Pessimistic Threaded Delaunay Triangulation by Randomized Incremental Insertion

Ivana Kolingerová[1], Josef Kohout
Department of Computer Science and Engineering, University of West Bohemia
Pilsen, Czech Republic

Abstract

Delaunay triangulation is one of the most popular triangulations and many algorithms for its construction has been developed, some of them also for parallel environment. However, most of the existing parallel solutions are intended for higher degree of parallelism; they are relatively effective but complicated. This paper describes simple parallel algorithm suitable for lower degree of parallelism, namely for workstations with 2-8 processors. Algorithm works with one shared triangulation, therefore, no expensive divide or merge phases are necessary. It is conceptually simple as it is based on well-known incremental insertion method.

Keywords: *Computer Graphics, Computational Geometry, Triangulation, Parallelization*

1. INTRODUCTION

To triangulate a set of points in the plane is often solved task as triangulations are used in many areas, such as computer graphics and computational geometry, mathematics, robotics, natural sciences. Usual requirements on triangulation are good shape of triangles (as equiangular as possible) and efficient computation. Both is fulfilled by Delaunay triangulation which produces the most equiangular triangles of all possible methods and can be computed in $O(n \log n)$ in the worst case; algorithms running in $O(n)$ expected time also exist [14]. Good survey and evaluation of existing methods can be found in [24, 22].

No wonder that efficient construction of Delaunay triangulation is one of the problems that are tried to be solved also in parallel and distributed environment and new solutions are developed.

In the past, parallel processing was possible only on special architectures, expensive and not widely accessible. The situation has changed nowadays as workstations with two processors become more and more often and not too expensive. Also architectures with 4, 8 processors are relatively available. These types of computers in hands of wider computer community mean that at least limited degree of parallelism is nowadays accessible to much wider group of computer people than used to be. This increases hunger for good and simple parallel solutions, suitable for low number of processors while in the past, solutions for large number of processors were preferred.

This paper suggests a very simple parallel solution for Delaunay triangulation intended for low number of processors (two to eight). The suggested approach is based on the randomized incremental insertion algorithm. Results of implementation and experiments done on Windows NT are shown and pros and cons of the proposed approach presented. Research directions for the future are suggested.

The paper is divided as follows. Section 2 defines Delaunay triangulation and surveys main parallel achievement on this topic. Section 3 describes serial version of incremental insertion and the proposed parallel version. Section 4 presents results of experiments and discussion. Section 5 concludes the paper.

2. STATE OF THE ART IN PARALLEL DELAUNAY TRIANGULATION

[Def.2.1: Triangulation]
A triangulation $T(P)$ of a set of points P in the Euclidean plane is a set of edges E such that

1. no two edges in E intersect in a point not in P
2. the edges in E divide the convex hull of P into triangles.

[Def.2.2: Delaunay triangulation]
The triangulation $T(P)$ of a set of points P in the plane is a Delaunay triangulation of P if and only if the circumcircle of any triangle of $T(P)$ does not contain a point of P in its interior.

Parallelizing Delaunay triangulation is not easy as each point may influence the whole triangulation. If each processing element (PE) knows only part of the input set, constructed parts of triangulation are not completely Delaunay and has to be corrected as a post-processing. Generally, divide and conquer (D&C) type of methods are considered most proper to parallelization. There are two ways how D&C in $DT(P)$ is done: either simple partition and difficult merge or vice versa. Therefore, either merging phase or subdivision phase is complicated and inherently serial. If parallelized, too, it substantially increases inter-processor communication.

Let's survey now what has been done on the topic of Delaunay triangulation.

Theoretical parallel algorithms appeared in [1, 8, 26, 21, 15]. Efficient parallel implementations are described in [20, 7, 25, 23], very often they depend on uniform distribution as they utilize bucketing techniques.

[1] was the first one to present a parallel version of the merge phase at the expense of algorithmic complications and decreased efficiency.

In [9], the given point set is partitioned into strips of the same size, each strip is allocated to one coarse-grained PE. Partial

[1] This work was supported by the Ministry of Education of The Czech Republic - project VS 97 155 and project GA AV A2030801

triangulations are then pairwise joined. This solution achieved acceleration 4.09-5.07 on 8 PE.

[2] provided another attempt using incremental construction; however, speedup on many-processor computer was only about 2.

DT(P) can be also computed over Voronoi diagram or over 3D convex hull (recall that Voronoi diagram is a dual structure to Delaunay triangulation and 3D convex hull is over inversion or over projection onto paraboloid transformable into 2D Delaunay triangulation). Parallel solution for these problems can be found in [13, 18, 19, 10, 6, 21, 12, 15].

Very interesting is [7]. It solves *DT(P)* in 3D; however, the method can be used also in the plane. It provides an interesting and simple modification of D&C algorithm: instead of usual merge phase at the end of computation, it builds the boundary part of triangulation first. It has advantage of avoiding changes at the end of partial triangulations and avoiding mutual waiting of PEs for merge phase. However, practical results are not too good due to load imbalance (individual tasks are of very different size). Achieved acceleration was from 1.7 up to 3.35 for 2 to 16 PE. As for 2 PE the algorithm provides reasonable speedup (1.7) and is conceptually simple, it would be one of good possibilities for low number of processors. Parallel efficiency decreases from 85% for 2 PE to 20% for 16 PE (for 8000 points). Let's recall that these results pay for 3D triangulation.

