Available online at www.sciencedirect.com
SCIENCE DIRECT®
Computers & Graphics 28 (2004) 703–718
COMPUTERS & GRAPHICS
www.elsevier.com/locate/cag

Technical Section

Practically oriented parallel Delaunay triangulation in E^2 for computers with shared memory ☆

Josef Kohout[a,*], Ivana Kolingerová[a], Jiří Žára[b]

[a] Department of Computer Science and Engineering, Centre of Computer Graphics and Data Visualisation, University of West Bohemia, Univerzitní 8, 306 14 Plzeň, Czech Republic

[b] Department of Computer Science and Engineering, Czech Technical University in Prague, Karlovo nám. 13,121 35 Praha 2, Czech Republic

Abstract

This paper describes two simple and efficient parallel algorithms for the construction of the Delaunay triangulation ($DT(S)$) in E^2 by randomized incremental insertion. The construction of the $DT(S)$ is one of the fundamental problems in computer graphics. The proposed algorithms are designed for parallel systems several processors and with shared memory. Such a hardware configuration (especially the case with two-processors) became widely available in the last few years thanks to low prices at present, but there is still a lack of parallel algorithms that are simple to implement and efficient enough to be an attractive alternative to existing serial algorithms. We have implemented both new algorithms in C++ and tested them on workstations with up to four processors. Thanks to memory caching we noticed several times even super-linear speed-up compared with the reference sequential algorithm.

Keywords: Parallel Delaunay triangulation; Incremental insertion; Computational geometry

1. Introduction

Triangulation $T(S)$ of a set of points S in E^2 is a set of triangles such that:

- A point $p \in E^2$ is a vertex of a triangle from $T(S)$ if and only if p belongs to S; i.e. the vertices of the triangles are some points from the input set.
- The intersection of two triangles is either empty or it is a shared edge or a shared vertex; i.e. the triangles do not overlap, and there is no vertex lying on an edge of another triangle.
- The set $T(S)$ is maximal: there is no triangle that can be added into $T(S)$ without violating previous rules; i.e. the union of triangles and the convex polygon formed by a convex hull $CH(S)$ are the same object.

Delaunay triangulation was proposed by the Russian scientist Boris N. Delone [1,2]. However, as his original papers are not written in English and their translations are usually rather complex, we would recommend Radke's paper [3] for a detailed description of the Delaunay triangulation. Further information can also be found in [4].

Delaunay triangulation $DT(S)$ of a set of points S in E^2 is a triangulation such that the circum-circle of any triangle does not contain any other point of S in its interior; this is called the empty circum-circle criterion.

The basic properties of the $DT(S)$ are as follows:

- It contains the most equiangular triangles of all possible triangulations (i.e. it limits the number of very narrow triangles that may cause problems in further processing).
- If no four points lie on a circle, the $DT(S)$ is unique.
- The boundary of the $DT(S)$ is a convex hull of S.
- It can be computed in $O(N \log(N))$ time in the worst case (whose N is the number of points to triangulate).

☆ This work was supported by the Ministry of Education of The Czech Republic—Project MSM 235 200 005.

*Corresponding author.
E-mail address: besoft@kiv.zcu.cz (J. Kohout).

0097-8493/$ - see front matter © 2004 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cag.2004.06.009

However, algorithms with $O(N)$ expected time also exist.

Due to these good properties, Delaunay triangulation is used in many areas such as terrain modeling (GIS) [5], scientific data visualization [6–9] and interpolation [10], robotics, pattern recognition [11,12], meshing for finite element methods (FEM) [13–15], natural sciences [16,17], computer graphics and multimedia [18,19], etc.

Although the Delaunay triangulation can be computed in $O(N\log(N))$ time, it still consumes a lot of time especially for larger N. Modern computer architectures allow us to compute Delaunay triangulation with thousands of points by a sequential algorithm in reasonable time. However, current applications often need to work with millions of points. Fortunately, parallel architectures, especially multiprocessors with several processors and shared memory, have come into consideration in the last few years due to their low prices. In such cases, a parallel algorithm is useful and desirable. Quite a large set of parallel algorithms exists, however, they were designed in times when parallel architectures, with hundreds of processors, dominated the research area and thus stress has been placed on the scalability rather than on the simplicity of such algorithms. Therefore, we have developed several parallel methods suitable for architectures with several processors and shared memory. Such algorithms are easier to understand and can be simply implemented even by computer graphics people without a deep knowledge of parallel computational techniques.

This paper is structured as follows: the next section gives a survey of existing parallel solutions for the construction of the Delaunay triangulation. In Section 3, we describe the chosen sequential algorithm and the proposed parallel modifications in detail. Section 4 presents the experiments and results and Section 5 concludes the paper.

2. Construction of the Delaunay triangulation

Many sequential algorithms for the construction of the Delaunay triangulation exist. We classify them according to Cignoni et al. [20] into several categories:

- Local improvement algorithms—starting with an arbitrary triangulation, these algorithms locally modify the edges of adjacent triangles according to the circum-circle criterion.
- Incremental insertion algorithms—starting with an auxiliary triangle that contains all points in its interior, these algorithms insert the points in S one at a time: the triangle containing the point to be inserted is subdivided and then the circum-circle criterion is tested recursively on all triangles adjacent to the new ones and if necessary, their edges are flipped [4,21].
- Incremental construction algorithms—the Delaunay triangulation is constructed by successively building triangles whose circum-circles contain no points in S. We can include also Fortune's sweeping algorithm into this category [20,22,23].
- Higher dimensional embedding algorithms—these algorithms transform the points into E^3 and then compute the convex hull of the transformed points, the Delaunay triangulation is obtained by projecting the resulting convex hull back into E^2 [24].
- Divide and conquer (D&C) algorithms—these algorithms are based on recursive partitioning and local triangulation of the point set, and then on a merging phase where the resulting triangulations are joined. Let us note that the recursion usually stops when the size of the point set matches some given threshold. Local triangulation is then constructed by an algorithm belonging to any of the previous categories [20,25,26].

Given that the parallelization of D&C algorithms appears to be straightforward, it is not surprising that these algorithms have been more frequently parallelized. However, naive D&C algorithms suffer from two drawbacks. First, their merge phase is quite complex. This phase involves the building of the edges among triangles from both triangulations. Simple connection usually does not satisfy the Delaunay criterion and, therefore, some corrections have to be done. In the worst case these corrections spread over the whole triangulation. The second drawback occurs when we try to parallelize such algorithms: the merging of two sets is limited to just one processing element (PE). It thus negatively influences the overall efficiency of the algorithm. The algorithm by Aggarwal et al. [27] demonstrates these drawbacks.

Cignoni et al. [20] present two algorithms. The DeWall algorithm uses a new approach to the D&C strategy. It first constructs the triangles at the joint and then the triangulation of both parts. No merge phase is required. The triangles at the joint as well as both triangulations are made by the principle of incremental construction. The authors present the results of their algorithm for $DT(S)$ in E^3. For example, speed-up 1.70–3.35 for 2–16 PEs was noticed when uniform data sets with 8000 points were tested.

The other algorithm from [20] is called InCode. It subdivides the plane into k rectangular areas. Each area is assigned to one processor as well as the whole set of the input points. The processor constructs triangles that have at least one vertex in the area, thus the triangles at the area's boundaries are created by more processors. The merge phase is simple; it involves only the removal of redundant triangles. This redundancy, indeed, affects

the efficiency of the algorithm. For the problem of the $DT(S)$ in E^3 the authors present, for example, speed-up 1.79–19.01 for 2–64 PEs using on nCUBE 2 system model 6410. Uniform data sets with 20,000 points were used for the test. Hardwick [28], however, notes that the InCode algorithm is about 10 times slower with non-uniform data sets.

Chen et al. [29] uses a similar approach. Each processor triangulates its part of the input set by the fastest sequential algorithm [25]. Moreover, it constructs an "interface" at each boundary by the principle of incremental construction. An interface is a set of triangles that run through the area's boundaries. It is clear that we have two interfaces at the same boundary. These two interfaces are merged together and the resulting joint is combined with both triangulations. Except for this merge phase, all other phases can be processed in parallel. The time needed for the merge phase is, however, negligible in comparison to other phases (thanks to the interfaces). Therefore, the algorithm achieved outstanding speed-up. For example, the tested uniform data sets with 96K points achieved speed-up 1.57–4.95 for 2–8 PEs with an IBM SP2 with High Performance Fortran.

