UNIVERSITY OF WEST BOHEMIA

Faculty of Applied Sciences

DOCTORAL THESIS

Plzeň, 2005 Josef Kohout

UNIVERSITY OF WEST BOHEMIA

Faculty of Applied Sciences

DOCTORAL THESIS

in partial fulfillment of the requirement for the degree of
Doctor of Philosophy
in specialization

Computer Science and Engineering

Josef Kohout

Delaunay triangulation in parallel and distributed environment

Supervisor: Doc Dr. Ing. Ivana Kolingerová
Date of State doctoral exam: 30.6.2004
Date of Thesis consignation: 3.5.2005

Plzeň, 2005

ZÁPADOČESKÁ
UNIVERZITA
V PLZNI

Fakulta Aplikovaných Věd

DISERTAČNÍ PRÁCE

k získání akademického titulu doktor
v oboru

Informatika a Výpočetní Technika

Josef Kohout

Delaunay triangulation in parallel and distributed environment

Školitel: Doc Dr. Ing. Ivana Kolingerová
Datum státní doktorské zkoušky: 30.6.2004
Datum odevzdání práce: 3.5.2005

V Plzni, 2005

Prohlášení

Předkládám tímto k posouzení a obhajobě disertační práci zpracovanou na závěr doktorského studia na Fakultě aplikovaných věd Západočeské univerzity v Plzni.

Prohlašuji, že tuto práci jsem vypracoval samostatně s použitím odborné literatury a dostupných pramenů uvedených v seznamu, jenž je součástí této práce.

V Plzni, 5.4.2005 Josef Kohout

Delaunay Triangulation in Parallel and Distributed Environment

Josef Kohout

Abstract

This thesis gives an overview of parallel techniques used in the construction of the Delaunay triangulation, one of the fundamentals of computational geometry. Several new parallel algorithms for the construction of Delaunay triangulation in E^2 and E^3 suitable for symmetric multiprocessors (i.e., architectures with several processors and the shared memory) and for clusters of workstations are proposed. These architectures belong to the common equipment of laboratories. The problems of balanced workloads, synchronization between processors and reduction of communication are discussed. The proposed solutions were tested and the results are summarized in this thesis. The comparison with other existing algorithms is also given in the thesis. In the final part, the surface reconstruction as an application of the proposed solution is described and possible extensions of the Delaunay triangulation and their impacts on the solution are investigated.

This work was supported by the Ministry of Education of the Czech Republic – project MSM 235200005, project AV2030801 and FRVŠ 1342/2004/G1.

University of West Bohemia
Department of Computer Science and Engineering
Univerzitni 8
306 14 Pilsen
Czech Republic

Delaunay Triangulation in Parallel and Distributed Environment

Josef Kohout

Abstrakt

V této práce je uveden přehled paralelních technik používaných pro konstrukci Delaunayovy triangulace, jednoho ze základních problémů výpočetní geometrie. Bylo navrženo několik nových paralelních algoritmů pro výpočet Delaunayovy triangulace v E^2 a E^3 vhodných pro symetrické multiprocesory (tj. architektury s několika procesory a sdílenou pamětí) a clustery počítačů. Tyto architektury patří v současnosti mezi běžné vybavení laboratoří. Problémy rovnoměrného přidělení práce, synchronizace mezi procesory a redukce množství nezbytné komunikace jsou rovněž prozkoumány. Navržené řešení bylo otestováno a výsledky experimentů jsou uvedeny v práci. Taktéž srovnání s ostatními existujícími algoritmy lze nalézt v textu. V poslední části práce je popsána rekonstrukce povrchů z množiny roztroušených bodů coby aplikace navrženého řešení a jsou prošetřena možná rozšíření konstrukce Delaunayovy triangulace a důsledky těchto rozšíření na navržené paralelní řešení.

Tato práce byla podporována MŠMT České republiky – projekt MSM 235200005, projekt AV2030801 and FRVŠ 1342/2004/G1.

Západočeská univerzita v Plzni
Katedra informatiky a výpočetní techniky
Univerzitní 8
306 14 Plzeň

Acknowledgments

At this place, I want to take the opportunity to thank all the people who have helped me to carry out this work, and whom it was a pleasure to work with, also to the ones not explicitly named here. The great thanks belong to my supervisor Dr. Ivana Kolingerová, from the University of West Bohemia, Pilsen, Czech Republic, for her guidance in this research thesis, and for many helpful discussions and her ideas that supported my work. Furthermore, I would like to express my gratitude to Prof. Václav Skala, from the same university, for providing conditions in which this work has been possible.

I would like to thank Dr. Jiří Žára, to my colleagues (especially, Michal Varnuška and Petr Vaněček) and anonymous reviewers for their valuable hints in my research. I am thankful to Prof. Borut Žalik for providing us with real data sets for our experiments. Thanks belong to Dell Computer, Czech Republic, especially to Mr. Ponnert, for allowing us to experiment on their computers and for lending a HotPlug SCSI disk into Dell Power Edge 6400 and in this way allowing us to continue with our experiments. UniSys, Czech Republic deserves also thanks for allowing us to experiment on their computer.

I would also like to thank our funding agencies, and mention that this work was supported by the Ministry of Education of The Czech Republic - project MSM 235 200 005, project AV2030801 and project FRVŠ 1342/2004/G1.

I will also take the chance to express my thanks to many friends with whom I spent much of my time and to thank to anybody else who has participated on this research for their help and support. Last but not the least, I want to express my thanks to my parents who generously supported me all years of my research, and helped me so to achieve my goals, especially to complete this work.

Basic terminology

In this subsection, we explain various basic terms that are used in the next text of this thesis. Advanced terms will be explained in the text at the place where they firstly appear.

Algorithm is a finite sequence of finite steps that are needed to achieve the given goal. **Asymptotic complexity of an algorithm** determines the speed of the growing of time elements consumed by an algorithm in the dependence on the number N of input elements, where N is larger than some constant N_0.

Let us assume that each access to the memory and each execution of a simple instruction (e.g., +,-,*,/,=,if,call) needs one time unit, then for all possible input data sets with some N we can find maximal, minimal and average number of time elements needed by an algorithm to process any data set with this N. The function $f_{max}(N)$, which assigns an appropriate maximal number of time elements to each number N (> 0) of input elements, is called the worst-case complexity of an algorithm. Similarly, the function $f_{max}(N)$, which assigns an appropriate minimal number of time elements to each number N, is called the best-case complexity of an algorithm and the function $f_{avg}(N)$, indeed, is called the average-case complexity. Usually, asymptotical versions of these functions, which are valid only for $N > N_0 > 0$, are used.

Let consider the constants c_1, c_2, $N_0 > 0$ and the asymptotical complexities $f_x(N)$ and $g_x(N)$, where x is either *max*, *min* or *avg*. If $c_1 \cdot g_x(N) \leq f_x(N) \leq c_2 \cdot g_x(N)$ for $N > N_0$, then the functions f_x and g_x describe the same complexity and we write: $f_x(N) = \Theta(g_x(N))$. For example, the complexity of Divide & Conquer (explained later in this subsection) algorithm for the multiplication of two NxN matrices is: $f_{max}(N) = 8 \cdot f_{max}(N/2) + 4 \cdot \Theta(1)$, i.e., $\Theta(N^3)$. It is often impossible to find a suitable function $g_x(N)$ with both constants such that they would limit the function $f_x(N)$ from both sides. Therefore, let us consider the constant c instead of the constants c_1 and c_2 then we write $f_x(N) = \Omega(g_x(N))$ if $f_x(N)$ is greater or equal to $c \cdot g_x(N)$ for $N > N_0$, and we write $f_x(N) = O(g_x(N))$ if $f_x(N)$ is less or equal to $c \cdot g_x(N)$ for $N > N_0$.

For the algorithm evaluation, the asymptotical complexity for the worst-case is usually used. However, rarely the data set is 'so bad' that the worst-case occurs and, therefore, often the complexity in the expected case is presented in addition to the classic worst-case complexity.

Program is an implementation of an algorithm, i.e., it is a static notation of computational procedure written in such a form of operations to be recognizable by a processor. As the difference between the algorithm and the program is not significant, we often will substitute the term program for the term algorithm in this thesis.

Processing element (PE) or **processor** is a general computational element characterized by the set of operations that is it able to execute. The operations usually load and/or store some data from the memory. The **control unit** determines which instruction should be executed.

Process is some activity derived from the program. Its state is determined by the position of the operation to be executed in the program and by immediate values of processing data. The state changes whenever the processor performs the operation to be executed. Usually one processor runs one process for the whole process' lifetime. In such a case, we will substitute the term processor for the term process and vice versa.

Thread does not differ too much from the process: to maintain the thread requires lower overheads than to maintain the process. Therefore, we will use the terms thread and process as the synonyms in this thesis.

Critical section denotes a piece of the code in a program that cannot be executed simultaneously by more threads. When a thread reaches a critical section, it checks whether

no thread operates in this critical section. If the outcome of this test is negative, the thread has to wait until the critical section is free, otherwise it continues. Critical section usually also denotes a synchronization object used to handle the entering (included the waiting) and the leaving a critical section (in the meaning of protected piece of the code).

***k*-Simplex**, with $k \leq d$, is the convex combination of k +1 affinely independent points in a point set S in E^d. These points are called vertices of the simplex. In E^1 it is a line segment, in E^2 a triangle and in E^3 a tetrahedron. In this thesis, we use the term simplex also as a synonym to the term node. When we, therefore, speak about a modification of a simplex or an access to a simplex, we mean, actually, the modification of the node that stores information about this simplex or the access to the node data structure.