[7] offers one more parallel solution – the bounding box of the given pointset is partitioned into rectangular regions, each of them is triangulated by incremental insertion algorithm. Each PE has access to the whole input point set; therefore, there are no errors on the boundary between regions. Triangles on the boundary between two regions are constructed by two PE and sequentially eliminated at the end. For n = 10 000, speedup achieved is from 1.74 to 19.2, efficiency from 87% to 30% for 2 to 64 PE.

D&C partition is in an interesting way solved in [4] (implementation and further improvement in [17]) with the use of 3D convex hull. Input points are divided into two groups by the vertical line in median in x-coordinate. Separating line is then substituted by Delaunay edges obtained over projection into paraboloid and convex hull approach. This a bit complicated work is outbalanced by the fact that no triangulation changes in merge phase are necessary. Although called 'practical', the algorithm seems a bit too complicated. Algorithm needs about twice as many floating point operations as Dwyer's serial D&C algorithm [11]. [17] combines this algorithm with efficient existing serial triangulation implementation [22]. Running time is equal to $(n \log n)(k1+k2*\log\lceil P \rceil)/P$ where P is the number of processors. From the resulting graphs, speedup about 1.6 to 1.7 for 2 PE, 3 to 4 for 8 PE can be derived. The implementation was tested on three different architectures and results were alike. Parallel efficiency (percentage of perfect speedup over good serial code) is for big data sets more than 50% while previous implementations of *DT(P)* had mostly less than 20%.

3. NEWLY PROPOSED SOLUTION

As described before, the proposed solution was developed for low-number-of-processors workstations. Therefore, efficiency and simplicity are preferred to scalability. Architecture with shared memory is supposed.

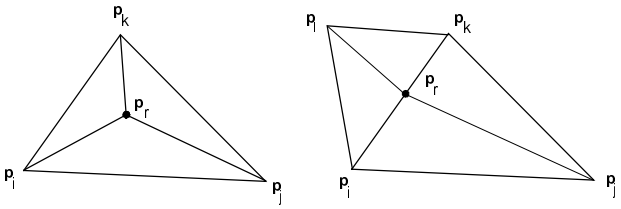
The incremental insertion algorithm is used as a basis for the parallel solution. Besides good qualities and behaviour of this algorithm in sequential version, we have for this decision reasons as follows:

Most of existing techniques are more or less based on D&C technique. Although it looks completely rational, *DT(P)* problem does not scale well as after partition by vertical and horizontal lines, size of the next task is about one half of previous which causes poor balancing. If partition is done into equally sized strips or areas, it is sensitive to uniformity of points. Changes of triangles on the boundary of subareas or elimination of twice computed triangles are unavoidable. Our idea is to let work all the PEs on the whole area, on one triangulation and to ensure that they do not make inconsistent transactions with the shared data structure (such as modifying the same triangle at the same time). Such a strategy could not work for higher number of processors where probability of collision could be high; however, for small number of PE and high number of input points collisions are not too probable and can be avoided.

Before we will explain the proposed parallel solution, let's describe serial version of randomized incremental insertion algorithm. For this presentation, we will mostly use description in [3].

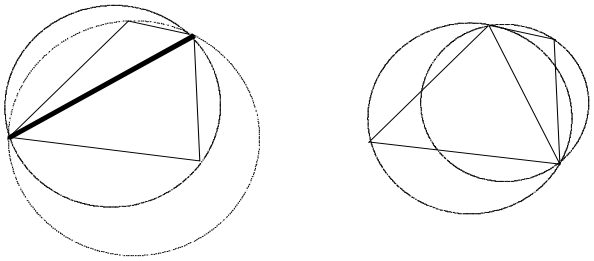
At the beginning of the triangulation, all points of P are enclosed into a big triangle. Then the points are inserted one at a time and the triangulation is updated with each insertion. At the end, the three added vertices of the starting big triangle and all edges and triangles incidating with them are removed. If the points were chosen "far enough away", result of this operation is the triangulation of the convex hull of P. Proper choice of these points will be explained later.

Let's describe what happens when a point is inserted. The triangle containing the point has to be found and subdivided into three new triangles, see Fig. 1a). The case when the new point lies in some existing edge must be distinguished and two neighbouring triangles have to be divided, see Fig. 1b). As the resulting triangulation after point insertion may not be Delaunay, edges of the new triangles have to be checked over empty circumcircle test. If the edge is not legal Delaunay's, it is flipped, see Fig. 2.