Hardwick [28] chooses another approach. Input points are subdivided into two groups by the orthogonal line that goes through the median in x or y coordinates. A paraboloid in E^3 related to this line is found. Then the algorithm transforms all points onto the paraboloid, where the lower convex hull is found and the projection of the resulting lower convex hull into the plane gives a set of line segments. These line segments form a joint. Both groups of input points are triangulated by Dwyer's algorithm [25]. As no merge phase is required and the described subdivision of input points can be solved in parallel, the algorithm achieves a very good speed-up about 1.8–5.8 for 2–8 PEs with a SGI Power Challenge with shared memory for the uniform data sets with 128K points. Let us note that this algorithm (sometimes wrongly categorized as a higher dimensional embedding algorithm), is extremely complex.

Lee et al. [30] combine Hardwick's approach [28] with the InCode algorithm [20]. The small variation is that their algorithm does not recursively subdivide the input points into two groups via a median line but subdivides them immediately (in one step) into several slabs. The authors claim that such partitioning leads to a simpler algorithm. According to published graphs it is also evident that a better speed-up is achieved. Their experiments were done using an INMOS TRAM network with 32 T800 processors. Their algorithm achieves a speed-up 1.36–12.5 for 2–32 PEs and uniform data. Better performance of the algorithm is presented for cluster data with a speed-up about 16.9 for 32 PEs. Let us note that objective evaluation of this algorithm is impossible because the authors have not published the numbers of points in their data sets.

All previously described solutions use the D&C strategy, most of them in addition to the principle of incremental construction. The following methods avoid the D&C approach.

The first of these appears in Lee [31] and is based on incremental construction. The presented algorithm is useful for massive parallelization. Each processor has a set P of several points to be processed (ideal loading is one point per PE) and the whole set S of input points for tests. For each point this algorithm looks up the point nearest to the currently processed one and constructs the edge between them. Afterwards the first PE collects all computed edges, removes redundant edges and distributes the computed edges among processors. It finds the two nearest points for each received edge (one point on the left half-plane, one on the right half-plane) and constructs two triangles. This second stage is repeated until no new elements are created. As Lee used an Intel Paragon for his experiments, we can expect that the overhead for communication is significantly reduced. Unfortunately, the author does not present the results of his experiments.

Solutions based on incremental insertions also exist, although they are rare. In our previous work [32], we suggested two methods, both suitable for architectures with several processors and shared memory. They work with a shared DAG (Directed Acyclic Graph) structure, i.e. shared triangulation. The DAG is modified simultaneously by more PEs; each PE inserts its subset of points. The maximal speed-up achieved using a Dell PowerEdge 8450 for the tested uniform data sets (with up to 1 million of points) was approximately 1.73–5.84 times for 2–8 PEs.

Chrisochoides et al. [33,34] parallelize the well-known Bowyer–Watson's algorithm that is based on incremental insertion with cavity retriangulation [35]. First, the sequential algorithm creates a coarse triangulation of a subset of points. The triangles are partitioned among k processors and the parallel insertion begins. Boundaries between areas are formed by some edges of the triangles and may change during triangulation. When more processors share a cavity, complex synchronization is necessary. In [34] it is shown that the speed-up is nearly linear due to a heuristic used to balance the load of the processors and to minimize the length of the boundary, but they offer neither proof nor experimental evidence to substantiate this statement.

In some areas of use, it is necessary to additionally insert new points into already existing $DT(S)$ to obtain better shapes for the triangles. The candidates for such points are usually the centers of circum-circles of the triangles, the algorithm has to decide whether to use such a point or not. A very popular sequential solution was proposed by Chew [36]. Okusanya et al. [6]

developed parallel version of this algorithm. The primary triangulation is partitioned among processors. If the insertion of a new point also affects remote triangles (i.e. triangles not physically present at the current processor), PE has to send a message to all participants to obtain these remote triangles. Concurrent processors locate remote triangles in a special tree structure, lock them for the initiator to avoid inconsistencies and send them in a compressed form to the initiator. For communication, MPI or PVM is used. When the initiator completes its operation, all remote triangles are unlocked. However, in case that any PE cannot grant an exclusive access to the requested triangles for the initiator, it denies the request and the initiator has to give up insertion of this point and focus on another point. Although authors also implemented some load balancing, the achieved speed-up is only 1.6–3.0 times for 2–8 PEs using an IBM SP2 for a uniform data set of 1,000,000 points. This is probably caused by the time-consuming communication among PEs. Let us note that authors use a similar strategy also in E^3 [7]—their speed-up is roughly 1.2–2.3 times for 2–8 PEs for 144,600 points. In our opinion, the achieved speed-up is quite low.

Better results could probably be achieved by the approach of Spielman et al. [37] as it is able to determine quickly which points can be inserted without any synchronization. The authors present their algorithms for E^2 and E^3, and prove correctness of these algorithms. However, there is no experimental section in their paper.

Puppo et al. [38] present another parallel algorithm based on incremental insertion. Their algorithm is not devoted to Delaunay triangulation only but rather to building a triangulated irregular network from a dense regular grid of points. It selectively inserts points into the $DT(S)$. Each input point, as well as each triangle, is allotted to one virtual PE. At the beginning, only two triangles containing the four corners of the domain exist. Then, for every yet unused point, the vertical distance between the point and its approximation is computed and for every triangle, the point with the maximum distance is chosen to be inserted. Conflicts between PEs (e.g., when the point to be inserted lies on an edge) are solved by a priority rule—a higher priority is given to the triangles whose inserted points lie further from their approximation. Analogous mutual exclusion must be solved after each round of point insertion when Delaunay triangulation is to be recovered by parallel edge swapping. The algorithm was implemented on a Connection Machine CM-2 with 16K processors, compared with a serial implementation on Sun SPARC1 and tested with up to 512^2 points. The speed-up was up to 80 times for 16K points. The highest speed-up was achieved with the smallest allowed approximation error because in such a case, more triangles are necessary and the load balance improves.

In this section, we have given a survey of some existing parallel algorithms. All described algorithms, perhaps excluding Lee's approach [31], can be used with some modifications for a hardware architecture with several processors and shared memory. However, it is a question whether the efficiency of such a modified algorithm is still good enough. Moreover, there is no doubt that the modified algorithm is often unnecessarily complex for the low-degree of parallelism. This led us to develop a new algorithm, more suitable for a limited number of processors, typically two or four. We discuss this topic in the next section.

3. Proposed parallel algorithms

3.1. Randomized incremental insertion

We have chosen a randomized incremental insertion as the base for our parallel algorithm. Let us remind the reader that the algorithm consists of three phases: the *location* where a triangle to be subdivided has to be quickly found, followed by the *subdivision* and by the *legalization* where the circum-circle criterion is applied if necessary edges of the tested triangles are flipped.

Although the algorithm has $O(N^2)$ complexity in the worst-case, better complexity $O(N\log(N))$ in the expected-case can be reached using the structures for the efficient location of the triangle that contains the point currently inserted. We use a Directed Acyclic Graph (DAG) data structure. The location of one point in this data structure is possible in $O(\log(N))$ expected time (which is also the optimal time) and in $O(N)$ worst time; worst time occurs when the DAG is "totally imbalanced", having the shape of a list—due to randomization, such a situation is highly improbable. The DAG structure stores the history of changes. Each inner node of the DAG stores one triangle that existed in some previous triangulation. The current triangulation is stored in the leaves of the structure. The DAG root describes an auxiliary triangle that is chosen in such a manner to include all points to be inserted. A triangle that contains some vertex of this big triangle has to be removed in the post-processing phase. When a triangle has to be subdivided (in the subdivision phase), new nodes are created and joined to the node that stores the subdivided triangle. In the legalization phase, two new nodes are created and are joined to both input nodes. This explains why the DAG structure is not a tree, although it resembles a tree. More details about this structure can be found in [4,32].

There are other possibilities for the quick location of triangles: the random walk techniques [26] and the use of

quadtrees or bucketing techniques. Random walk techniques are especially popular. They consume less memory, however, expected time $O(N^{1/4})$ is needed per one point. The various alternatives for locating triangles are compared in [39]. In an effort to reduce memory use, Devillers in [40] suggests a hierarchical structure similar to the DAG. This structure consists of several connected levels, where each level contains a random sample of the level below it. The lowest level contains the current triangulation. Thus, time $O(\log(N))$ for the location of triangles is ensured.

The advantages of the incremental insertion algorithm are its simplicity and robustness. In the case of an incorrect or inconsistent Delaunay criterion evaluation caused by numerical inaccuracy, a triangulation with two or more non-Delaunay triangles is obtained, but this is still a valid triangulation. Moreover, the algorithm can be simply modified to incorporate constraints given in the form of prescribed edges [41], to use non-Euclidian metrics [42,43] and, additionally, its E^2 version is very similar to its E^3 version [44,45]. All input points do not need to be available at the beginning of computation (although their range of coordinates is required) which can also be an advantage for some applications. If the algorithm uses a randomized order of insertion, it becomes almost insensitive to the type of point distributions.