Node denotes a data structure, usually containing some links to other nodes, which describes a simplex. In this thesis, we often supersede the term node for the term simplex.

Convex hull *CH*(*S*) of a set of points S is the smallest convex geometrical object (polygon in E^2 and polyhedron in E^3) such that any point from S lies inside the interior of $CH(S)$ or it is one of the vertices of $CH(S)$.

Divide & Conquer denotes a recursive strategy consisting of two stages. In the first one, the divide stage, the input data set is repeatedly subdivided as equally as possible into smaller subsets until each subset is small enough to be solved directly. Afterwards, the solution for each subset is found. In the second stage, the merge stage, solutions of subsets (i.e., subsolutions) are repeatedly merged until the solution for the whole input set is obtained.

Used Shortcuts

The shortcuts used in this thesis are summarized in the following table:

2D	E^2	two-dimensional case, i.e., planar case
3D	E^3	three-dimensional case
CH(S)		convex hull of *S*
D&C		Divide & Conquer
DAG		Directed Acyclic Graph
DT	DT(S)	Delaunay triangulation
MS		Microsoft
PDT		Parallel Delaunay Triangulation
PE		Processing Element – processor
PEs		number of processing elements

Table of Contents

1 Introduction

Delaunay triangulation is one of the fundamental topics in the computational geometry and it is used in many areas, such as terrain modeling (GIS), scientific data visualization and interpolation, robotics, pattern recognition, meshing for finite element methods (FEM), natural sciences, computer graphics and multimedia, etc.

Many sequential algorithms for the computation of the Delaunay triangulation exist. Modern computer architectures allow us to compute the Delaunay triangulation in E^2 or E^3 with thousands of points by any of these sequential algorithms in a reasonable time. However, current applications often need to work with data sets such that they cannot be computed in one piece because of the common memory size limitations or their processing consumes too much time. In such cases, a parallel algorithm is useful and welcome. We can identify two different parallelization purposes. The first one is to compute the Delaunay triangulation of a set of points in as short time as possible without considering memory limitations. A different problem is to compute the Delaunay triangulation of very large input data sets where the final time is not as important as the memory utilization of PEs. A good example of such an application is the surface reconstruction from scattered point data.

Quite a big set of parallel algorithms exists, however, most of these parallel algorithms were designed in times when parallel architectures, with hundreds of processors, dominated in the research area and thus they put stress usually on the scalability rather than on the robustness and simplicity. For most applications, a less scalable but easy to implement and stable algorithm is preferred to a more scalable but complex one. The current situation on the hardware market supports this tendency – in the last few years, multiprocessors with several processors and shared memory have come into the consideration due to their low prices; especially two-processor workstations are now widely spread. Clusters of workstations are also available; especially small clusters are very popular. Many existing algorithms can be used after some modifications for these hardware architectures. However, it is a question whether the efficiency of a modified parallel algorithm is still good enough. There is no doubt that the modified algorithm is often unnecessarily complicated.

At present, there is a lack of algorithms proper for the currently used architectures, which we have just described. Such algorithms should use simple means of parallelization to be implemented by a wide computer community, but they should be effective enough to be an attractive choice in competition with long-existing serial algorithms.

This thesis consists of two parts: theoretical and practical. The goal of the first part is to investigate the construction of the Delaunay triangulation in both E^2 and E^3, the possibilities of its parallelization and to give a brief survey of existing parallel algorithms for the construction of $DT(S)$. As the properties of a parallel algorithm are mainly influenced by the features of the parallel architecture for which it was designed, we give an overview of available parallel architectures in the Section 2. In that section, we also describe general principles for the parallel algorithm design. The problems of even work distribution, load balancing, communication, traffic reduction and merging of partial solutions are discussed. The section should provide a good basic background to understand the problem of parallelization. More details about the parallelization can be found in our work [Koh04a], which gives a detailed study of the parallel techniques used in the computer graphics (related to the parallel rendering).

In Section 3, the problem of the Delaunay triangulation is described in detail. We discuss the properties of the Delaunay triangulation and sequential principles for its construction. Pros

and cons of each principle are discussed. In general, a principle that produces better quality of triangulations consumes also more time or it is more difficult to implement. None is ideal for any purposes. A survey of existing parallel algorithms is presented in Section 4. Many of them are based on Divide & Conquer strategy, which is natural for the parallelization but not simple. Only several of these algorithms offer robustness to numerical errors.

The second part of this thesis is more practically oriented. The main goal of this second part is to design a robust and simple parallel algorithm suitable for architectures with several processors and the shared memory as well as for cluster architectures. Using the knowledge acquisitioned in the theoretical part, we have chosen the method of the incremental insertion with local transformation as a base for our parallel algorithms because this method is easy to implement and it is stable (it produces fine quality meshes, no matter which kind of point distribution is used). The method is described in detail in Section 5. As this method has rather a sequential character, no wonder that, as far as we know, there is no existing parallel algorithm based on it. A parallelization is, therefore, a challenge for us.

Section 6 discusses options of the parallelization of the chosen algorithm of incremental insertion with local transformations and possibilities of even workload. Several parallel algorithms for symmetric multiprocessors are proposed in Section 7 and a few parallel algorithms for clusters of workstations are proposed in Section 8. These algorithms were implemented and tested. The results of the performed experiments are given in Section 9.

In Section 10, the algorithm by Varnuška [Var05] for surface reconstruction from scattered point data based on the construction of the Delaunay triangulation is described and the parallelization of the whole algorithm is proposed exploiting a solution of virtual shared memory from Section 8. The results of experiments with the developed application are given.

Section 11 investigates possible extensions of the Delaunay triangulation, e.g., to incorporate constraints given in the form of prescribed faces into the triangulation, to use non-Euclidian metrics or weights of points during the computation, and their impacts on the parallelization. In the section, we propose also another parallel algorithm for the construction of the Delaunay triangulation for clusters of workstations that, unlike algorithms described in Section 8, does not need to communicate during the insertion of points.

Last section, Section 12, summarizes current results and concludes this thesis. In this section, the objectives of the future work are discussed.

2 Parallel and Distributed Computing

Sequential computers are based on the model presented by John von Neumann. The performance of this model is limited by the speed of information exchange between the memory and the processing unit and by the execution rate of the instructions. In modern sequential computers, the speed of information exchange is improved by using memory interleaving (i.e., simultaneous memory access by having several memory banks) and by using caches; the execution rate is improved by pipelining. Despite these improvements, in many areas of human activity, there is a necessity to work with such data sets that their processing by any sequential algorithm cannot be finished in a reasonable time at a single sequential computer or the processing even requires more resources than are available at this computer. For example, sometimes the task has to be finished in a real time or it needs more memory than is the addressable amount. Let us mention [Tam01] quantum chemistry, statistical mechanics, relativistic physics, astrophysics, computational fluid dynamics and turbulence, genetic engineering, cell modeling, medicine, modeling human organs, global weather and environmental modeling, speech processing, data mining, computer graphics and computational geometry as examples of human activities where such data sets are common. In all these cases, a 'parallel computer' and an appropriate parallel algorithm are welcome.

2.1 Parallel Models

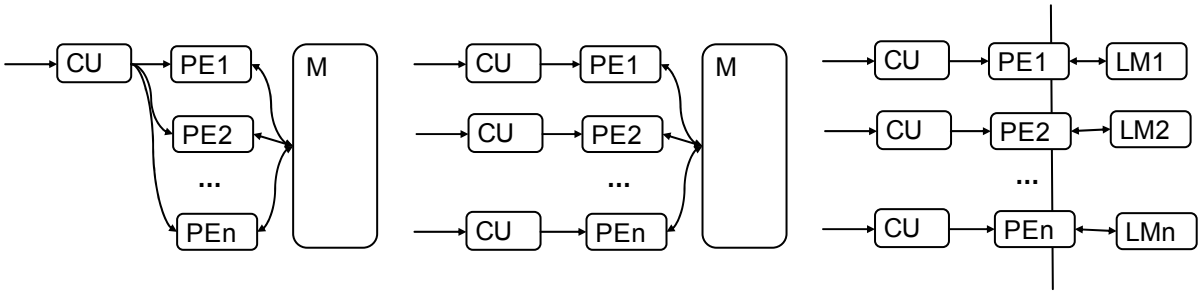
A parallel computer is a collection of processing elements (PE) that cooperatively solve the given task. While any sequential computer was based on the same sequential model, there is quite a big set of physical parallel models to be adopted in a parallel computer. The taxonomy of parallel architectures is ambiguous. Very popular is Flynn's taxonomy, which was firstly introduced by Flynn in 1966 and which classifies all architectures into four categories: SISD, MISD, SIMD, and MIMD depending on whether a single (S) or multiple streams (M) are used for instructions (I) and data (D). The problem with this classification is that it is nowadays too rough: SISD denotes sequential computers, MISD does not really exist (unless pipelining is considered as MISD configuration), SIMD denotes only a small group of architectures but MIMD contains tens of architectures.

Therefore, we classify parallel architectures into six practical models [Per99]: SIMD, parallel vector processor, symmetric multiprocessor, massive parallel processor, cluster (or network) of workstations and distributed shared memory. Let us note that the original Flynn's taxonomy classifies parallel vector processor as SIMD and all the remaining models as MIMD.