a) Point inside the triangle b) Point on the edge

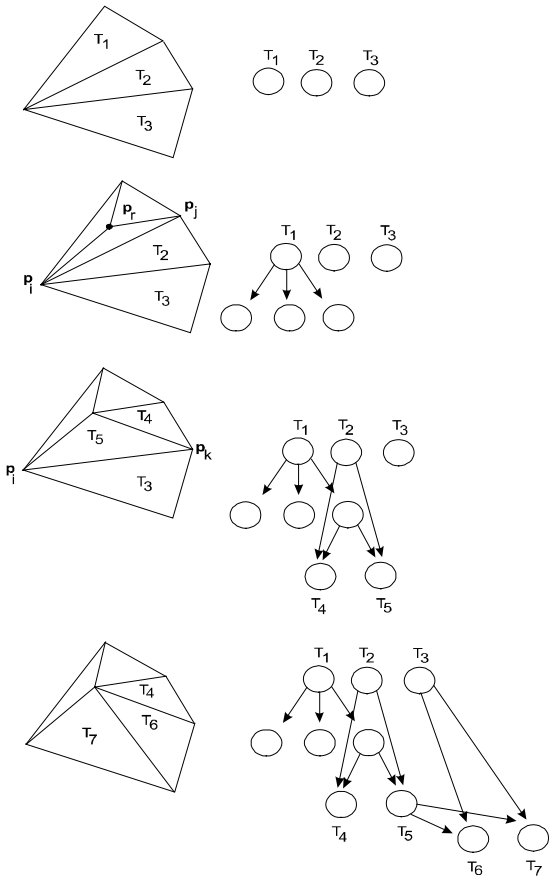
Figure 1: Point insertion



a) The thick edge fails the test b) After edge flip

Figure 2: Delaunay test of the edge

The changes may spread to all triangles whose circumcircles contain the newly inserted point. Instead of testing this condition, the triangle tests are propagated recursively as a wave from the newly inserted point until no change in the given direction is necessary, see example in Fig. 3a).



a) Flips b) Corresponding DAG

Figure 3: Propagation of edge flips

The whole algorithm is formulated in Fig. 4, Legalize_edge procedure in Fig. 5.

We postponed explanation how to pick the vertices for the starting triangle. The points should be far away not to influence the circumcircles of the triangles in *DT(P)*. Actually, for this purpose they should be "infinitely far away". However, if they are too far away, they may cause numerical inconsistences as their coordinates are very different from those of *P* which may lead to ill-conditioned determinants. We took over the idea from [3] to place the points in mutual position as in Fig. 6 (dark box is the minmax box of P). Recommended position of the points in [3] is (K,0), (0,K) and (-K,-K) where K = 3*size of the minmax box. According to the results of our implementation, such points may not be far enough and sometimes, after removal of outside triangles, some convex hull edge is missing. We use K = 10M instead.

procedure Delaunay_triangulation
Input: A set $P = \{\mathbf{p}_i, i = 0,1,...,n\text{-}1\}$ of n points in the plane
Output: A Delaunay triangulation of P $DT(P)$

1. **begin**
2. Find three outside points $\mathbf{p}_{-1}$, $\mathbf{p}_{-2}$, $\mathbf{p}_{-3}$;
3. $DT(P) := [\mathbf{p}_{-1}\mathbf{p}_{-2}\mathbf{p}_{-3}]$;
   // Initialize $DT(P)$ on the big triangle
4. Compute a random permutation of $\mathbf{p}_0$, $\mathbf{p}_1$, ..., $\mathbf{p}_{n-1}$ of P;
5. **for** r := 0 **to** n-1 **do**
6. **begin** // insert $\mathbf{p}_r$ into $DT(P)$
7. Locate the triangle $\mathbf{p}_i\mathbf{p}_j\mathbf{p}_k \in DT(P)$ containing $\mathbf{p}_r$;
8. **if** $\mathbf{p}_r$ lies inside the interior of $\mathbf{p}_i\mathbf{p}_j\mathbf{p}_k$ **then**
9. **begin**
10. Add edges from $\mathbf{p}_r$ to the vertices of $\mathbf{p}_i\mathbf{p}_j\mathbf{p}_k$ and subdivide $\mathbf{p}_i\mathbf{p}_j\mathbf{p}_k$ into three triangles;
11. Legalize_edge ($\mathbf{p}_r$,$\mathbf{p}_i\mathbf{p}_j$,$DT(P)$);
12. Legalize_edge ($\mathbf{p}_r$,$\mathbf{p}_j\mathbf{p}_k$,$DT(P)$);
13. Legalize_edge ($\mathbf{p}_r$,$\mathbf{p}_k\mathbf{p}_i$,$DT(P)$)
14. **end**
15. **else**
16. **begin** // $\mathbf{p}_r$ lies on the edge of the $\mathbf{p}_i\mathbf{p}_j\mathbf{p}_k$, say $\mathbf{p}_i\mathbf{p}_k$
17. Add edges from $\mathbf{p}_r$ to $\mathbf{p}_j$ and to $\mathbf{p}_l$, the third vertex of the other triangle sharing $\mathbf{p}_i\mathbf{p}_k$, and subdivide two triangles sharing $\mathbf{p}_i\mathbf{p}_k$ into four triangles;
18. Legalize_edge ($\mathbf{p}_r$,$\mathbf{p}_i\mathbf{p}_j$,$DT(P)$);
19. Legalize_edge ($\mathbf{p}_r$,$\mathbf{p}_j\mathbf{p}_k$,$DT(P)$);
20. Legalize_edge ($\mathbf{p}_r$,$\mathbf{p}_k\mathbf{p}_l$,$DT(P)$);
21. Legalize_edge ($\mathbf{p}_r$,$\mathbf{p}_l\mathbf{p}_i$,$DT(P)$)
22. **end**
23. **end;**
24. Remove $\mathbf{p}_{-1}$, $\mathbf{p}_{-2}$, $\mathbf{p}_{-3}$ and all the incidating triangles and edges from $DT(P)$
25. **end**