3.2. Analysis of the sequential algorithm and its parallelization

Let us first analyze the execution of the sequential algorithm. All three phases of the algorithm need to access to the DAG structure, however, each one in a different way. In the location phase, the DAG is accessed read-only to find the triangle containing the point to be inserted. Then all corresponding triangles are subdivided and new nodes are added to the DAG structure. The DAG is also modified during the legalization phase. Typical runtimes needed for these phases are in Fig. 1. The majority of time is consumed by the location phase (about 60–70%). The phases where the structure is modified take up to 25%. The remaining time is used for extraction of the $DT(S)$ from the DAG structure and for the deallocation of this structure, i.e. for the sequential part of the algorithm.

Let us assume that we use an architecture with shared memory. Usually, one thread runs on one processing element (PE). The input set is subdivided by the master thread (main thread of the application) among the worker threads. They simultaneously insert points into the shared triangulation, i.e. they access the shared DAG structure. Algorithm 1 shows a brief outline of the computation of the Delaunay triangulation.

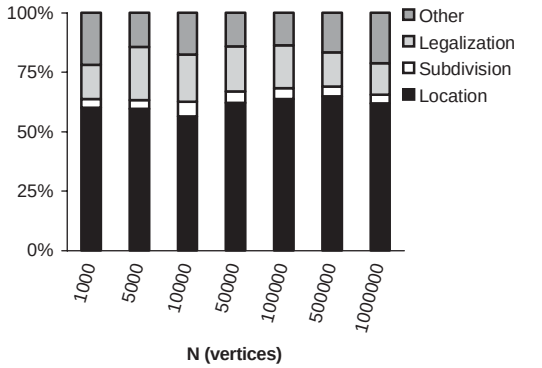

Fig. 1. Typical runtimes needed for the sequential algorithm with uniform data sets.

Algorithm 1. Parallel Delaunay triangulation—master thread.

Master thread:
Input: A set $S = p_0, p_1, \ldots, p_{n-1}$ of N points in E^2
Output: A Delaunay triangulation $DT(S)$

1. **begin**
2. Initialize the auxiliary big triangle;
3. Compute a random permutation of $p_0, p_1, \ldots, p_{n-1}$ of S;
4. Subdivide S into k subsets where k is the number of threads;
5. Start k 'worker' threads;
6. Wait inactively until they are finished;
7. *//Now, leaves of the DAG contain the //Delaunay triangulation of S and of //vertices of the auxiliary triangle*; remove all triangles containing vertices of the auxiliary triangle to get $DT(S)$;
8. **end**

While threads may run unsynchronized (if the leaves of the DAG are not considered) in the location phase, some synchronization in the subdivision and the legalization phases has to be implemented. We have identified three approaches of synchronization:

- The batch approach—several searching threads perform the location phase and only one specialized thread handles the subdivision and the legalization phase.
- The pessimistic approach—all threads perform the same work, however, the subdivision and the legalization phases can be done only in a critical section to ensure an exclusive access to the shared DAG structure.
- The optimistic approach—all threads perform simultaneously all parts of the algorithm, if they want to modify a triangle, they need to get an exclusive access to it.

We addressed the pessimistic approach to synchronization in [46]. Thanks to the use of the critical section in the algorithm, the achieved speed-up is significantly limited. Therefore, we designed several methods based on the optimistic approach. They achieved better and, moreover, scalable speed-up [32]. The stress was put on efficiency rather than on simplicity of these methods, thus they usually require little synchronization even in the location phase, otherwise, artifacts in resulting Delaunay triangulation could appear.

In this paper, we concentrate on two new methods (called the batch method and the circum-circle method) that are easier to implement, i.e. they do not need any synchronization in the location phase and their performance is comparable (or even better) than that of the methods presented in [32,5]. However, these methods are less scalable.

3.3. The batch method

This method is based on the batch approach introduced above. Not all threads perform the same work. When a searching thread (producer) finishes its unsynchronized location phase, it sends an index of the currently inserted point and the pointer of the last tested node to the specialized thread (consumer). The specialized thread completes the location of the node to be subdivided, subdivides it and proceeds with the legalization phase.

Communication between the searching thread(s) and the specialized thread is ensured by a queue in the shared memory. If this queue is empty, the specialized thread must wait. If it is full, the searching thread(s) must wait. The key issue here is how long the queue should be to prevent the waiting of the threads. A long queue implies a greater probability that the unsynchronized part of the location stops for many points in the same node and the specialized thread spends more time to complete the location phase. Even worse is to have a short queue because it would be full in a short time and the performance would decrease. The queue length is discussed in Section 4.2.

Let us note that the insertion of element into the queue needs to be done as an atomic operation, thus a short critical section is required. There is no such need for getting an element because we have only one specialized thread.

Algorithms for searching thread(s) and a specialized thread are described in Algorithm 2.

Algorithm 2. Parallel construction—the batch method.

The searching thread (a kind of worker thread):
Input: A set $S_k = p_0, p_1, \ldots, p_{m-1}$ of m points in $E^2, S_k \subset S$
Output: Modifies the shared queue

```
1.  begin
2.      for r:= 0 to m − 1 do
3.          begin
4.              Locate the triangle T0 containing pr on the level of the parents
                of the leaves;
5.              if the shared queue is full then
                wait;
6.              put T0, pr into the queue;
7.              end;
8.  end
```

The special thread (a kind of worker thread):
Input: Requests T_i, p_j in the shared queue
Output: Modifies the shared DAG structure, i.e. participates on the construction of the $DT(S)$

```
1.  begin
2.      while not all points inserted do
3.          begin
4.              if the shared queue is empty then
                wait;
5.              get T0, p0 from the queue;
6.              Locate the triangle T1 ∈ DT(S)
                containing p0;//start at T0
7.              Subdivide T1;//in the case where pr
                //lies on the shared edge, say between,
                //T1 and T2, subdivide also T2.
8.              Legalize all new triangles;
                //flipping
9.          end;
10. end
```

Let us suggest an interesting possibility[1] how to improve the performance of algorithm. If a searching thread cannot put new element into the queue because this queue is full, it takes an element from the queue, completes the location of the node to be subdivided and places the result back into the queue. This means that the searching thread takes over a part of job of the specialized thread. Unfortunately, a short critical section is required even for getting an element from the queue, thus an additional overhead is introduced. To counterbalance this overhead, the specialized thread has to take longer time to locate the node than to get an element (according to our experiments, we need at least 5 times larger time to get an element in this approach than in the original one). Another problem with this approach is that we have to somehow ensure that when the searching thread finishes the location of the node, it will place the

[1] The original idea was proposed by the unknown reviewer of this paper.

result into the queue immediately (i.e. there must be a free position in the queue). Therefore, this approach is suitable only for such data sets that the specialized thread requires checking of three or more nodes (on average) of the DAG structure to complete the location.

3.4. The circum-circle method

Preliminary version of this method appeared in [47]. The method is based on the optimistic approach, thus we need to get exclusive access to all nodes that are needed in the subdivision or the legalization phase of algorithm. This means that when a thread needs to operate with a node, it has to test whether the node is not already accessed by another thread and depending on the outcome it has to wait or it can proceed. Such a test can be done either by system resources, or using the geometric properties of the Delaunay triangulation. A "system" implementation of this test was used in [32]. The possibility of using geometry based test is described below.

It can be proved that the subdivision and the legalization influence only such triangles where the input point lies in their circum-spheres. Therefore, any thread that is currently inserting such a point that lies inside the circum-sphere of some triangle will access the node of this triangle during the insertion. It means that when a thread needs to access the node, it has to check whether no other currently inserted point lies inside the circum-sphere of the triangle of this node. If the result of this test is negative, the thread has to wait, otherwise it continues.

Now, we explain the synchronization in the circum-circle method on an example in detail. Let us assume that the thread T_0 wishes to flip the edge between the triangles T_1 and T_2 (see Fig. 2a) and that it has already got an exclusive access to these triangles (i.e. to the nodes storing these triangles). As the DAG structure also stores the neighborhood of the triangles, it is also necessary to modify the corresponding items in the nodes of the adjacent triangles. We have to prevent these nodes against being subdivided, and therefore, the thread has to get an exclusive access to them as well. Thus let us further assume that to complete this operation T_0 requires also an access to T_0. If no point currently inserted by other threads lies in the circum-circle C_0 of the triangle T_0, T_0 can operate with T_0 without any synchronization because it is clear that T_0 will not be reached by any other thread—see Fig. 2b.

Now, let us consider the opposite case, i.e. the point P_2 of the thread T_1 lies inside C_0. In such a case, the thread T_0 has to wait because the thread T_1 is going to work with T_0. The problem is that we know that the thread T_1 will reach the triangle T_0 in the near future but we are unable to determine exactly when. Thus sometimes a thread waits for a long time—see Fig. 2c.