Single Instruction Multiple Data (SIMD) is a model that has only a single control unit, i.e., only one process runs (one copy of an algorithm is stored in the memory). The control unit dispatches the instruction to all processing elements and each PE executes the instruction with different data. Therefore, this model suits for such an algorithm where input data can be subdivided into several groups and processed simultaneously (e.g., operations with vectors or matrices). The structure of SIMD model is sketched in *Figure 2.1a*. Famous commercial parallel computers based on this model are: CM-2, Illiac IV, MP-1 and MP-2.

Parallel vector processor is an extension of SIMD. It contains a small number of powerful vector processing elements (they are based on SIMD) connected together and to the common shared memory (i.e., accessible to all processing elements) by a crossbar network switch. Famous parallel computers based on this model are Cray C-90, Cray T-90 and NEC SX-4. They are mainly used for the numerical computations. As this model is too specialized, the popularity of parallel vector processor has been going down in the recent years.

Symmetric multiprocessor contains a small group of common processing elements that are used in the sequential computers, i.e., it suits for any algorithm. Each processing element has an equal access to common shared memory and I/O devices via bus or crossbar switch. The structure of the model is sketched in *Figure 2.1b*. The efficiency of the system goes down with the increasing number of PEs because of the limited speed of data transfers. Therefore, the number of PEs is limited to a small number only. Great advantage of the symmetric multiprocessor model is, however, the low cost of parallel computers based on this model. No wonder that these parallel computers (especially computers with 2 PEs) became widely spread in the last years. From the set of commercial computers, let us name SGI Power Challenge, DEC Alpha server 8400, Dell Power Edge 7150, Dell Power Edge 8400, etc.



a) SIMD *b) symmetric multiprocessors* *c) massive parallel processor*

Figure 2.1: *Scheme of the model structures. Legend: PE - processing element, CU - control unit, M - memory, LM - local memory.*

Massive parallel processor contains a large group (often thousands) of common processing elements. Each processing element has an exclusive access to its local memory, all memory is distributed (i.e., there is no shared memory in the system). The processing elements are connected together by serial lines ensuring high communication bandwidth and low latency. *Figure 2.1c* brings the scheme of this model. There are various types of topology of the connection. Each topology is predetermined for some types of algorithms. A typical topology is a closed cube in E^n, briefly called n-Cube, where each element is directly connected to 2^{n-1} others. *Figure 2.2* shows an example of closed two-dimensional grid. Famous parallel computers based on this model are CM-5 (topology of a tree) and Intel Paragon (two-dimensional grid).

Cluster of workstations (or also computer network) is very similar to the massive parallel processor. It consists of a large group of sequential computers. As each processing unit is a complete computer (i.e., it has its own processing element, local memory, I/O devices and operating system), each computer can be different. These computers are connected together via some low-cost, however, in the comparison with the massive parallel processor, slow network (e.g., Ethernet, FDDI, Fiber-channel or ATM). Clusters are very popular. Namely, IBM SP2 is a successful commercial cluster.

Distributed Shared Memory (DSM) model belongs to a new generation of parallel computing. It combines advantages of the massive parallel processor and the symmetric multiprocessor. The model has distributed memory allowing higher number of PEs and, in addition, it creates an illusion of the shared memory, thus the implementation of algorithms is simplified. Moreover, the model is general enough. As far as we know, there is only one parallel computer based on this model – Cray T3D.

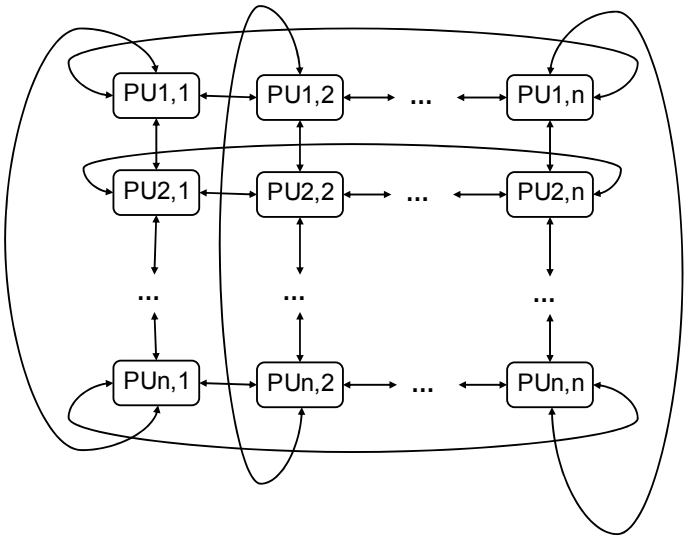


Figure 2.2: *An example of the topology of E^2 grid used in a massive parallel processor. Processing unit (PU) includes a processing element (PE) and its local memory (LM).*

Besides the practical models that we have just described, several theoretical models and parallel algorithms developed for them are published in the literature. Su [Su94] complains that many theoretical algorithms use complicated data structures or scheduling techniques to reduce the parallel "runtime" of basic algorithms, however, when they are implemented, they are so complicated and need so much data movement that achieve worse result than simpler solution. Despite it, the theoretical models can be used as a basis for designing parallel algorithms that are efficient in practice, for the explanation of results achieved by a parallel algorithm, or for the comparison of parallel algorithms. Therefore, let us introduce Parallel Random Access Machine (PRAM). This model assumes a machine with k (or even unlimited number of) processing elements and an unlimited random-access common shared memory. In a single machine cycle, each processing element can fetch a word, perform an operation and write the result back to memory. Different PRAM machines allow a different amount of concurrent memory access. EREW machines allow no concurrent access, CREW machines allow concurrent reads and CRCW machines allow concurrent writes as well.

Each parallel model described above offers different means and, therefore, requires using of a different programming technique in the design of a parallel algorithm. While an algorithm developed for the architectures with the distributed memory can use the message passing (i.e., sending and receiving messages) for the communication among the processes only, the architectures with the shared memory, including DSM systems, allow very simple and low-cost (except for the DSM model) inter-process communication through the shared memory. Therefore, we have to keep the pros and cons of all parallel models in mind when parallel algorithms are evaluated.

2.2 Parallel Programming

Although many parallel models exist, as it was described in the previous subsection, we can identify only a few parallel programming techniques used in parallel algorithms for the decomposition of computation [Jež97, Cro97]. These techniques include *Multiple Program Single Data* (*MPSD*) *parallelism*, *Single Program Multiple Data* (*SPMD*) *parallelism* and *Multiple Program Multiple Data* (*MPMD*) *parallelism*. Usually, a parallel algorithm uses only one of these programming techniques.

2.2.1 Multiple Program Single Data

MPSD is a functional parallelism. The computation is split into several distinct functions which can be applied in series to individual data items. Each function is exclusively assigned to its processing element and a data path is provided from one PE to another one. As the resulting scheme resembles a processor pipeline, the term pipeline is often used in the literature when MPSD parallelism is considered. This parallelism has two significant limitations. First, the overall speed of the pipeline is limited by its slowest stage, so functional units must be designed carefully to avoid bottlenecks. In an ideal case, all stages consume the same time. More importantly, the available speed-up is limited to the number of stages in the pipeline. The number of stages can be increased only as long as we are able to split all functions into smaller pieces that run in a similar time. If some function has to be performed atomically, it is useless to split other functions.

Let us assume a sequential algorithm that allows subdivision into k stages. A stage takes the time t_i. To process N data items, the algorithm consumes the total time t_{seq} where

$$t_{seq} = N \cdot \sum_{i=1}^{k} t_i \tag{2.1}$$

The parallel version of the algorithm then requires the total time t_{par} where

$$t_{par} = \sum_{i=1}^{k} t_i + (N-1) \cdot \max_{i=1}^{k} t_i \tag{2.2}$$

Speed-up s of this parallel algorithm can be calculated as t_{seq} / t_{par}. The efficiency of such an algorithm is given by the expression s / k.

2.2.2 Single Program Multiple Data

In SPMD parallelism, data parallelism, instead of performing a sequence of functions on a single data stream, the data set is split into multiple streams that are processed simultaneously by several processing elements executing the same program (i.e., the same functions).

There are two possibilities how to split the data set: static and dynamic load-balancing. In the first one, the data set is subdivided into streams at the beginning of the computation and this partition remains intact for the entire process. Fixed partition of the data set into streams of the same amount of input data elements introduces a possibility of imbalanced workloads of processors. This means that the difference of the time consumed by the PE that finished its work as the last one and the time consumed by the PE that finished as the first one is not insignificant. The load imbalance can be reduced by an adaptive partition where the amounts of elements in streams differ but the estimated time for their processing is the same. As the optimal load balancing strategy is *NP*-complete problem, some more or less efficient heuristics have to be used.

If an adaptive static load-balancing cannot be used or it does not brings a substantial improvement, dynamic load-balancing, which provides more flexibility in assigning workloads to processing elements, is used. There are two principal approaches: demand-driven and work stealing. In the demand-driven approach, the input data set is split into such very small streams that the number of these streams is much larger than the number of processing elements. The streams are assigned to processors one at a time. When a processor completes one task (i.e., the processing of a stream), it receives another task, and the process continues until all of tasks are complete.

Work stealing strategy starts with a fixed partition of the input data set. However, when a processor becomes idle, the remaining workloads of busy processors are split (i.e., their work is 'stolen') and reassigned to this idle processor.

In SPMD, the outputs of the processing of streams obtained by the processing elements are afterwards merged together to get the final result. As the merge stage very often cannot be parallelized and has to be performed in a sequential way, it limits the available speed-up of an algorithm. In many algorithms, it is also necessary to finish the execution of the operation j in the process A before the execution of the operation k in the process B can start. In such a case, B has to wait until A completes the operation. Therefore, the processes have to synchronize themselves, i.e., some inter-process communication is required. The inter-process communication and the waiting of processes limit the performance of a parallel algorithm.