Figure 4: Delaunay triangulation by randomized incremental insertion

procedure Legalize_edge ($\mathbf{p}_r$,$\mathbf{p}_i\mathbf{p}_j$,T);
Input: The inserted point $\mathbf{p}_r$,
the flipped edge $\mathbf{p}_i\mathbf{p}_j$,
the triangulation T
Output: The modified triangulation T

1. **begin**
2. **if** $\mathbf{p}_i\mathbf{p}_j$ is illegal **then**
3. **begin**
   // $\mathbf{p}_i\mathbf{p}_j\mathbf{p}_k$ is the triangle sharing the edge $\mathbf{p}_i\mathbf{p}_j$ with $\mathbf{p}_r\mathbf{p}_i\mathbf{p}_j$
   // flip the edge
4. Replace $\mathbf{p}_i\mathbf{p}_j$ by the edge $\mathbf{p}_r\mathbf{p}_k$
5. Legalize_edge ($\mathbf{p}_r$,$\mathbf{p}_i\mathbf{p}_k$,T)
6. Legalize_edge ($\mathbf{p}_r$,$\mathbf{p}_k\mathbf{p}_j$,T)
7. **end**
8. **end**

Figure 5: Legalize_edge procedure for algorithm in Fig. 4

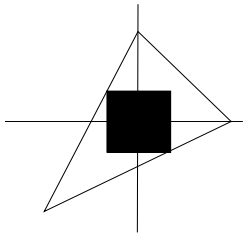


Figure 6: To the choice of the starting triangle

The key question in this algorithm is quick detection of the triangle containing the inserted point. We used an approach from [3] where triangles are kept in directed acyclic graph (DAG) – in this case, it is a tree with the history of insertion. Each node corresponds to a triangle; when the triangle is subdivided or flipped, the node gets sons corresponding to newly created triangles, see Fig. 3a), b). Thus DAG contains the current triangulation in leaves, "big" triangle being the root. Location of the inserted point is in this data structure possible in *O(log n)* expected and *O(n)* worst time (worst time happens when the tree is unbalanced – due to randomization, such a situation is highly improbable.)

The algorithm is not worst-case optimal as it has $O(n^2)$ time complexity in the worst case; however, its expected complexity is *O(n log n)*. Memory complexity is *O(n)* [16].

We implemented this algorithm in Delphi 3 on Windows NT. Results for uniform, normal, cluster and grid data, average for 5 data sets, are shown in Table 1 and Fig. 7. Due to randomization, non-uniform data are not a problem. (It can be seen in the graph that for cluster or grid type of data, runtimes are even lower.) After some work with this algorithm, we must appreciate its relative robustness in comparison with some other solutions - numerical inaccuracy may cause that some triangles whose vertices are nearly in singular position are not Delaunay, but the triangulation as a whole is all the time consistent, no gaps, no mutual covers of triangles. When we worked with incremental *construction* type of algorithm (to the given edge, the nearest point fulfilling Delaunay condition is searched), such artifacts were quite often for big number of points. Also runtimes are relatively good, although in references [24, 17 etc.] Dwyer's D&C type of algorithm [11] is evaluated to be the fastest.

n	**t_unif** [s]	**t_norm** [s]	**t_grid** [s]	**t_clus** [s]
100	0.006	0.012	0.003	0.010
500	0.031	0.041	0.031	0.040
1000	0.090	0.087	0.069	0.088
5000	0.516	0.516	0.403	0.522
10 000	1.112	1.131	0.891	1.113
50 000	6.544	6.594	5.403	6.506
100 000	14.000	14.122	11.644	13.653
500 000	86.556	86.000	71.140	77.812
1 000 000	193.747	193.153	151.003	158.790

Table 1: Serial algorithm runtimes in sec for various input points distribution: uniform, normal, points on the regular grid, points in clusters. Computed on Dell Precision 410 (2x Pentium III, 500 MHz, 1 GB RAM)

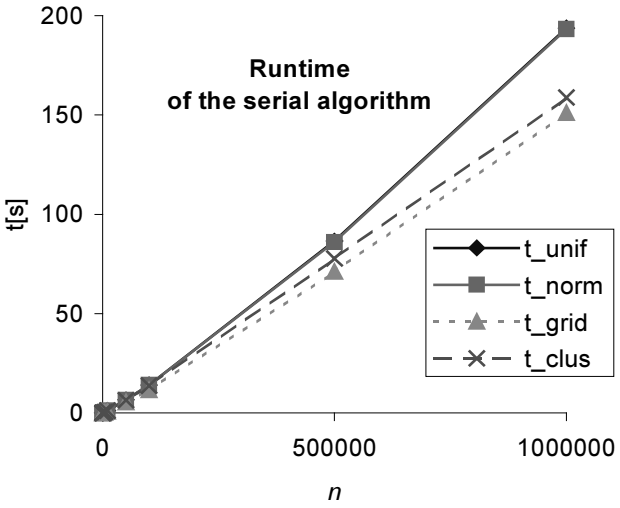


Figure 7: Runtimes of the serial randomized incremental insertion algorithm for various types of input data (t_unif and t_norm are not distinguishable)