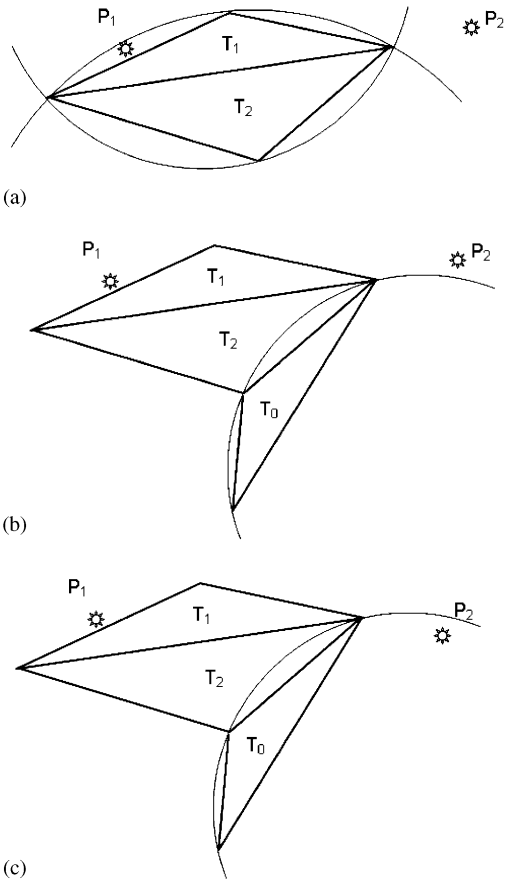

Fig. 2. Example of the legalization with the circum-circle method: (a) the edge between T_1 and T_2 has to be swapped, (b) P_2 lies outside the circum-circle C_0 of T_0—no synchronization needed and (c) P_2 lies inside the circum-circle C_0 of T_0—synchronization is necessary.

When the concurrent thread T_1 reaches the triangle T_0, it detects that the point of the thread T_0 lies in the circum-circle C_0. As mutual waiting of threads leads to a deadlock, each thread, before it starts waiting, has to test whether its waiting will be deadlock free. This detection requires a short critical section. In our example, T_0 is already waiting. When the thread T_1 detects a deadlock problem, the problem is handled as follows: the thread ignores the result of the circum-circle test, uses the triangle T_0 for its own purpose and sets a flag for the thread T_0 that informs the waiting thread (T_0) about an exceptional activity of another thread. When it finishes its activity, the thread T_0 is released. As T_1 could have subdivided T_0, T_1 or T_2, the thread T_0 has to check whether it can complete its operation.

If T_0 cannot continue, a non-zero probability of non-Delaunay triangles in the resulting triangulation arises. This is only caused by some unprocessed swap operation (i.e. the subdivision is always successfully completed);

the result of this is still a valid triangulation. Moreover, we know where the collision problem occurred and thus we can correct the results in an additional sequential phase. Our experiments show that the probability of incorrect triangulation is almost zero. When it happens, the number of wrong triangles is very low (no more than five in 2,000,000 triangles), therefore, we ceased to check the triangulation and correct it any way.

Now, we would like to discuss two problems whose occurrence affects negatively the performance of the algorithm and, moreover, the possibility of their occurrence increases with the growth of the number of threads.

When dealing with very narrow triangles in the $DT(S)$ (such triangles often occur at the boundaries), their circum-circles are quite large (in the worst case all points lie in such a circum-circle). A larger circum-circle implies a larger probability that all the points inserted simultaneously by concurrent threads lie inside its interior and that the thread will have to wait. However, such a large circum-circle can block a thread too quickly, i.e. a thread has to wait although a concurrent thread currently works with distant triangles. Moreover, when a narrow triangle goes through the significant part of the triangulation process, the probability that several threads will have to wait to access this triangle is quite high.

The second problem is related to the circum-circle test evaluation. This test has to be done for each accessed triangle. However, the complexity of this test depends linearly on the number of tested points, i.e. on the number of used threads.

Algorithm 3 shows an example of a worker thread.

Algorithm 3. Parallel construction—the circum-circle method (simplified version).

The worker thread:
Input: A set $S_k = p_0, p_1, \ldots, p_{m-1}$ of m points in $E^2, S_k \subset S$
Output: Modifies the shared DAG structure, i.e. it participates on the construction of $DT(S)$

```
1.  begin
2.    for r := 0 to m − 1 do
3.      begin
4.        Locate the triangle T0 ∈ DT(S) containing pr;
5.        Put pr into the pool of tested points;
6.        Use the geometric test for all points in the pool,
            T0 and all its neighbours; //in the
            //case that pr lies on the shared edge,
            //say, between T0 //and T1, include into
            //the test also //neighbours of T1;
7.        if the test fails then wait and then go to 4;
8.        Subdivide T0;// or also T1
9.        Legalize all new triangles;
            //flipping
10.     end;
11.   end
```

Procedure of the legalization:
Input: A triangle T_0 to be legalized
Output: Modifies the shared DAG structure, i.e. participates in the construction of the $DS(S)$

```
1.  begin
2.    if T0 is not a leaf in the DAG then exit;
3.    if edge between T0 and its 2nd neighbor
      T1 has to be swapped
4.      then begin
5.        Use the geometric test for all points
          in the pool,
            T0, T1 and all their neighbours;
6.        if the test fails then wait and then
          go to 2;
7.        Swap the edge between T0 and T1;
8.        Legalize all new triangles;// flipping
9.      end;
10. end
```

4. Experiments and results

4.1. The experiments

The parallel solution of the Delaunay triangulation was implemented in Microsoft Visual Studio.NET 7.0 C++ using serial incremental algorithm implemented in Delphi 6. The main tests were done on a Dell Precision 410 (2× Intel Pentium III 500 MHz, cache 512 KB, 1 GB RAM) with the Microsoft Windows XP Professional operating system and on a Dell Power Edge 6400 (4× Intel Pentium III Xeon 550 MHz, cache 1 MB, 4 GB RAM) with the Microsoft Windows XP Advanced Server. For additional tests, we used a Shalla (2× Intel Pentium Celeron 533 MHz, cache 128 KB, 512 MB RAM) with the Microsoft Windows 2000 Professional operating system.

We tested several different point distributions such as grid, uniform, gauss, cluster etc. The points were generated in a unit square. Examples of these distributions are shown in Fig. 3. As we found our methods behaved uniformly (or equally well) with all tested data types (as will be shown later), we choose the uniform data set being representative. Besides the artificial data

sets, we tested with real data sets from [48,49]. The tested number of input points, i.e. data size N, was between 1000 and 1,000,000. In our experiments, there were no observable performance differences between real and uniform data sets of comparable sizes.

For each data size we tested several different data sets with the same distribution except for the real data. The artificial data sets were generated and stored on the disk before the experiment. Experiments were repeated several times (at least five times) to increase reliability of the results. Let us note that the differences in time consumed by the different data sets with the same number of points did not exceed 10%. The resulting speed-up was calculated as the median of the total sequential time divided by the median of the total parallel time. Time for I/O operations (i.e. reading the point file into the memory and storing the resulting triangulation onto disk) is excluded. We prefer to use the median rather than the average because this way we eliminate singular cases. The difference between the results of both functions, however, is insignificant. The efficiency is computed as speed-up divided by the number of used processing elements. Since there is no architecture with three PEs (if the machine is fully equipped), we did not test this number of PEs in our experiments.

4.2. The results of the batch method

Fig. 4 shows the achieved speed-up for the batch method running on the Dell Precision 410 for both uniform and real data sets. We tested three different numbers of used searching threads, i.e. three different configurations. As shown in this figure, the behavior of the achieved speed-up is the same for all tested configurations. At first, the speed-up quickly increases with the growing size of the input data set, then for some sizes it is almost constant and when larger data sets are processed, it tends to slowly decrease.

Let us discuss this behavior. The algorithm has to process one subdivision and zero or more, say L, swaps after the location. The time needed for one subdivision, indeed, does not depend on the data size. Although in the worst case swaps go through the whole triangulation, usually swaps are done locally and thus L is limited to a small number. This means that the time needed for the legalization is also independent of the size of the data set. While expected complexity of both parts of the algorithm is $O(1)$, expected complexity of the location is $O(\log(N))$ and, therefore, the time needed to locate a triangle is significantly dependent on the data size. How does this fact influence the speed-up? When the data set is very small, the searching threads produce many points in a short time, the shared queue becomes full and the searching threads have to wait. The possibility of waiting drops quickly as the data size grows and, therefore, we can notice a fast increase in the speed-up. When the searching threads are unable to insert a point into the queue in time, the queue becomes empty and the specialized thread has to wait. As this situation occurs rarely and, moreover, in such a case only one thread has to wait the performance decreases very slowly. We tested with queue lengths of 16, 32, 64, 128, 256, 512, 1024, 2048 and 4096. Table 1 shows an

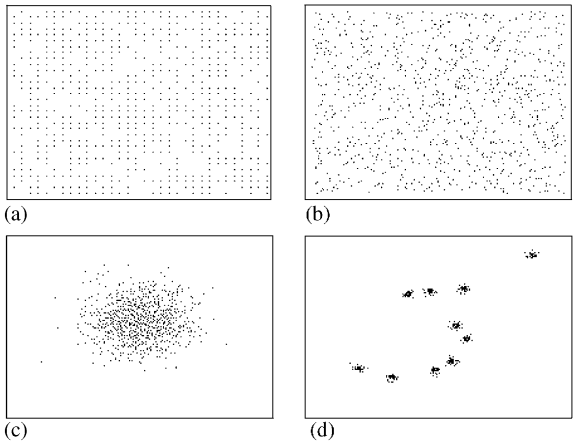

Fig. 3. Examples of tested distributions of the input points: (a) grid data, (b) uniform data, (c) gauss data and (d) cluster data.