Despite these troubles, the data parallel approach opens an opportunity to use more processing elements than in the functional parallelism and, therefore, to reach higher speed-up. Let us assume a sequential algorithm that consumes the total time t_{seq} to process N data items. If this algorithm can be fully parallelized and its parallel version running on k processors requires the total time t_{par}, then speed-up s of this parallel algorithm can be calculated as t_{seq} / t_{par} and its efficiency is given by the expression s / k.

2.2.3 Multiple Program Multiple Data

MPMD enriches SPMD parallelism. The input data set is split into multiple streams. The streams are assigned (statically or dynamically) to processors executing multiple programs. Let us note that parallel algorithms combining the functional and data parallelism can also be included into this category. For the MPMD approach, it is typical that the input data set contains data items of various data structures requiring various programs to be processed.

A very popular model, which exploits MPMD parallelism, involves two different programs, one program is executed only by one processor and the second program is executed by several processors. In the *client-server* strategy, several clients assign tasks to the server that has to perform a task and return results to the client. The opposite strategy is called the *farmer-worker*. There is one farmer, which drives the computation (it assigns tasks, collects sub-results and merges them into the final result), and several workers performing their tasks. Such a strategy is welcome for parallel architectures with the distributed memory.

As parallel algorithms with this parallelism are usually designed for the architecture with the distributed memory, they put the stress on the problems that cannot be processed on a sequential computer because of its technical limitations. Therefore, many MPMD algorithms do not compute speed-up and efficiency. A very important characteristic, however, is the scalability of a MPMD algorithm, which shows the capability of the algorithm to employ a large number of processors efficiently. Let us assume a parallel algorithm that consumes the total time t_{par1} to process N data items on k_1 processor. When $k_2 > k_1$ processors are used, the computation requires the total time t_{par2}. A value calculated as t_{par1} / t_{par2} evaluates the scalability of the given parallel algorithm.

2.3 Amdahl's Law

A sequential algorithm cannot usually be fully parallelized, i.e., some operations have to be performed in a sequential mode (e.g., write access to the shared memory, merge phase and I/O operations). Consider a program containing m operations, which take the same time T, to be executed on a parallel machine with k processors. If the fraction $1 - q$ of the operations has to operate in a sequential mode, and q can be executed in parallel, the speedup is limited to:

$$s = \frac{t_{seq}}{t_{par}} \leq \frac{m \cdot T}{\frac{q \cdot m \cdot T}{k} + (1-q) \cdot m \cdot T} = \frac{1}{\frac{q}{k} + (1-q)} \tag{2.3}$$

This equation is called Amdahl's law [Amd67]. Although it ignores many of the realities of computing (e.g., parallelization is free, communication is free, parallel operation is a multiple of sequential operation, without changing data structures or operational design), researches in the parallel processing community have been using Amdahl's Law to estimate the highest possible speed-ups of their parallel programs. For example, if a parallel algorithm contains 99% of operations that can be executed in parallel, then speed-up on a parallel architecture with unlimited number of processors will reach value 100. If we consider an architecture with 10 000 PEs, speed-up will be about 99.

Let us warn against a common error that some researches make when dealing with Amdahl's law. Assume a sequential program. First, parts that can be performed in parallel are found. The total time t_p needed by the program in these parts to process N data items is measured as well as the total consumed time t_{tot}. The parameter q is computed as t_p / t_{tot}. Let us assume the parallelizable parts contain m_p operations and the remaining parts contain m_s operations. However, when the considered sequential algorithm is parallelized, new operations have to be included into the code. Therefore, the resulting parallel algorithm has m_p+d_p and m_s+d_s operations. It may not be true that $(m_p+d_p) / m_p = (m_s+d_s) / m_s$. As $d_p << d_s$ in many parallel algorithms, the value of q calculated as t_p / t_{tot} is incorrect!

Even if the Amdahl's law is used correctly, there are some situations when a parallel program reaches higher speed-up than was the estimated one, or even it reaches super-linear speed-up, i.e., achieved speed-up is bigger than the number of used processing elements. Sun et al. [Sun95] discussed the reasons for such a behavior. First, the time needed to process a data set depends on the place where the required data set resides. If the data is stored in the cache, the shortest time is reached. On the other hand, if the data is stored in the external or remote memory (i.e., on the disk or in the memory of a remote computer), a large time is consumed. Sequential applications often need to work with such a data set that a part of this set has to be stored in slower local memory or in the, even slower, external or remote memory. Let us assume such a data set of N items that only $x < N$ items can be stored in the cache. While each of these items requires the time t_0 to be processed, an item from the local memory consumes the time $t_1 > t_0$. The total times t_{seq} and t_{par} required to process the entire data set on a parallel computer using one and two processors are in an ideal case:

$$\begin{aligned} t_{seq} &= x \cdot t_0 + (N-x) \cdot t_1 \\ t_{par} &= x \cdot t_0 + (\frac{N}{2} - x) \cdot t_1 \end{aligned} \tag{2.4}$$

Therefore, we achieve a super-linear speed-up s:

$$s = \frac{t_{seq}}{t_{par}} = \frac{N \cdot t_1 - x \cdot (t_1 - t_0)}{\frac{N}{2} \cdot t_1 - x \cdot (t_1 - t_0)} > 2 = \frac{N \cdot t_1 - x \cdot (t_1 - t_0)}{\frac{N}{2} \cdot t_1 - \frac{x}{2} \cdot (t_1 - t_0)} \tag{2.5}$$

There is another reason for the super-linear speed-up related to the caches in [Sun95, Sie94]. Parallel computers very often contain larger caches than their sequential counterparts and, therefore, cache hit ratio could be increased. More or less theoretical reason is that the parallelization of a sequential algorithm reduces overheads hidden in the original code.

A last but not the least reason is typical for algorithms that perform a task only if some condition is fulfilled. If we parallelize such an algorithm, we can avoid processing of less useful tasks than in the original serial algorithm. A good example is the backtracking [Rao92] where a node of the tree is expanded only in the case that we expect the solution somewhere in the sub-tree of this node and we expand a node with the highest probability of the success first. In the parallel backtracking, we skip the expansion of a sub-tree earlier and, therefore, the result is often achieved sooner.

3 Delaunay Triangulation

In this section, we describe the Delaunay triangulation and sequential methods for its construction. The Delaunay triangulation is a good representative of the problems of computational geometry that are often solved in a parallel or distributed environment. Parallelization of the Delaunay triangulation is discussed in the following sections.

Given a point set S in E^d (for our purpose let $d = \{2,3\}$) , the triangulation $T(S)$ of this set of points is a set of simplices such that:

- The point $p \in E^d$ is a vertex of a simplex from $T(S)$ if and only if p belongs to S; i.e., the vertices of the simplices are some points from the input set.
- The intersection of two simplices is either empty or it is a shared face, a shared edge, or a shared vertex.
- The set $T(S)$ is maximal: there is no simplex that can be added into $T(S)$ without violating previous rules; i.e., union of simplices and convex polyhedron formed by a convex hull $CH(S)$ is the same object.

Delaunay triangulation (shortly *DT*) was proposed by a Russian scientist Boris N. Delone [Del34a, Del34b]. However, as his original papers are not written in English and their translations are usually rather complex, we would recommend Radke's [Rad99] or de Berg's [Ber97] texts for details about Delaunay triangulation.

Delaunay triangulation $DT(S)$ of a set of points S in E^d is a triangulation such that the circum-sphere of any simplex does not contain any other point of S in its interior. In the next text, this criterion is also called the circum-sphere criterion (or circum-circle criterion in E^2).

There is also an alternative definition of the Delaunay triangulation: the *DT* is a dual of the Voronoi diagram $Vor(S)$, which is a set of points having the same distance from at least two points from S and, moreover, there is no other point from S with a smaller distance. The mathematical expression of the $Vor(S)$ can be written as:

$$Vor(S) = \{x \in E^d : \forall p_i \in S, \exists p_j \in S; i \neq j : |p_i x| = |p_j x| \wedge \neg \exists p_k \in S; k \neq i; k \neq j : |p_k x| \leq |p_i x|\}$$

Figure 3.1 shows the mutual relationship of the $Vor(S)$ and the $DT(S)$.

The basic properties of the $DT(S)$ are as follows [God97]:

- If no $d+2$ points lie on a common hyper-sphere and no $k+2$ points lie on a common subspace of the dimension k, where k is less than d, then the $DT(S)$ is unique. E.g., four points lying in the vertices of an empty square in E^2 have a common circle and two possible configurations of their triangulation.
- The Delaunay triangulation includes at most $O(N^{\lceil d/2 \rceil})$ simplices, where N is the number of points to be triangulated.
- It minimizes the maximum radius of the containment spheres of the simplices in the triangulation. The containment sphere of a simplex is the smallest possible sphere that encapsulates this simplex.
- The boundary of the $DT(S)$ is a convex hull of S.

- In E^2, it maximizes the minimal angle and, therefore, the Delaunay triangulation contains the most equiangular triangles of all triangulations (i.e., it limits the number of too narrow triangles that may cause problems in further processing).
- In the worst case, it can be computed in $O(N \cdot \log N + N^{\lfloor (d+1)/2 \rfloor})$, i.e., in $O(N \cdot \log N)$ for E^2 and in $O(N^2)$ for E^3. However, algorithms with $O(N)$ expected time also exist in both E^2 and E^3.