Figure 8 shows how the runtime is spent percentually in main parts of the algorithm. It can be seen that most of the time is needed for point location. The shortest time is 50% of the total for $n = 100$, the biggest value is 64.5% for $n = 500000$, average being 58.9%). Therefore, parallel algorithm oriented on speeding up of the location part may expect up to 60% acceleration.

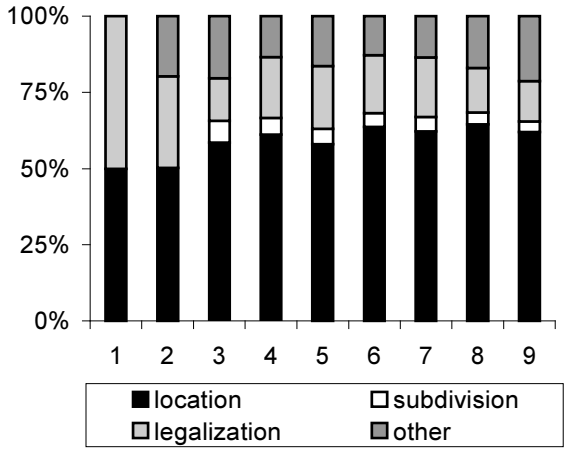


Figure 8: Percentage of the runtime occupied by the main parts of the serial algorithm for uniform data, $n = 100$, 500, 1000, 5000, 10 000, 50 000, 100 000, 500 000, 1 000 000

Let's proceed with the possibilities to parallelize this method. As stated before, we expect an architecture with small number of PEs and shared memory. We will keep the whole triangulation in one

shared DAG. Computation will be partitioned among **threads**[2]. One PE will usually run one thread. (More then one thread on one PE is possible but not too useful.) The input point set P will be divided among threads and each thread will insert its subset of points into the triangulation. The consistency of the triangulation has to be protected to prevent the situation that the same triangle is accessed at the same time by more than one thread. There are three possible ways:

1. Pessimistic method

At the beginning, each thread obtains a set of input points to insert them independently. Threads are allowed to locate the inserted point to the last but one level of the DAG - up to the node whose immediate sons are leaves. Last step of the location – location in DAG leaves, triangle subdivision and edge legalization are done in critical section, i.e., only one thread has access at the same time.

2. Optimistic method

Threads have the same job as in pessimistic method; however, DAG is not protected as a whole but only the triangles that should be subdivided or edge-flipped are locked. If a thread locates the point to the triangle which is locked by someone else it has to wait in critical section until the operation is finished and the triangle is unlocked again.

This method may cause deadlock by mutual waiting of threads. The problem can be solved by deadlock detection and by priorities – the thread with lower priority gives up the operation.

3. Batch method

Threads only locate the point in DAG, they do not subdivide triangles and do not legalize edges. Last level of search, subdivision and legalization are done by one specialized thread. Tasks are stored in shared memory.

What are pros and cons of these approaches ? Pessimistic method is very simple and can be done fully with the tools present in classes in the libraries available in today's compilers (in our case, Delphi and Visual C++). Its disadvantage is that it is too careful – it disables modification of all the triangles, although most of them is free. On the other hand, location is the main time consumer of the incremental insertion, recall Fig. 8, so even pipelining of search part may bring some speedup.

Batch method is the most safe as all DAG modifications are done by one process. Problem can be that this method expects much longer time for search than for DAG modification. If it is not so, the queue of tasks will be choked and locating threads will have to wait for the specialized one. Another problem is proper size of the queue; with increasing size, probability of waiting is lower at the beginning but at the same time, the task in queue may become partly inactual – i.e., it may want to work with the triangle which is not the leaf any more. Such a task means longer work for the specialized thread and longer waiting time for other threads.

[2] When an application is run, it is loaded into memory ready for execution. At this point it becomes a process containing one or more threads that contain the data, code and other system resources for the program. A thread executes one part of an application and is allocated CPU time by the operating system. All threads of a process share the same address space and can access the process's global variables [5].

Optimistic method looks most promising and most difficult to program as it needs to insert special semaphors into DAG nodes and deadlock handling into process administration.

Of these three possibilities, we chose the pessimistic method as most simple to start with. According to experience with this solution, we can continue in the future to more difficult, but probably more efficient optimistic method.

Algorithm of the pessimistic method is in more details given in Fig. 10.