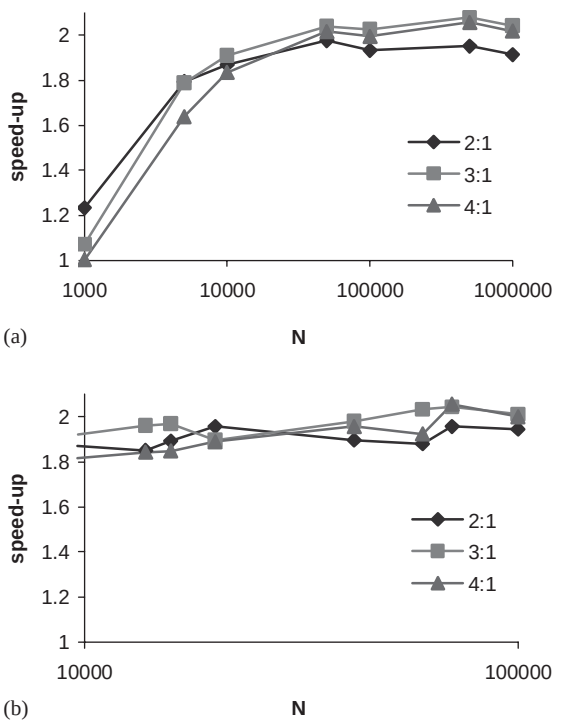

Fig. 4. The speed-up of the batch method running on a Dell P 410 (two PEs) when two, three or four searching threads were used: (a) uniform data and (b) real data.

Table 1
The ideal queue length

N	Threads	Length
1 000	2:1	64
5 000	3:1	256
10 000	2:1	512
50 000	3:1	512
100 000	3:1	1024
500 000	3:1	1024
1 000 000	3:1	1024

The queue lengths of 16, 32, 64, 128, 256, 512, 1024, 2048 and 4096 on a Dell P 410 (two PEs) were tested.

ideal length of the queue for different uniform data sizes. The queue length of 1024 seems to be optimal for larger data.

We recommend setting the number of the searching threads according to the size of the data sets to be processed. For typical data sets (of 100,000–1,000,000 points), the best configuration seems to be 3:1 (i.e. 3 searching threads to 1 specialized thread). For this configuration the efficiency is near one. Surprisingly, on a system with two processors, the use of five threads in total (i.e. 4:1 configuration) can lead to even better results. This will be discussed further together with the achieved super-linear speed-up.

Fig. 5 demonstrates the situation on an architecture with four processors. The behavior of the speed-up is similar to the one described above. Again, it seems that the best configuration is 3:1 for data sets up to 100,000 points. For larger data sets, the use of a 4:1 configuration becomes worth considering comes into consideration and even the use of five searching threads makes sense for data sets with more than approximately 1.5 millions of points. We also tested the configurations with more threads. They are probably useful only if the data sets with several million points are considered. However, computation of such data sets would require more memory than is available on 32-bits computers.

As there is no configuration ideal for all sizes, we recommend an improvement of the proposed batch that automatically estimates the number of threads that should be used for the location depending on the number of input points.

We have also tested other distributions of input points, however, as the results do not differ significantly from the results valid for the uniform and real data sets that we have just presented, we will omit the detailed presentation of these results. Fig. 6 presents a brief comparison. Fig. 6a shows the speed-up of the batch method with a configuration 3:1 using Dell P410 and Fig. 6b shows the speed-up of the improved batch method, i.e. using the configurations 3:1 for data sets up to 100,000 points and 4:1 for larger data sets, using a Dell PE 6400. The largest noted differences in speed-up

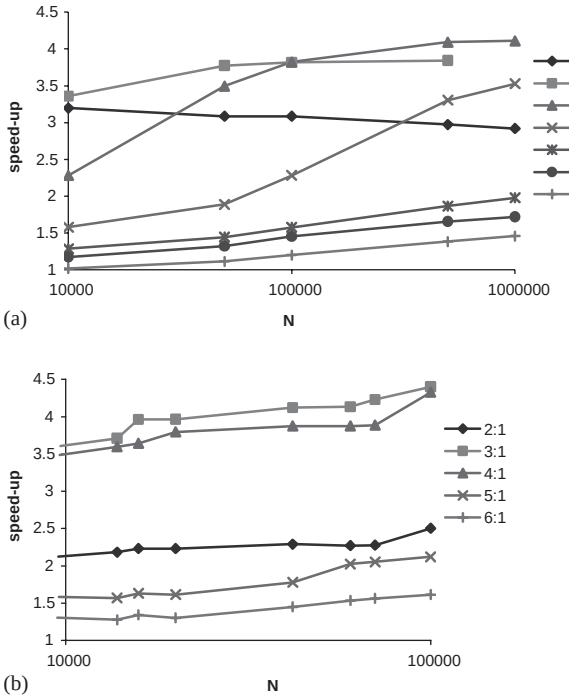

Fig. 5. The speed-u prate of the batch method using a Dell PE 6400 (4 PEs) when 2–6 searching threads were used: (a) uniform data and (b) real data.

are about 5% on the Dell P410 and 13% on the Dell PE 6400. On average the difference is lower than 7% for both multiprocessors.

Let us now discuss the possibility of super-linear speed-up apparent in almost all the graphs presented above. A super-linear speed-up occurs when the speed-up is larger than the number of processors. This situation generally occurs for two main reasons as explained in [50]. Typically it is caused by the more efficient use of the caches of the processors. There is no doubt that the code stored in the cache runs faster than the code stored in the RAM. Multiprocessors often have big caches able to hold large pieces of code or blocks of data. The code of the location phase is simple and short, thus it fits in the cache. With the sequential algorithm, location code remains in the cache for some time and then it is partially replaced by the code needed for the subdivision or legalization phase. That means that the RAM has to be accessed often. However, in the batch method our searching threads do only the location and, therefore, their code can stay in the caches longer and so the location takes a noticeably shorter time to be performed. Memory cache also influences the time needed for processing data with the algorithm. For example, if data already loaded into the cache for the thread T_0 is also needed for the threads T_1 and T_2, the data has not to be reloaded for T_1 and T_2. Such a case is

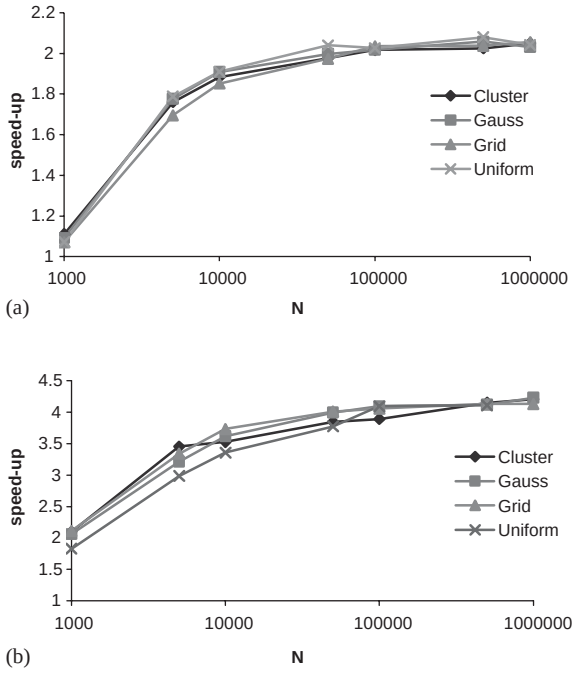

Fig. 6. The speed-up of the batch method for different distributions of input points: (a) Dell P 410 (two PEs), configuration 3:1 and (b) Dell PE 6400 (four PEs), configuration 3:1 up to 100 000, 4:1 for larger data sets.

rare for the sequential algorithm. The importance of cache effects grows with the size of the DAG structure, i.e. the number of points to be inserted.