Due to these good properties, Delaunay triangulation is used in many areas, such as terrain modeling (GIS) [Gon02], scientific data visualization [Oku96, Oku97, Wal00, Att01] and interpolation [Par03], robotics, pattern recognition [Pra00, Xia02], meshing for finite element methods (FEM) [Chu03, Béc02, Nis01], natural sciences [Mul03, Ada03], computer graphics and multimedia [Ost99, Tek00], etc.

Many algorithms for construction of the Delaunay triangulation exist. Some of them exploit the duality and construct the Delaunay triangulation using the Voronoi diagram – see *Figure 3.1*. It is, however, more efficient to use some direct algorithm. We classify them into several categories: local improvement, incremental construction, incremental insertion, higher dimension embedding and divide & conquer.

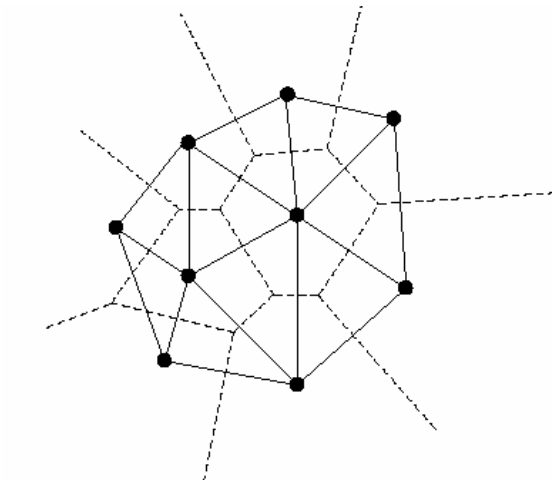
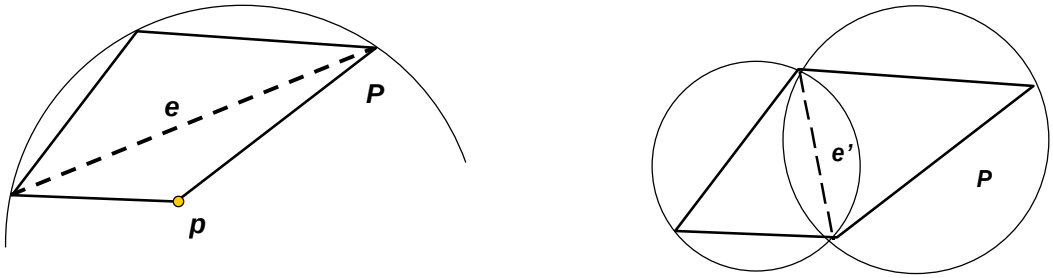


Figure 3.1: *The Delaunay triangulation (solid lines) and the Voronoi diagram (dashed lines) of the same set of points (big black dots)*

3.1 Local Improvement

Local improvement method is used mainly in E^2. First, a general triangulation $T(S)$ is created. In the second stage, this triangulation is successively converted into the *DT* by applying some local transformations. The Delaunay triangulation is such a triangulation where all edges are locally optimal. The edge e is locally optimal if and only if the polygon P formed by two triangles sharing this edge is not convex or the circum-circle of one of these two triangles does not contain the far point of the second triangle in its interior. If the edge e, i.e., the first diagonal of the convex polygon P, is not optimal, it is removed from the triangulation and the second diagonal e' is inserted into the triangulation. Then it is necessary to check all four edges of the polygon P on optimality. An example of the process is given also in *Figure 3.2*. Let us note that another strategy of local transformations exists: both possible triangulations of the polygon P, i.e., with the edge e and with the edge e', are investigated and the triangulation having the larger minimum angle of six angles in its triangles is picked as the proper configuration. This criterion is called max-min angle criterion.

The algorithms based on local improvement method are simple and robust: 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 it is still a valid triangulation. In practice, however, they are rarely used because it is not straightforward – the primary triangulation is required to be constructed, and because the convergence of the algorithm in E^3 is not ensured [God97]. Let us note that their complexity depends on the way how the primary triangulation is constructed, which is usually $O(N^2)$ in the worst case for E^2 and $O(N)$ in the expected case.



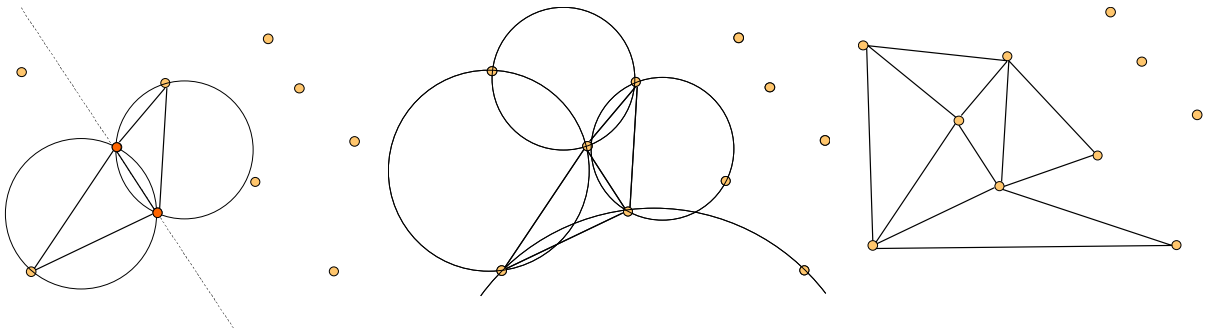
a) not optimal edge e – the circum-circle contains the far point p inside its interior

b) optimal edge e'

Figure 3.2: *Flipping the diagonals to convert a general triangulation into DT.*

3.2 Incremental Construction

The Delaunay triangulation is constructed by successively building simplices whose circum-spheres contain no points in S. Starting with a simplex as the primary triangulation, this triangulation is expanded by adding a proper point until the input set is not empty. Already constructed simplices are never reversed. Let us describe the general approach of the incremental construction [Cig93, Su94]. For easier understanding, we limit our explanation to the problem in E^2. First, a point from S is picked, the point nearest to the starting one is found and the first edge is created. Then for each outer side of an edge on the boundary of the current Delaunay triangulation (the first edge has two sides), the algorithm finds a point lying in the outer half-space such that the circum-circle of a triangle formed by the tested edge and this point has the smallest radius. If the point is successfully found, a new triangle is built. *Figure 3.3* shows several steps of the construction.



a) initial two triangles

b) construction of next triangles

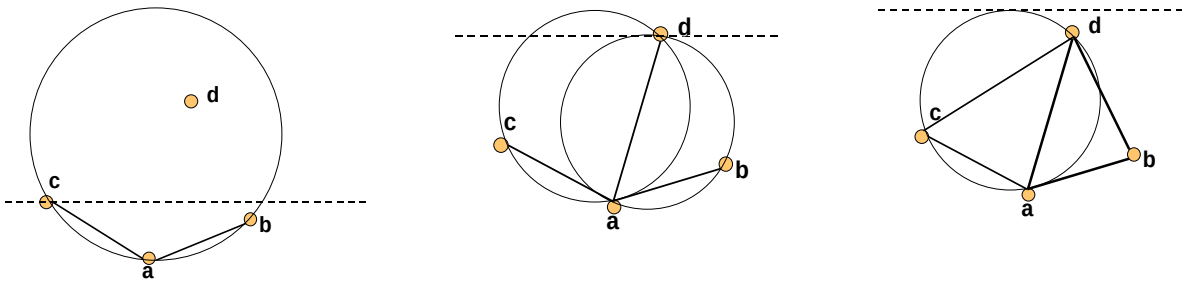
c) partial triangulation

Figure 3.3: *The incremental construction in E^2.*

This general approach is simple enough but if the algorithm does not incorporate some data structures to speed-up the location of the points, it has low efficiency: the worst-case complexity in E^3 is $O(N^3)$.

Another approach of the incremental construction is based on the sweeping paradigm. The sweeping algorithm by Fortune [For87] for the problem in E^2 is well-known. Although his algorithm seems to be quite complex, the worst-case complexity is $O(N \cdot \log N)$. The extensions for E^3 also exist but it is not used in practice because of its complexity.

Let us explain the main idea of the original Fortune's algorithm. More details can be found in [For87, Su94]. First, the points are sorted according to their y-coordinates. A moving up horizontal sweeping line separates the plane into two parts: in the lower half-space there is a current Delaunay triangulation, in the upper one are the points to be processed. The line stops in either a point or the top of a circum-circle of a potential triangle that was generated during the process. If the line does not meet any point before the top of the circum-circle is reached, the triangle with this circum-circle is added into the triangulation. *Figure 3.4* explains the generation of potential triangles. Whenever the line reaches a new point, a potential edge between this point and some previously visited point is added into a special list called frontier. If another edge sharing a point with the currently inserted edge is already in the frontier, the circum-circle of the triangle formed by these two edges is constructed – see *Figure 3.4a*. When the point d is reached it invalidates the circum-circle of the triangle a, b, c and generates a new potential edge a, d and two circum-circles – see *Figure 3.4b*. Then the sweeping line reaches the top of the circum-circle of the potential triangle a, b, d and, therefore, this triangle is added into the Delaunay triangulation. Finally, it reaches the second circum-circle and the triangle a, c, d is constructed as it is shown in *Figure 3.4c*.



a) generation of a circum-circle of the potential triangle a,b,c

b) the previous circum-circle is invalidated and new circum-circles are generated

c) partial triangulation

Figure 3.4: *The sweeping in E^2 by Fortune.*

3.3 Incremental Insertion

Starting with an auxiliary simplex that contains all points in its interior (or with the convex hull of all the points divided into simplices), these algorithms insert the points in S one at a time. In each step, a simplex containing the point to be inserted inside in its interior has to be found and this simplex and appropriate simplices in the neighborhood are modified in such a manner to incorporate current point and to ensure that the resulting triangulation is the *DT*.