Master thread:

Input: A set $P = \{\mathbf{p}_i, i = 0,1,...,n\text{-}1\}$ of n points in the plane
Output: A Delaunay triangulation of P $DT(P)$

```
1.  begin
2.    Find three outside points p_-1, p_-2 , p_-3;
3.    DT(P) := [p_-1 p_-2 p_-3];
                // Initialize DT(P) on the big triangle
4.    Compute a random permutation of p_0, p_1, ..., p_n-1 of P;
5.    Subdivide P into m subsets where m is the total number
                of worker threads;
6.    Start all worker threads and wait until finished;
7.    Remove p_-1, p_-2, p_-3 and all the incidating triangles
                and edges from DT(P)

8.  end
```

Worker thread:

Input: A set $Pt = \{\mathbf{p}_i, i = 0,1,...,n_m\text{-}1\}$ of n_m points in the plane, $P_t \subset P$
Output: Modifies the shared $DT(P)$

```
1.  begin
2.    for r := 0 to n_m-1 do
3.      begin   // insert p_r into DT(P)
4.        Start to locate in DAG the triangle
                on the level of leaves' parents containing p_r ;
5.        if any thread working with leaves exists then wait;
6.        Enter critical section;  // start of work with leaves
7.          Finish location on the leaf level
                  and find the triangle p_i p_j p_k ∈ DT(P)
                  containing p_r;
8.          Subdivision and legalization; // see Figs. 4, 5
9.        Leave critical section;  // end of work with leaves
10.     end
11. end
```

Figure 10: Pessimistic method of parallel incremental Delaunay insertion (there is one master thread and one or more worker threads in the system)

4. EXPERIMENTS AND RESULTS

Suggested parallel solution was implemented in MS Visual C++ v. 6.0, using serial $DT(P)$ incremental algorithm implemented in Delphi v. 3.0. Tests were run on data sets with $n = 100$ up to 1 000 000. From various input points distributions, uniform and grid were measured as the most representative (recall Fig. 7). Randomized input sets were divided into subsets of the same size

for each thread. As the results were very much alike, we will show only uniform data results. Tests were run on Dell Precision 410 (2x Pentium III, 500 MHz, 1 GB RAM), Dell Power Edge 8450 (8x Pentium III, cache 2MB, 550 MHz, 2GB RAM) and Hewlett Packard HP XU 6 (2x Pentium Pro, 200 MHz, 128 MB).

Table 2 presents runtimes in sec measured on Dell 8 processor computer. Times were obtained as average of up to 5 data sets. Fig. 11 shows dependence of the total runtime on the number of worker threads, parametrized by the number of points. Fig. 12 shows dependence of the time for points insertion on the number of threads, parametrized by the number of points. Figure 13 presents both time measures for n = 500 000.

Table 3 are results of measurement on Dell 2 processor, averaged over 5 data sets. Fig. 14 shows speedup for this type of computer.

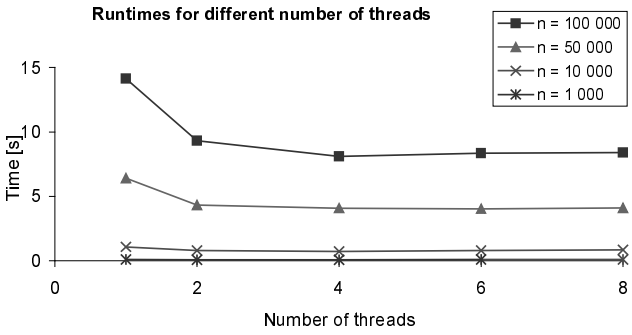


Figure 11: Dependence of the runtime on the number of threads, parametrized by the number of points.

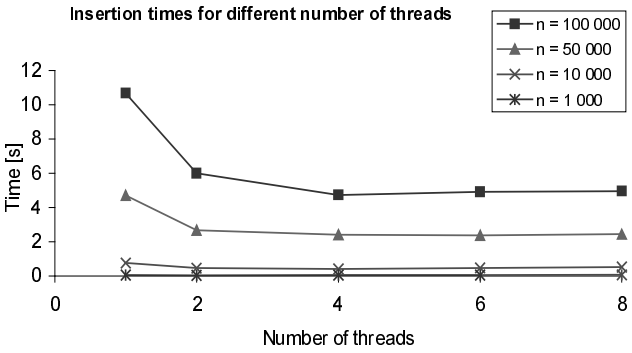


Figure 12: Dependence of the time for insertion on the number of threads, parametrized by the number of points.

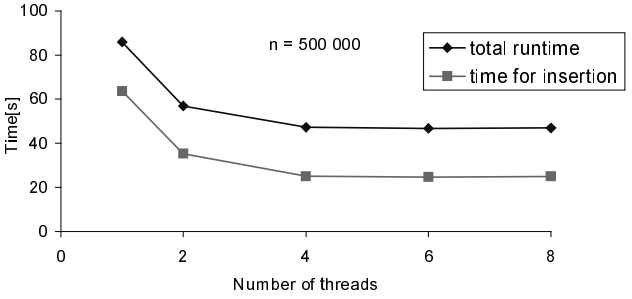


Figure 13: Dependence of the runtime and of the time for insertion on the number of threads, for n = 500 000.

