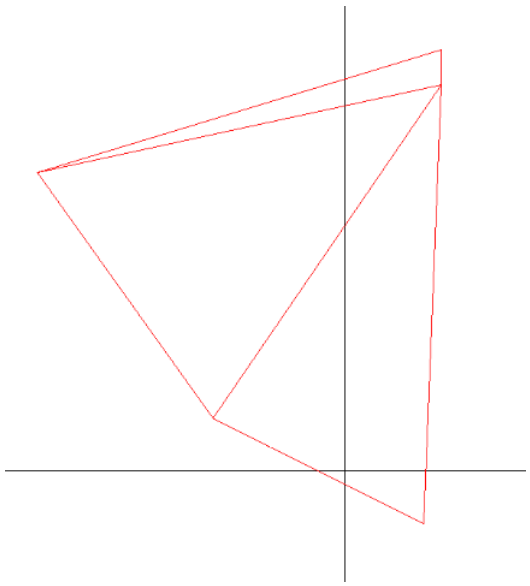

Fig. 5. A point set with 5 points

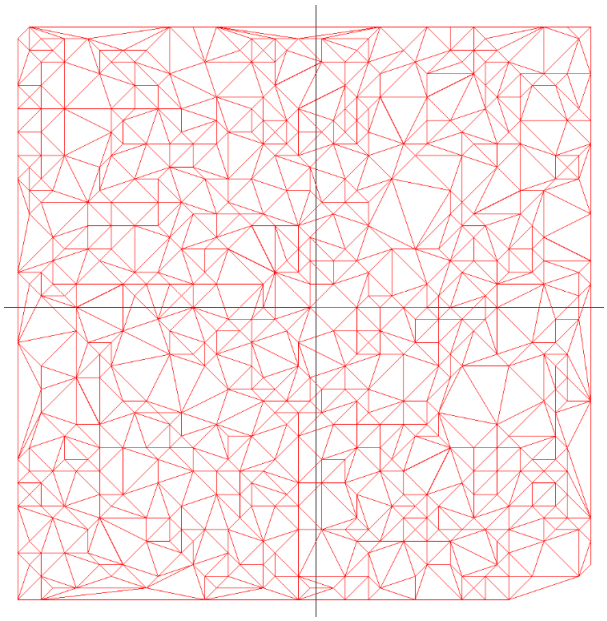

Fig. 7. A point set with 707 points

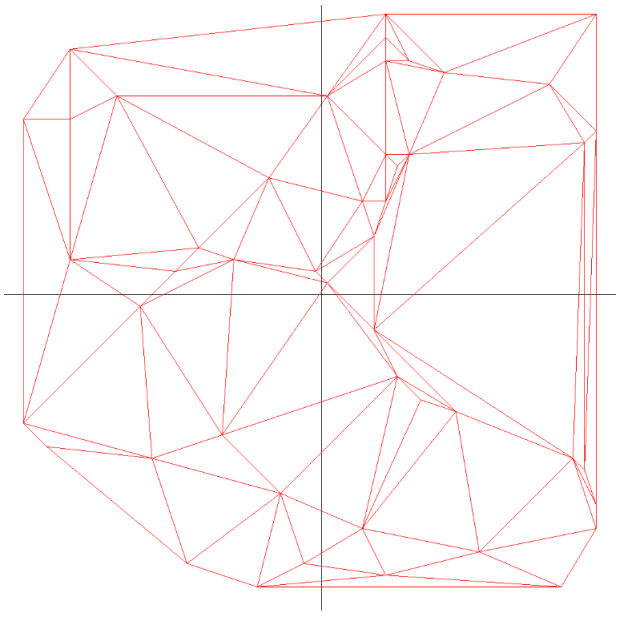

Fig. 6. A point set with 50 points

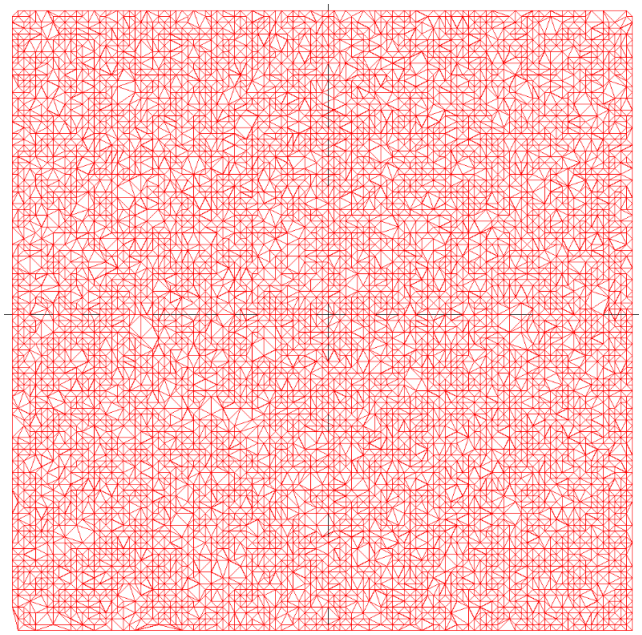

Fig. 8. A point set with 11585 points