Table 2 confirms the cache-effect. In this table, we compare the speed-up achieved using a Shalla (cache 128 KB), a Dell Precision 410 (cache 512 KB) and a Dell Power Edge 6400 (cache 1 MB) for the batch method. To ensure a comparable environment we limit the run of the algorithm on PE 6400 to two processors.

Caches are not the only reason for the noted super-linear behavior. Speed-up are also influenced by the internal parallelization of the kernel of the operating system. In further experiments we show that when a sequential algorithm runs only on the first processor as in our test, it takes about 4% longer than when it runs on any of other processors.

4.3. The results of the circum-circle method

Fig. 7 presents the results of tests using the Dell Power Edge 6400 for uniform and real data sets. While for two processors we can see super-linear speed-up, for more processors no unexpected speed-up occurs. Our experiments reveal that the efficiency of the algorithm decreases when the number of processors increases. When we analyzed the proposed method in the theoretical section, we had already explained the reason

Table 2
The influence of the cache-effect on the speed-up on three different computers for the batch method with the configuration 3:1 (run limited to two PEs)

N	Speed-up		
	Shalla	P 410	PE 6400
1 000	1.132	1.072	1.642
5 000	1.698	1.787	2.218
10 000	1.606	1.910	2.391
50 000	1.695	2.039	2.342
100 000	1.749	2.025	2.422
500 000	1.745	2.079	2.399

of such behavior. Let us recall that the complexity $O(k)$ of the 'geometric' test, where k is number of used threads and the possibility that a point inserted by a concurrent thread already lies inside the larger circum-circle (corresponding to narrow triangles) increases with the number of used threads.

When searching for the reasons for the super-linear speed-up, we have found that there is no significant partition of load via processors' caches as was the case with the batch method because all threads run the same program. However, the influence of the memory cache and the influence of the internal parallelization of the operating system are important. Table 3 shows the cache-effects for this method.

Fig. 8 compares the speed-up reached for different point distributions. Fig. 8a shows the speed-up using a Dell P410. The largest noted difference in speed-up is about 3.5%, on average the difference does not exceed 2.5%. A slightly different situation occurs using a Dell PE 6400—see Fig. 8b. Experiments with cluster data sets reached better speed-up than experiments with any other point distribution. If we ignore cluster data, then the largest difference in speed-up is about 9% and on average about 5%, otherwise the largest difference achieved 11% and on average about 8%. It seems that the circum-circle method is more efficient for cluster data sets. The reason for such behavior is directly related to the previously described problem of circum-circles of narrow triangles. When dealing with cluster data sets, the possibility that any point lies inside the circum-circle of the tested triangle grows slower comparing with other distributions. This is caused by the fact that insertion of a point into one cluster barely influences another cluster, thus if the thread T_0 works with cluster A and the thread T_1 works with cluster B then the probability that T_0 or T_1 will have to wait is almost zero. Therefore, we can see better speed-up for large cluster data sets.

4.4. The results of both methods for singular data sets

To reveal how the proposed methods perform with unusual data sets, we chose a data set with points lying

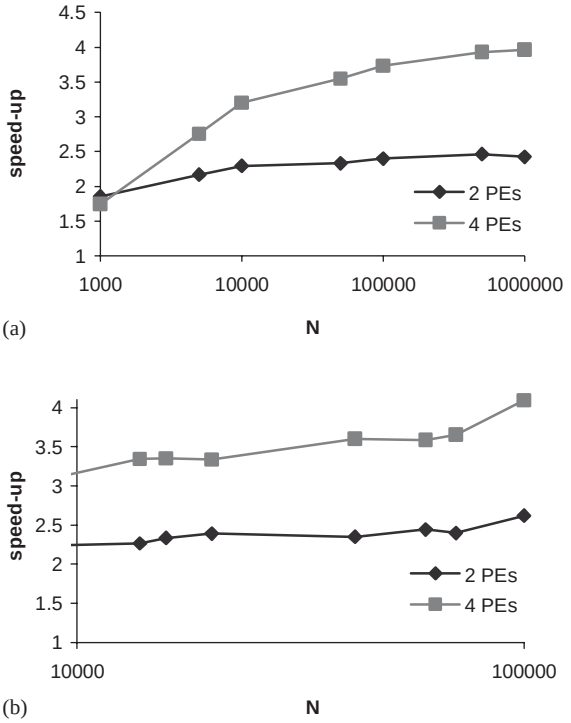

Fig. 7. The speed-up of the circum-circle method using a Dell PE 6400: (a) uniform data and (b) real data.

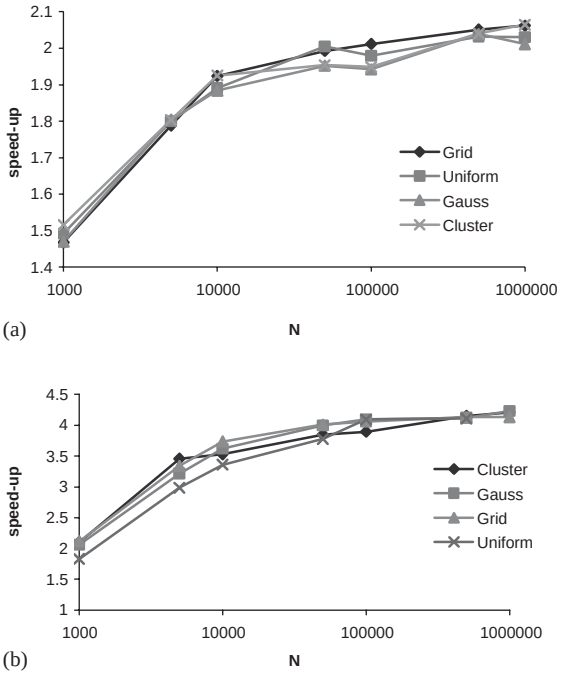

Fig. 8. The speed-up of the circum-circle method for different distributions of input points: (a) Dell P 410 (two PEs) and (b) Dell PE 6400 (four PEs).

Table 3
The influence of the cache-effect on the speed-up on three different computers (using two threads) for the circum-circle method

N	Speed-up		
	Shalla	P 410	PE 6400
1 000	1.297	1.491	1.857
5 000	1.516	1.800	2.169
10 000	1.531	1.891	2.297
50 000	1.637	2.004	2.333
100 000	1.610	1.979	2.399
500 000	1.717	2.032	2.463
1 000 000	N/A	2.030	2.424

on an arc. An example of the Delaunay triangulation of this arc data set is given in Fig. 9. It contains many narrow triangles—most of them are near the convex hull of the given points.

Fig. 10 presents the speed-up that the batch method reached for the "arc" data sets. Except for the lower (but still sufficiently good) efficiency of the algorithm, there is no difference in behavior of the speed-up. Our further experiments show that, the construction of the Delaunay triangulation of the arc data set requires a smaller number of processed swaps, i.e. it require a significantly shorter time for the legalization phase. Therefore, the

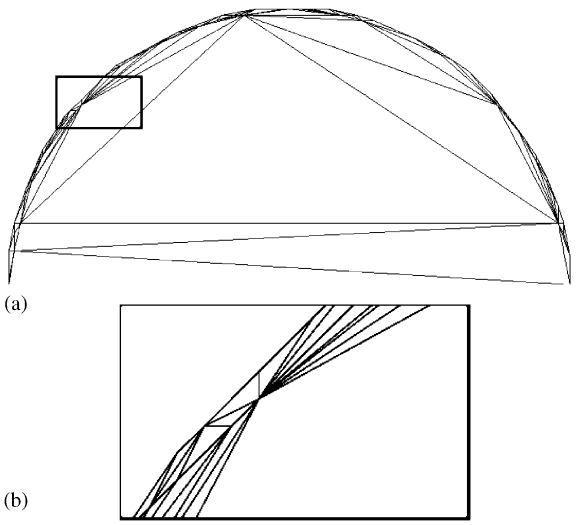

Fig. 9. An example of the Delaunay triangulation of 100 points lying on unit arc: (a) the entire Delaunay triangulation and (b) the detail of the selected area.

specialized thread has to wait quite often and as it waits pseudo-actively,[2] performance of the algorithm is

[2]The thread repeatedly yields its short time interval unless it cannot continue. Thus spared time can be used by another thread. If a thread waits only for a short time (as is usual in our

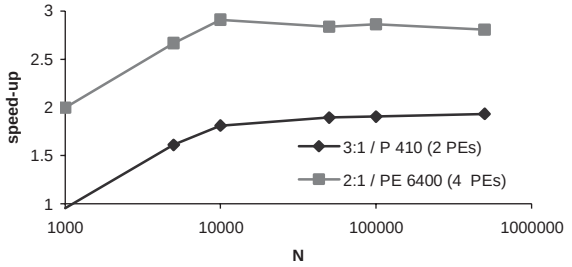

Fig. 10. The speed-up of the batch method for data sets with points lying on unit arc.