n	No of threads	Total runtime	Time for insertion	Time in critical section	Waiting time
100	1	0.007	0.005	0.002	0.000
100	2	0.007	0.005	0.001	0.001
1000	1	0.076	0.063	0.024	0.001
1000	2	0.058	0.044	0.017	0.006
10 000	1	1.048	0.898	0.283	0.014
10 000	2	0.683	0.533	0.164	0.052
50 000	1	6.200	5.364	1.389	0.702
50 000	2	3.960	3.117	0.819	0.214
100 000	1	13.750	11.860	2.823	0.142
100 000	2	8.749	6.849	1.647	0.409
500 000	1	87.000	68.000	14.643	0.714
500 000	2	58.276	38.499	8.359	2.027
1 000 000	1	200.400	143.000	29.807	1.441
1 000 000	2	135.400	78.600	16.893	4.130

Table 3: Runtimes [s] for uniform data, on 2 processor computer.

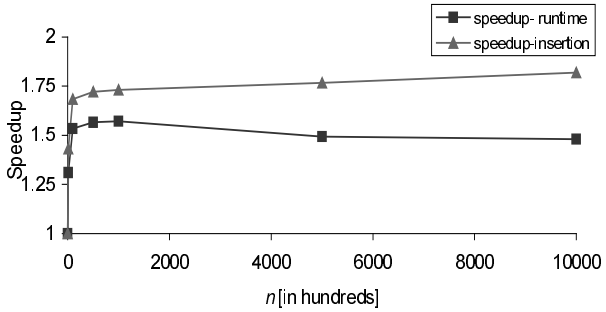


Figure 14: Speedup achieved on 2 processor computer computed as the rate of runtime for one thread divided by the runtime for two threads (data from Table 3)

From the results can be seen that for small number of points (1000 and less), there is no or unsubstantial improvement. For higher n, speedup achieved is from 26% (n = 10 000, 8 threads) to 84% (n = 500 000, 6 threads). Generally, speedup for higher number of points is better. However, Figs. 11-13 show that most improvement is obtained for 2 threads. If n is 100 000 or more, also 4 threads bring some further improvement but not so substantial as in 2-threads case. For higher number of threads, probability of collisions in critical section are higher and no further substantial speedup is achieved, more time is spent by waiting.

Conclusion can be done that the supposed pessimistic method can be successfully used, first of all, for 2-processor workstations. Higher number of processors brings further, but non-substantial speedup. The method is especially suitable for high number of points (hundreds of thousands, millions) which is positive feature. For higher degree of parallelism, optimistic method is supposed to be implemented and tried in the future.

n	No of threads	Total runtime	Time for insertion	Time in critical section	Waiting time
100	1	0.009	0.005	0.002	0.000
100	2	0.010	0.006	0.002	0.001
100	4	0.012	0.006	0.001	0.001
100	6	0.015	0.007	0.001	0.001
100	8	0.017	0.008	0.001	0.001
1000	1	0.095	0.059	0.024	0.001
1000	2	0.077	0.042	0.016	0.007
1000	4	0.081	0.046	0.010	0.025
1000	6	0.092	0.057	0.008	0.038
1000	8	0.097	0.061	0.006	0.044
10 000	1	1.079	0.760	0.231	0.012
10 000	2	0.808	0.456	0.143	0.044
10 000	4	0.726	0.405	0.089	0.178
10 000	6	0.803	0.474	0.069	0.312
10 000	8	0.853	0.522	0.057	0.389
50 000	1	6.413	4.709	1.262	0.062
50 000	2	4.309	2.678	0.724	0.180
50 000	4	4.093	2.417	0.533	0.973
50 000	6	4.045	2.370	0.351	1.405
50 000	8	4.110	2.438	0.271	1.692
100 000	1	14.124	10.669	2.619	0.124
100 000	2	9.305	5.994	1.618	0.409
100 000	4	8.091	4.729	1.045	1.671
100 000	6	8.350	4.923	0.726	2.789
100 000	8	8.379	4.959	0.548	3.318
500 000	1	85.899	63.678	14.126	0.630
500 000	2	56.986	35.348	7.905	1.630
500 000	4	47.285	25.043	5.411	6.367
500 000	6	46.746	24.703	3.645	12.156
500 000	8	46.938	24.960	2.750	15.498

Table 2: Runtimes [s] for uniform data, on 8 processor computer

5. CONCLUSION

The paper presents very simple method of parallel Delaunay triangulation construction. The method was developed for workstations with 2 or more processors and achieved best speedup for two processors, results for higher degree of parallelism are not too persuading. Further improvement of the method decreasing mutual waiting (the so-called optimistic method) was suggested and is expected to be implemented and measured in the future.