As long as we do not consider time requirements, the order of the insertion is not important. The points do not need to be known in advance (although their range of coordinates is

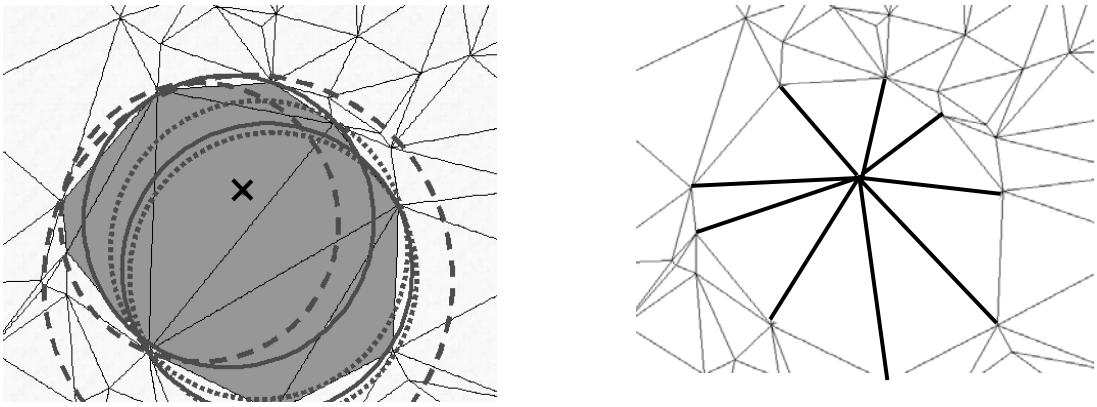
needed). If the algorithm uses a randomized order of insertion, it becomes almost insensitive to the type of points distributions.

The location of simplex per one point is possible in $O(\log N)$ expected time (and it is also the optimal time) and $O(N)$ worst time, if some data structure speeding-up the location, e.g., Directed Acyclic Graph (DAG) [Ber97], is used. The worst case happens 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 simplex that existed in some previous triangulation, the current triangulation is stored in the leaves. The DAG root describes an auxiliary simplex. Further information about this structure will be given later.

There are other possibilities for quick location: random walk techniques [Gui85], a use of quadtrees or bucketing techniques. The random walk techniques are especially popular. They consume less memory, however, expected time $O(N^{1/3})$ is needed per one location. The possibilities for location are compared in [Žal03]. In the effort to reduce memory use, Devillers in [Dev98] suggests a hierarchical structure similar to the DAG. It consists of several connected levels; each level contains a random sample of the level below. The lowest level contains the current triangulation. Time $O(\log N)$ for location is ensured.

After the location, there are two different methods. In the first one, the simplex containing the point to be inserted is subdivided and then the circum-sphere criterion is tested recursively on all simplices adjacent to the new ones and if necessary, their edges (faces) are flipped as in the local improvement approach [Ber97, Gui92]. This method was chosen as the basis for our parallel solution and, therefore, it will be described in more detail in the next section.

Another approach was presented by [Wat81] and it is known under the name Bowyer-Watson. The original algorithm needs $O(N^{(2d-1)/d})$ time in the worst-case. It works as follows: all simplices, which contain the point to be inserted in their circum-spheres, are removed from the triangulation and a convex cavity formed by the removed simplices is retriangulated using the currently inserted point. The retriangulation involves two steps. First, all vertices of the cavity are connected with the point to be inserted. Then, if it is necessary, the currently constructed edges (or faces) are swapped similarly to local improvement. *Figure 3.5* shows an example of the insertion. In this example, no local improvement technique is required, after the retriangulation we have already the Delaunay triangulation.



a) triangles forming the cavity *b) retriangulation of the cavity*

Figure 3.5: *Insertion of a point (marked by the cross) inside the DT by the Bowyer-Watson approach.*

In its simplest form, the Bowyer-Watson algorithm is not robust against floating-point roundoff error, which may cause appearance of overlapping simplices – see *Figure* 3.6. It is also more difficult for the implementation than the approach with successive performing of local transformations.

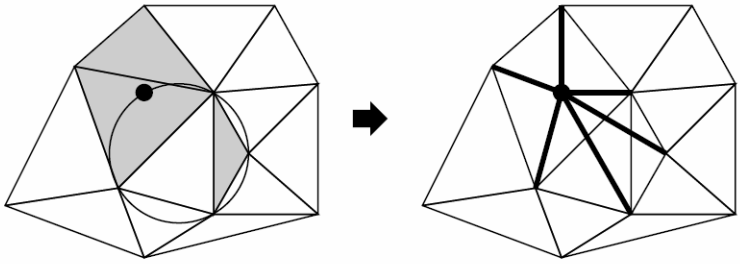


Figure 3.6: *Incorrect detection of triangles to be removed caused by floating-point roundoff error and the result of cavity retriangulation of such a set of triangles [Sch99].*

3.4 Higher Dimensional Embedding

These algorithms transform the points in E^d into the E^{d+1} space and then compute the convex hull of the transformed points. It was proven that the projection of the convex hull into E^d gives the Delaunay triangulation [God97]. In practice, these algorithms are mainly used only for E^2 [e.g., Bro79]. In such a case, a lifting projection onto the surface of a paraboloid is used. *Figure 3.7* illustrates this approach. Let us note that the complexity of the construction of the Delaunay triangulation in E^d is given by the complexity of construction $CH(S)$ in E^{d+1}, which is $O(N^{\lfloor (d+1)/2 \rfloor + 1})$ using gift-wrapping method.

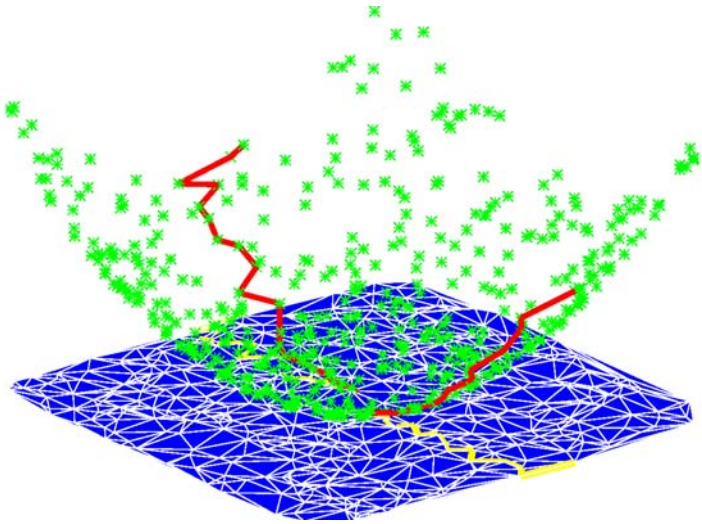
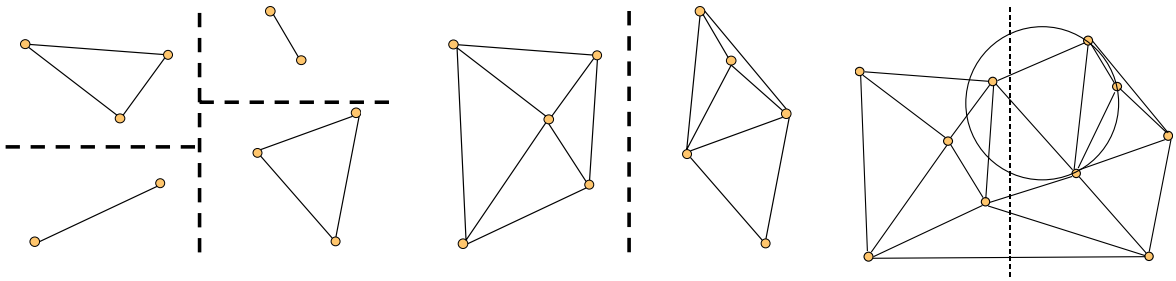


Figure 3.7 (see also Color Plates): *The projected points and the corresponding Delaunay triangulation [Har97]. Only a tiny part of the 3D convex hull is shown (black bold line segments).*

3.5 Divide & Conquer (D&C)

The main idea of these algorithms [e.g., Dwy86, Cig93, Gui85] is to recursively divide the input set of points until only a few points is in one group so that they can be easily triangulated in $O(1)$ time. Then the local triangulations are recursively merged together in

order to get the final Delaunay triangulation. *Figure 3.8* shows several steps of the D&C algorithm. The merge phase is quite complicated because it involves not only building of the faces among simplices from both triangulations but also corrections of existing simplices to satisfy Delaunay criterion. In the worst case, these corrections spread over the whole triangulation. Although the D&C algorithms are not simple to implement, they are in E^2 optimal for the worst-case. Let us note that the recursion usually stops when the size of the point set matches some given threshold and the local triangulation is then constructed by an algorithm belonging to any of the previous category (often the incremental construction).



a) local triangulations | *b) the first merging, simple* | *c) the second merging, local improvement necessary*

Figure 3.8: *Several steps of the construction of the DT in E^2 by D&C.*

4 Parallel Delaunay Triangulation

In this section, we give a survey of existing parallel algorithms for the construction of the Delaunay triangulation. Although the sequential construction of the Delaunay triangulation seems to be suitable for any application, it is not true in practice. The applications demanding real-time processing of some relatively small number of points exist as well as applications that need to process data sets with millions of points in reasonable time. The second kind of applications introduces the problem that to process their typical data sets, the applications need more physical memory than is available on a single computer. Good example of such an application is the surface reconstruction of the David's statue [Mich].