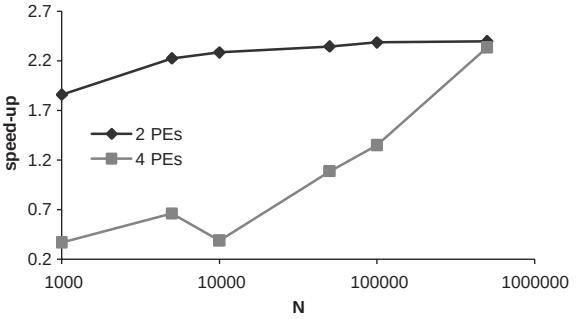

Fig. 11. A speed-up of the circum-circle method achieved when data sets with points lying on unit arc were tested on a Dell PE 6400 (with four PEs).

reduced (by about 8% on average in comparison with the uniform data).

Fig. 11 presents the achieved speed-up for the circum-circle method. There is no significant difference between the arc data sets and the uniform data sets when we consider two PEs only. Different results are obtained when we use four PEs—for smaller data sets the algorithm consumes even more time than the sequential one. The explanation for this is simple: as many triangles are narrow, many circum-circles are very large and thus the probability that a thread has to wait is quite high.

4.5. The subdivision of input points among threads

Let us now discuss the possibilities of subdivision of the input points among several threads and their influences on the performance of the proposed methods. In the previously described results, we assumed that points are subdivided randomly into k groups (where k is the number of used threads) in such a manner to ensure that there is equal number of points in each group. This static subdivision is sufficient, any load balancing is a kind of luxury because as the difference between two insertions are negligible, the threads finish their work almost at the same time. It does not seem necessary to use a different strategy with the batch method.

Whether the strategy of random subdivision is also ideal for the method of the circum-circle is an open question. It is quite clear that the convex hulls of the points processed by each thread are overlapping. It implies that there is a greater probability for a collision in the subdivision or the legalization. As we have shown, our algorithm seems to work best for cluster data sets. Therefore, a better strategy seems to be a subdivision of input points into k groups in such a manner to ensure that we have an equal number of points in each group and also that the convex hulls of these groups have a minimal intersection. This means that we artificially "convert" input points into clusters.

The simplest way is to subdivide the input points into k slabs in the x-coordinate direction. To do this, we use a modified algorithm for the median computation.[3] Another possibility is to apply a median subdivision on the distances of the input points from the origin, i.e. $x^2 + y^2$. Both possibilities, indeed, require additional processing time.

Fig. 12 compares the speed-up of all three types of subdivision for uniform data. As we can see, additional time is required for a more sophisticated subdivision scheme is not counterbalanced in the computation and, therefore, the use of such a scheme reduces the performance of the algorithm. We have noticed that this is also true for any other distribution of input points. We expect better behavior with six processors. However, as we have limited access to systems with a higher number of processors, we were not able to confirm this hypothesis.

4.6. Comparison with other methods

In this subsection, we compare the results of the presented methods with other existing methods. This is not an easy task as many existing algorithms work with distributed memory only or they are designed according to different principles for the construction of the $DT(S)$ or they were tested for tasks in E^3 only. Moreover, their authors often do not present the speed-up but raw execution times only. The best candidate for such comparison would be Chrisochoides and Sukup [34], however, authors do not present any results. Therefore, Table 4 compares the speed-up of the newly proposed methods, our previous optimistic method [32] and Hardwick–Bleloch algorithm from [28] only. The

(*footnote continued*)
case), this solution leads to better performance than the use of standard resources for synchronization supported by the operating systems, such as semaphores, etc.

[3] The algorithm for median computation was implemented by our students Mr. Kroc and Mr. Šimána.

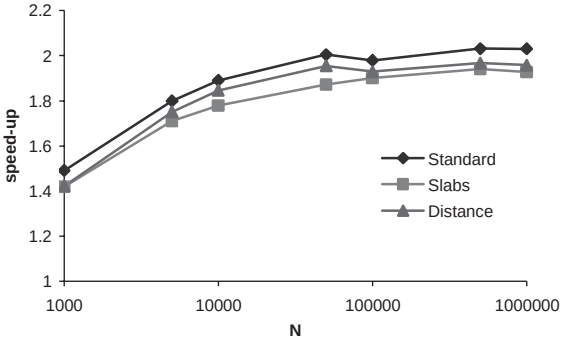

Fig. 12. The influence of subdivision of the input points among threads on the achieved speed-up for the circum-circle method. Uniform data sets were tested on a Dell P 410 (with two PEs).

Table 4
The comparison of the speed-up reached by different methods for a uniform data set with about 100,000 points

PEs	Hardwick [22]	Optimistic method [26]	Batch method 3:1	Circum-circle method
2	1.82	2.28	2.42	2.39
4	3.33	3.98	3.81	3.73

algorithm of Hardwick [28] is based on the Dwyer's sequential algorithm [25] and Hardwick uses the times of that sequential algorithm to compute his speed-up. All our methods are based on the incremental insertion algorithm and, therefore, the speed-up of our methods was related to that sequential algorithm. All our methods ran on a Dell Power Edge 6400, Hardwick's algorithm on a SGI Power Challenge. All methods worked with shared memory and processed a uniform data set with approximately 100,000 points.

It would not be fair to claim that our proposed methods are better than the algorithm of [28] because the experiment were performed on different architectures and the speed-up evaluation is also different. However, we can claim that our methods achieve at least the same speed-up and, moreover, these methods are easier to implement than Hardwick's algorithm. If we compare our new methods with the algorithm presented in [32], both the batch and the circum-circle methods achieve better speed-up for two PEs. A slightly worse speed-up was noticed with the circum-circle method using four PEs.

5. Conclusion

In this paper, we have described two new parallel methods for the construction of the Delaunay triangulation based on randomized incremental insertion. The methods are designed for multiprocessors with shared memory and limited number of processors, optimally two or four. Although the stress was originally put on the simplicity of the implementation, both methods also reached a significant speed-up compared with the sequential algorithm.

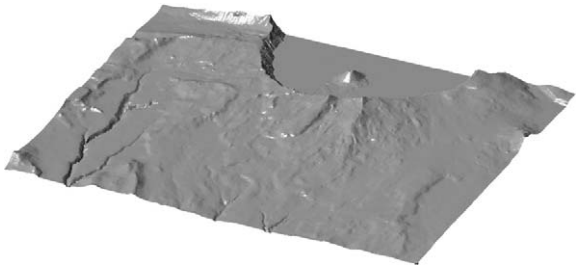

Fig. 13. Crater Lake, USA. An example of a tested real data set.

The batch method is not scalable and is intended to be used only with up to four processors. When two PEs are used, it shows a super-linear speed-up for data sizes larger than 10,000 points, i.e. a rate of 2.34–2.42 for uniform data sets and 2.38–2.65 for real data sets. A speed-up 3.36–3.84 for uniform data and 3.71–4.40 for real data was achieved when four processors were used.

The method of the circum-circle is easier to implement than our previously published methods (the batch method excluded) and it shows super-linear speed-up when two processors are used, i.e. a rate of 2.34–2.46 for uniform data and of 2.27–2.62 for real data. Also a good speed-up is achieved when four processors are used, i.e. rate of 3.21–3.93 for uniform data and of 3.34–4.09 for real data. This method can be used also in architectures with more processors, however, its efficiency decreases by increasing the number of PEs.

Fig. 13 shows an example of a real data set. This is a digital elevation model of Crater Lake, USA containing 100,001 points. The batch method (with a 3:1 configuration) reached speed-up 2.01 on a Dell Precision 410 with two PEs (i.e. a 100% efficiency gain) and of 4.40 on a Dell Power Edge 6400 with four PEs (i.e. an efficiency gain larger than 100%). The method of circum-circle achieved slightly worse speed-up. Here, the speed-up was 2.00 on a Dells P 410 (i.e. a 100% efficiency gain) and of 4.09 on a Dell PE 6400 (i.e. an efficiency gain larger than 100%). Let us note that the model was rendered using the MVE visualization package [51].

Acknowledgements

We would like to thank Dell Computer in the Czech Republic, especially Mr. Ponnert, for lending us a HotPlug SCSI disk for the Dell Power Edge 6400 which

allowed us to experiment on this computer. We would also like to thank Prof. V. Skala from the University of West Bohemia, Pilsen, Czech Republic for providing the conditions in which made this work possible, Mr. Hale, an American lector, for his help with English language and an unknown reviewer for his or her hints.

References

[1] Delaunay B. Sur la sphere vide. Izvestiya Akademii Nauk SSSR, Otdelenie Matematicheskii i Estestvennyka Nauk 1934;7:793–800.

[2] Делоне БН, Александров АД, ПадуровIм Н. Математические основI структурного анализа кристал лов, Москва, Матем. литература, 1934.