6. REFERENCES

[1] Aggarwal A., Chazelle B., Guibas L., O'Dunlaig C, Yap C.: Parallel Computational Geometry, *Algorithmica*, Vol.**3**, No.**3**, pp.293-327, 1988

[2] Beichl I., Sullivan F.: Parallelizing Computational Geometry: First Steps, *SIAM News*, No.**6**, Vol**. 24**, pp.1-17, 1991

[3] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf: *Computational Geometry. Algorithms and Applications*, Springer-Verlag Berlin Heidelberg, 1997

[4] Blelloch G.E., Miller G.L., Talmor D.: Developing a Practical projection-Based Parallel Delaunay Algorithm, *Proc. of the 12th Annual Symposium on Comput. Geometry*, ACM, 1996

[5] Borland Delphi v.3.0 – on-line help, 1997

[6] Chandrasekhar N., Franklin W.R.: *A Fast Practical Parallel Convex Hull Algorithm*, TR Electrical, Computer and Systems Engineering Department, Rensselaer Polytechnic Institute, Troy, NY, 1989

[7] Cignoni P., Montani, C., Perego, R., Scopigno, R.: Parallel 3D Delaunay Triangulation, *Proc. of Eurographics'93*, pp. C129-C142

[8] Cole R., Goodrich M.T., O'Dunlaig C.: Merging Free Trees in Parallel for Efficient Voronoi Diagram Construction, *International Colloquium on Automata Languages and Programming*, pp. 32-45, 1990

[9] Davy J.R., Dew P.M.: A Note on Improving the Performance of Delaunay Triangulation, In: *Proceedings of CGI'89*, pp. 209-226, 1989

[10] Day A.M.: Parallel Implementation of 3D Convex Hull Algorithm. *CAD*, Vol. **23**, No. **3**, pp.177-188, 1991

[11] Dwyer R.A.: A Simple Divide-and-conquer Algorithm for Constructing Delaunay Triangulation in *O(n log log n)* expected time. In *Proc. of the 2nd Annual Symposium on Comp. Geom*., pp. 276-284, ACM, 1986

[12] Edelsbrunner H., Shi W.: An $O(n \log^2 h)$ time algorithm for the three-dimensional convex hull problem. *SIAM J.Computing*,, Vol. **20**, pp. 259-277, 1991

[13] Evans D.J., Stojmenovic I.: On Parallel Computation of Voronoi Diagrams, *Parallel computing*, Vol.**12**, pp. 121-125, 1989

[14] T.-P. Fang, L. A. Piegl: Delaunay Triangulation Using a Uniform Grid, *IEEE CompGraph & Appl*, May 1993, pp. 36-47

[15] Ghouse M., Goodrich M. T.: In-place Techniques for Parallel Convex Hull Algorithms, *Proc. of the 3rd ACM Symposium on Parallel Algorithms and Architectures*, 1991

[16] L. J. Guibas, D. E. Knuth, M. Sharir: Randomized Incremental Construction of Delaunay and Voronoi Diagrams, *Algorithmica*, Vol. **7**, 1992, pp.381-413

[17] Hardwick J.C.: Implementation and Evaluation of an Efficient Parallel Delaunay Triangulation Algorithm, *9th Annual Symposium on Parallel Algorithms and Architectures*, pp. 22-25, 1997

[18] Jeong C.S.: Parallel Voronoi Diagram in L_1 (L_∞) on a Mesh-connected Computer, *Parallel Computing*, Vol. **17**, pp. 241-252, 1991

[19] Jeong C.S. : An Improved Parallel Algorithm for Constructing Voronoi Diagram on a Mesh-connected Computer, *Parallel Computing*, Vol. **17**, pp. 505-514, 1991

[20] Merriam M.L.: Parallel Implementation of an Algorithm for Delaunay Triangulation, In *1-st European Fluid Dynamics Conference*, pp. 907-912, 1992

[21] Reif J., Sen S.: Polling: A New Randomized Sampling Technique for Computational Geometry, *Proc. of the 21st Annual Symposium on Theory of Computing*, 1989

[22] J. R. Schewchuk: Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator, In: Ming C. Lin, Dinesh Manocha, Eds.: *Applied Computational Geometry Towards Geometric Engineering*, Vol. 1148 of Lecture Notes in Computer Scinece, pp. 203-222, Springer-Verlag, May 1996

[23] Su P.*: Efficient Parallel Algorithms for Closest Point problems*, PhD thesis, Darthmouth College, 1994, PCS-TR94-238

[24] Su P., Drysdale R.L.S.: A Comparison of Sequential Delaunay Triangulation Algorithms, In*: Proc. of the 11-th Annual Symposium on Computational Geometry*, pp. 61-70, ACM, June 1995

[25] Teng Y.A., Sullivan F., Beichl I., Puppo E.: A Data Parallel Algorithm for Three-dimensional Delaunay Triangulation and Its Implementation, *SuperComputing '93*, 112-121, 1993

[26] Vemuri B.C., Varadajaran R., Mayya N.: An Efficient Expected Time Parallel Algorithm for Voronoi Construction, *Proc. of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 392-400, 1992

ACKNOWLEDGEMENT

We would like to thank to Dell Computer, Czech Rep., for allowing us to experiment on their 8-processor computer: Dell Power Edge 8450 – 8x Pentium III, cache 2MB, 550 MHz, 2GB RAM.

About the authors

Ivana Kolingerová is an associate professor on the Department of computer science and engineering of the University of West Bohemia, Pilsen. Josef Kohout is an undergraduate students of informatics and computer science.

E-mail: kolinger@kiv.zcu.cz, besoft@students.zcu.cz