4.1 Incremental Construction

As the nature of the incremental construction allows a relatively easy parallelization, many parallel algorithms are based on this approach. In this section, we describe several well-known algorithms. The D&C algorithms exploiting the incremental construction are described in an independent section.

4.1.1 InCode

Cignoni et al. proposed a parallel algorithm called InCode [Cig93]. The algorithm subdivides the E^3 space (but can be easily modified for E^2) into k cubical areas. Each area as well as the whole set of the input points are assigned to one processor. The processor constructs simplices that have at least one vertex in its area, thus the simplices at the area's boundaries are created by more processors. To get the final triangulation, redundant simplices have to be removed in the post-processing sequential phase. This, indeed, affects the efficiency of the algorithm. *Figure 4.1* shows an example of the triangulation.

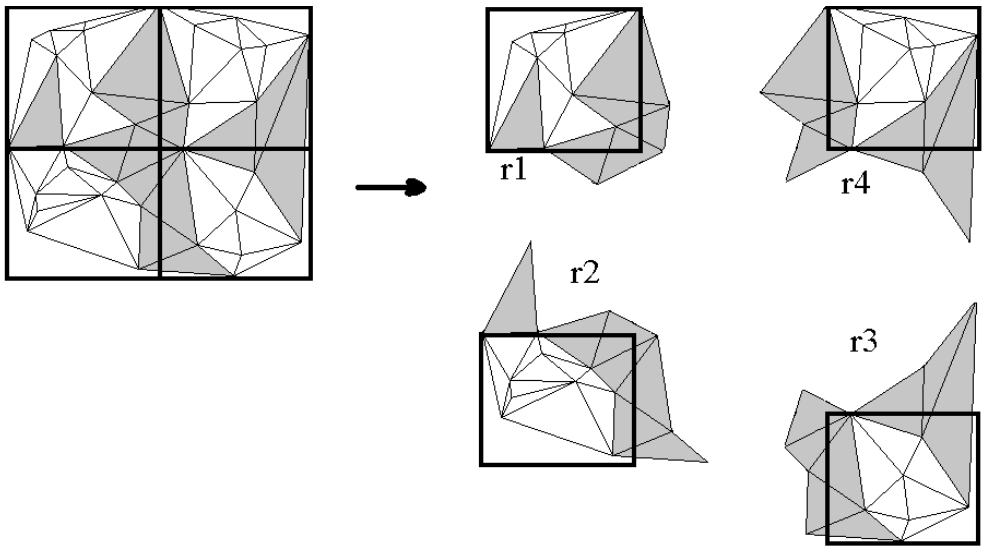


Figure 4.1: *The Delaunay triangulation in E^2 of using 4 PEs. The gray triangles are constructed redundantly by more processors.*

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 at nCUBE 2 system model 6410. The uniform data sets with 20 000 points were tested. Hardwick [Har97], however, notes that InCode is about 10 times slower for non-uniform data sets.

A similar algorithm was proposed by Teng et al. [Ten93]. The difference is that the algorithm avoids the redundancy by some synchronization of the processors during the computation. For the problem of the $DT(S)$ in E^3, the authors present, e.g., the speed-up 3.43 for 128 PEs and 6.08 for 256 PEs on CM-5 parallel architecture for uniform data set with 16 000 points where the speed-up is calculated according to the time spent by their algorithm using 32 PEs.

4.1.2 Lee et al.

Another interesting approach appears in Lee [Lee97]. The algorithm is useful for massive parallelization. Each processor has a set P of several points to be processed (an ideal loading is one point per PE) and the whole set S of input points for tests. For each point, it looks up the point nearest to the currently processed one and constructs the edge between them. The first PE afterwards collects all computed edges, removes redundant edges and distributes the computed edges among processors. They find the two nearest points for each received edge (one point in the left half-plane, one in the right half-plane) and construct two triangles. This second stage is repeated until no new elements are created. As Lee used Intel Paragon equipped by fast message routing chips for his experiments, we can expect that overhead for communication is significantly reduced. Unfortunately, the author does not present any results of his experiments.

4.2 Incremental Insertion

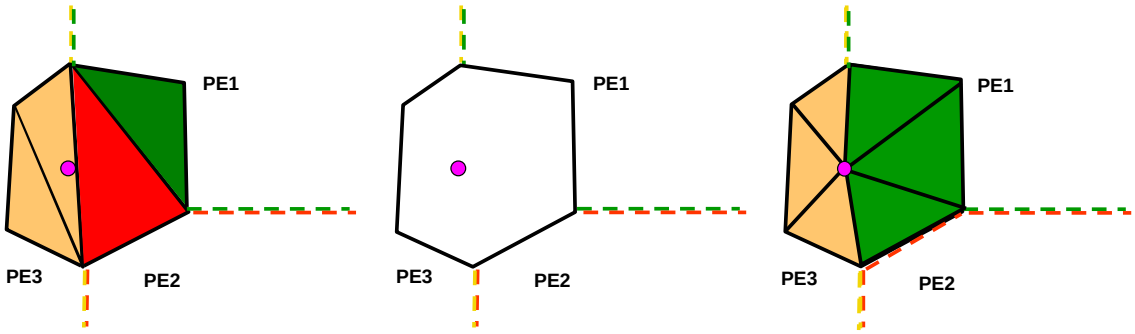
The nature of the incremental insertion is rather sequential and, therefore, sequential algorithms belonging to this category are rarely parallelized. As far as we know, there is just one purely parallel algorithm and several algorithms exploiting the principle of incremental insertion for their refinement purpose. These refinement algorithms start with already existing Delaunay triangulation and they try to find new artificial points that should be inserted in order to improve the quality of the triangulation (e.g., to improve the shape of simplices). The candidates for such points are usually the centers of circum-spheres of the simplices, the algorithm has to decide whether to use such a point or not. One very popular sequential solution was proposed by Chew [Che89].

4.2.1 Chrisochoides et al.

Chrisochoides et al. [Chr99, Chr96] parallelizes the Bowyer-Watson's algorithm. Let us remind that this algorithm is based on incremental insertion with cavity retriangulation. The parallel algorithm by Chrisochoides et al. starts by a sequential construction of a coarse triangulation of a subset of points by a sequential algorithm. The created simplices are partitioned into k continuous regions and distributed over k processors. The boundaries among regions are formed by some faces of the simplices and they may change during the process.

After the distribution of work, the processors insert simultaneously the points. In each step, the cavity to be retriangulated is found. If the cavity crosses the boundaries, it is necessary to use some synchronization of processors sharing the simplices in this cavity before the retriangulation can be performed. The new simplices are redistributed heuristically over the participants in order to balance the load of the processors and to minimize the length of the boundary. *Figure 4.2* shows an example of the retriangulation of a shared cavity.

In their recent work [Chr99], the authors present that the speed-up is nearly linear due to the used heuristics, but there is neither proof nor experimental evidence for this statement.



a) the original partition *b) the shared cavity* *c) the cavity retriangulation*

Figure 4.2 (see also Color Plates): *The insertion of a point (big dot) into the triangulation in E^2. This insertion causes retriangulation of a cavity shared by three processors (PE1, PE2 and PE3) including the update of their regions' boundaries.*

4.2.2 Pupo et al.

A parallel algorithm based on incremental insertion with local transformations was proposed by Puppo et al. [Pup94]. It works with a dense regular grid of points in E^2 only and it is determined to be used for terrain triangulations. The algorithm does not construct the Delaunay triangulation of all given points; it triangulates only such a subset of points that the error between the approximated triangulation and the full triangulation does not exceed a required threshold.

At the beginning, only two triangles containing the four corners of the grid exist. Each input point, as well as each triangle, is allotted to one virtual PE. The parallel algorithm consists of a loop with three phases that are performed in parallel. In the first phase, for each triangle, the processor responsible for this triangle founds all yet non-used points such that they are, in the planar projection, overlapped by the projected triangle. The vertical distance between each point and its approximation lying on the tested triangle is computed. The point with the maximum distance is chosen to be inserted – see *Figure 4.3*. Let us note that if this maximum distance does not exceed a given threshold, the insertion is omitted. If there is no point to be inserted after the evaluation of all triangles, the algorithm finishes its work.

The points that have been just chosen are inserted into the triangulation in the second phase. When a processor wants to insert a point that lies inside a triangle (as in the example in *Figure 4.3*), it inserts the point locally (using the incremental insertion principle). However, in a case that the point lies on a common edge of two triangles, only one point per these two triangles can be inserted and, therefore, the processors owning these triangles have to synchronize their work. After the synchronization of PEs, the processor with higher priority continues with the insertion. A higher priority is given to the processor holding the triangle whose point to be inserted lies further from the approximation.

In the last phase, the Delaunay triangulation is restored. It involves the detection of all non-optimal edges followed by the local transformations. Mutual exclusion similar to the previous one must be solved also in this phase.

The algorithm was implemented on Connection Machine CM-2 with 16K processors, compared with a serial implementation on Sun SPARC1 and tested up to 512^2 points. The speed-up was up to 80 for 16K points. The highest speed-up was achieved for the smallest

allowed approximation error because in such a case, more triangles are necessary and workload balance is improved.