[3] Radke J, Flodmark A. The use of spatial decomposition for constructing street centerlines. Geographic Information Services 1999;5(1):15–23.

[4] de Berg M, van Kreveld M, Overmars M, Schwarzkopf O. Computational geometry. Algorithms and applications. Berlin Heidelberg: Springer; 1997. ISBN: 3-540-65620-0.

[5] Gonçalves G, Julien P, Riazanoff S, Cervelle B. Preserving cartographic quality in DTM interpolation from contour lines. ISPRS Journal of Photogrammetry and Remote Sensing 2002;56(3):210–20.

[6] Okusanya T, Peraire J. Parallel unstructured mesh generation, Presented at Fifth International Conference on Numerical Grid Generation in Computational Fluid Dynamics and Related Fields, Mississippi, 1996.

[7] Okusanya T, Peraire J. 3D parallel unstructured mesh generation, http://citeseer.nj.nec.com/article/okusanya97-parallel.html.

[8] Walkington NJ, Antaki JF, Blelloch GE, Ghattas O, Melcevic I, Miller GL. A parallel dynamic-mesh Lagrangian method for simulation of flows with dynamic interfaces. In: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (CDROM), November 2000, Dallas, TX, United States, 2000. p. 26-es.

[9] Attali D, Lachaud OJ. Delaunay conforming iso-surface, skeleton extraction and noise removal. Computational Geometry 2001;19(2–3):175–89.

[10] Park JH, Park HW. Fast view interpolation of stereo images using image gradient and disparity triangulation. Signal Processing: Image Communication 2003;18(5): 401–16.

[11] Prasad L, Rao L.R. A geometric transform for shape feature extraction. In: Proceedings of the 45th SPIE Annual Meeting, San Diego,CA, 2000.

[12] Xiao Y, Yan H. Text region extraction in a document image based on the Delaunay tessellation. The Pattern Recognition 2003;36(3):799–809.

[13] Chung SW, Kim SJ. A remeshing algorithm based on bubble packing method and its application to large deformation problems. Finite Elements in Analysis and Design 2003;39(4):301–24.

[14] Béchet E, Cuilliere JC, Trochu F. Generation of a finite element MESH from stereolithography (STL) files. Computer-Aided Design 2002;34(1):1–17.

[15] Nishioka T, Tokudome H, Kinoshita M. Dynamic fracture-path prediction in impact fracture phenomena using moving finite element method based on Delaunay automatic mesh generation. International Journal of Solids and Structures 2001;38(30–31):5273–301.

[16] Mulchrone KF. Application of Delaunay triangulation to the nearest neighbour method of strain analysis. Journal of Structural Geology 2003;25(5):689–702.

[17] Adamian L, Jackups R, Binkowski AT, Liang J. Higher-order interhelical spatial interactions in membrane proteins. Journal of Molecular Biology 2003;327(1):251–72.

[18] Ostromoukhov V, Hersch RD. Stochastic clustered-dot dithering. Color imaging: device-independent color, color hardcopy, and graphic arts IV, SPIE vol. 3648, 1999. p. 496–505.

[19] Tekalp AM, Ostermann J. Face and 2-D mesh animation in MPEG-4. Signal Processing: Image Communication 2000;15(4–5):387–421.

[20] Cignoni P, Montani C, Perego R, Scopigno R. Parallel 3D Delaunay triangulation. Computer Graphics Forum (Eurographics'93), vol. 12, no. 3, 1993. p. C129–C42.

[21] Guibas LJ, Knuth DE, Sharir M. Randomized incremental construction of Delaunay and Voronoi diagrams. Algorithmica 1992;7:381–413.

[22] Fortune S. A sweepline algorithm for Voronoi diagrams. Algorithmica 1987;2:153–74.

[23] Sue P. Efficient parallel algorithms for closest point problems. Hanover, NH: Dartmouth College; 1994.

[24] Brown KQ. Voronoi diagrams from convex hulls. Information Processing Letters 1979;9(5):223–8.

[25] Dwyer RA. A simple divide-and-conquer algorithm for constructing Delaunay triangulation in $O(n \log\log n)$ expected time. In: Proceedings of the Second Annual Symposium on Computational Geometry. Washington, DC: ACM, 1986. p. 276–84.

[26] Guibas LJ, Stolfi J. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. ACM Transactions on Graphics 1985;4(2): 75–123.

[27] Aggarwal A, Chazelle B, Guibas L, O'Dunlaig C, Yap C. Parallel computational geometry. Algorithmica 1988;3(3): 293–327.

[28] Hardwick JC. Implementation and evaluation of an efficient parallel Delaunay triangulation algorithm. In: Proceedings of Nineth Annual Symposium on Parallel Algorithm and Architectures, 1997. p. 22–5.

[29] Chen MB, Chuang TR, Wu JJ. Efficient parallel implementations of 2D Delaunay triangulation with High Performance Fortran. In: Proceedings of 10th SIAM Conference on Parallel Processing for Scientific Computing, Portsmouth, VA, USA, Philadelphia, PA: SIAM Press; March 2001. 11pp.

[30] Lee S, Park CI, Park CM. An improved parallel algorithm for Delaunay triangulation on distributed memory parallel computers. Parallel Processing Letters 2001;11(2–3): 341–52.

[31] Lee F. Constructing the constrained Delaunay triangulation on the Intel Paragon. In: Proceedings of the 13th Annual Symposium on Computational Geometry, Washington, DC: ACM, 1997. p. 464–7.

[32] Kolingerová I, Kohout J. Optimistic parallel Delaunay triangulation. The Visual Computer 2002;18(8):511–29.
[33] Chrisochoides N, Nave D. Simultaneous mesh generation and partitioning for Delaunay meshes. In: Proceedings of the Eighth International Meshing Roundtable, South Lake Tahoe, CA, USA, 1999. p. 55–66.
[34] Chrisochoides N, Sukup F. Task parallel implementation of the Bowyer–Watson algorithm. In: Proceedings of the Fifth International Conference on Numerical Grid Generation in Computational Fluid Dynamic and Related Fields, 1996.
[35] Watson DF. Computing the *n*-dimensional Delaunay tessellation with application to Voronoi polytopes. Computer Journal 1981;24(2):167–72.
[36] Chew L. Guaranteed-quality triangular meshes. Technical Report TR-89-983, Cornell University, Ithaca, 1989.
[37] Spielman DA, Shang HT, Alper U. Parallel Delaunay refinement: algorithms and analyses. In: Proceedings of 11th International Meshing Roundtable, Sandia National Laboratories, September 15–18, 2002. p. 205–18.
[38] Puppo E, Davis LS, DeMenthon D, Teng A. Parallel terrain triangulation. International Journal of Geographical Information Systems 1994;8(2):105–28.
[39] Žalik B, Kolingerová I. An incremental construction algorithm for Delaunay triangulation using the nearest-point paradigm. International Journal of Geographical Information Science 2003;17(2):119–38.
[40] Devillers O. Improved incremental randomized Delaunay triangulation. In: Proceedings of 14th Annual Symposium on Computational Geometry. Washington, DC: ACM, 1998. p. 106–15.
[41] Vigo M. An improved incremental algorithm for constructing restricted Delaunay triangulations. Computers & Graphics 1997;22:215–23.
[42] Okabe, Boots B, Sugihara K. Spatial tesselations: concepts and applications of Voronoi diagrams, New York: Wiley, 1992. ISBN 0 471 93430 5.
[43] Vigo M, Pla N. Computing directional constrained Delaunay triangulations. Computer & Graphics 2000;24:181–90.
[44] Joe B. Construction of three-dimensional Delaunay triangulations using local transformations. Computer Aided Geometric Design 1991;8:123–42.
[45] Kohout J, Kolingerová I. Parallel Delaunay triangulation in E^3: make it simple. The Visual Computer 2003;19 (7 & 8):532–48.
[46] Kolingerová I, Kohout J. Pessimistic threaded Delaunay triangulation by randomized incremental insertion. In: Proceedings of Graphicon 2000, Russia, Moscow, 2000. p. 76–83.
[47] Kohout J, Kolingerová I. Parallel Delaunay triangulation based on circum-circle criterion. In: Proceedings of SCCG 2003. Budmerice, Slovakia, April 24–26, 2003. p. 85–93.
[48] Garland M. Sample data for terrain simplification, http://graphics.cs.uiuc.edu/~garland/research/quadrics.html.
[49] Žalik B. Database of Terrain data, University of Maribor, 1999.
[50] Sienicki J, Agrawal P, Agrawal VD, Bushnell ML. Superlinear speed-up in multiprocessing environment. In: Proceedings of the First International Workshop on Parallel Processing, 1994. p. 261–65.
[51] Modular Visualization Environment. http://herakles.zcu.cz/research.php.