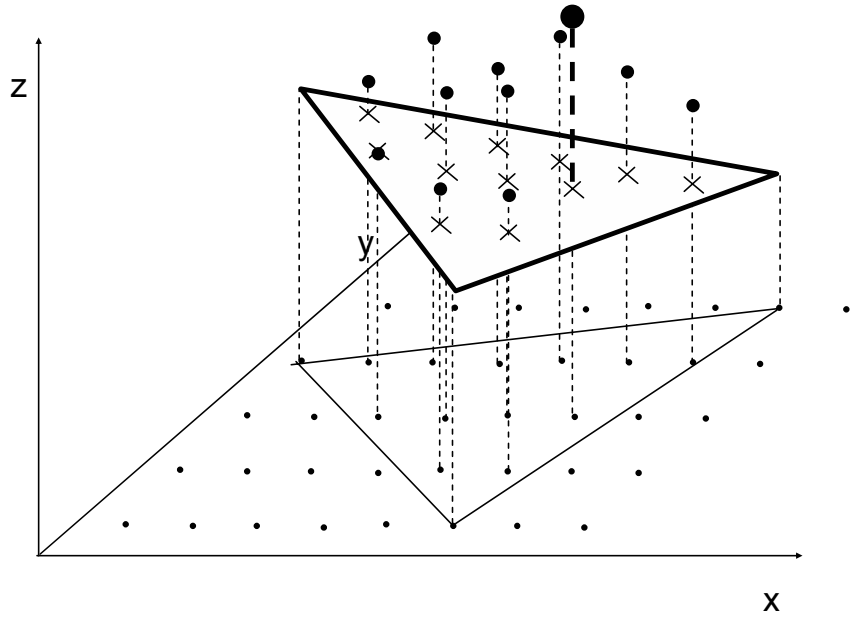


Figure 4.3: *A triangular patch (bold triangle) and its appropriate set of input points (big dots). The point with the worst approximation (the approximations are marked by crosses) is shown bigger.*

4.2.3 Okusanya et al.

Okusanya et al. [Oku96] developed a parallel version of Chew's refinement algorithm in E^2. The primary triangulation is partitioned among processors. If the insertion of a new point also affects remote triangles (i.e., the triangles not physically present at the current processor), the processor 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 inconsistency and send them in a compressed way to the initiator. For communication, MPI [MPI] or PVM is used. When the initiator completes its operation, all remote triangles are unlocked. However, in such a case that a contacted processor 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 itself on another point. Although authors also implemented some load balancing, they reached the speed-up only 1.6 – 3.0 for 2 – 8 PEs at IBM SP2 for uniform data set of 1 000 000 points. Probably it is caused by the time-consuming communication between PEs. Let us note that the authors use a similar strategy also in E^3 [Oku97]; their speed-up is roughly 1.2 – 2.3 for 2 – 8 PEs and 144 600 points. In our opinion, the reached speed-up is quite low.

Probably better results could be achieved by the approach by Spielman et al. [Spi02] because it is able to determine quickly which points can be inserted without any synchronization. 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.

4.3 Divide & Conquer (D&C)

The parallelization of D&C algorithms seems to be straightforward, no wonder that parallel algorithms based on the D&C approach dominate. A naive parallel solution (e.g., Aggarwal et al. [Agg88]), however, suffers from a serious drawback: the merging of two local triangulations is limited to just one processing element (PE) and thus it negatively influences the overall efficiency of the algorithm.

4.3.1 DeWall

The DeWall algorithm by Cignoni et al. [Cig93] uses a slight modification of naive D&C strategy. In the divide phase, a cutting plane α separates the points to be triangulated into two groups. The simplices intersected by this cutting plane are constructed and then the simultaneous triangulation (based on the incremental construction) of both parts is started. No complex merge phase is required, the final triangulation is obtained by a simple union of all three triangulations (i.e., local triangulations of both parts and the joint of the simplices intersecting the cutting plane) – see *Figure 4.4*.

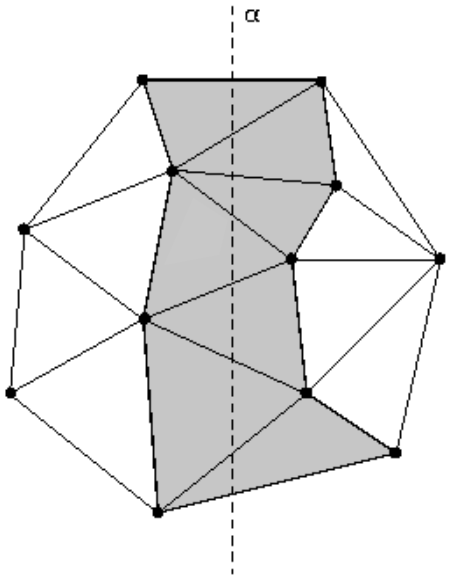


Figure 4.4: *The triangulation by DeWall in E^2 [Mag98]*

A natural solution is to start a new process in each step of the recursion and assign it one half of the points. The second half will be processed by the currently running proccess. However, as the dynamic starting of the processes could be expensive on a parallel architecture with the distributed memory, [Cig93] recommend to start all required processes, say k processes, at the beginning. In such a case, all processes run the same task up to the level of recursion log k and their intermediate results are, except for one per level, discarded.

Authors presented results of their algorithm for $DT(S)$ in E^3 (however, it may be used in E^2 as well). The experiments ran at nCUBE 2 system model 6410. For example, speed-up 1.70 – 3.35 for 2 – 16 PEs was noticed when the uniform data sets with 8 000 points were tested. The achieved speed-up is relatively low because of workload imbalance. Hardwick [Har97] claims that the situation is much worse (up to ten times) if we consider non-uniform data sets.

4.3.2 Chen et al.

Chen et al. [Che01] use an approach similar to DeWall. In their algorithm, which is intended to be used in E^2 only, the input points are subdivided according to their coordinates into k rectangular areas (where k is the number of processors). Each processor is responsible for the triangulation of points lying in one area. To fulfill its task, the processor requires to have available all points lying in the areas adjacent to the area assigned to this processor (there are at most four such areas). As the processor does not need the whole input set (in contrast to InCode), the algorithm is able to triangulate huge data sets.

The processor triangulates its 'central' area by the fastest sequential algorithm [Dwy86], which is based on the divide & conquer principle. Only points lying in this area are required. Then, an "interface" is constructed at each boundary using the principle of incremental construction. The interface is such a set of triangles that crosses the area's boundaries – indeed, the knowledge of the points of adjacent areas is needed. It is quite clear that we have two interfaces (constructed by two processors) at the same boundary.

The merging of results of two processors consists of two stages. In the first one, both interfaces at the shared boundary are merged together in order to get a wall of Delaunay triangles. Then this resulting wall (or joint) is combined successively with both triangulations. Thanks to the interface methodology, this second stage involves only removal of triangles from both triangulations such that they overlap the constructed wall. The final triangulation is obtained afterwards by a simple union of all three products. *Figure 4.5* shows the merging of two local triangulations and their wall of the points lying inside a circle.

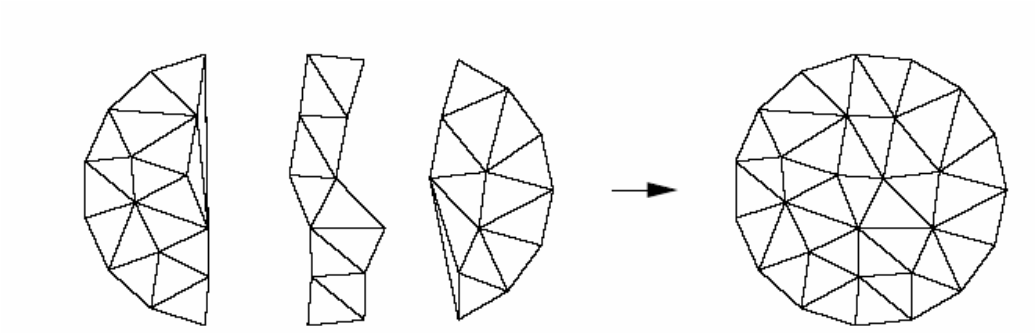


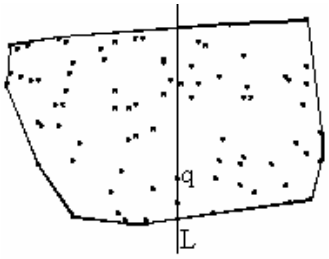
Figure 4.5: *The Delaunay triangulation given by a merge of two local triangulations and its common interface (created from two local interfaces) [Che01]*

The entire algorithm, except for the merge phase, can be processed in parallel. Time needed for the merge phase is, however, negligible (thanks to interfaces) in comparison to other phases. 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 at IBM SP2 with High Performance Fortran.

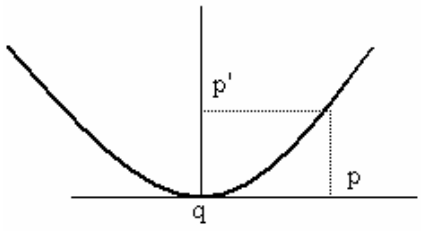
4.3.3 Hardwick

Hardwick [Har97] chooses another approach allowing avoidance of the merge phase. Input points are subdivided recursively into two groups by the orthogonal line L that goes through a median q in x-coordinate (or y-coordinate at even levels of the recursion) – see *Figure 4.6a*. As it is not necessary to compute the exact value of the median, the author uses a very simple parallel algorithm for its computation: each processor computes a median of its local points

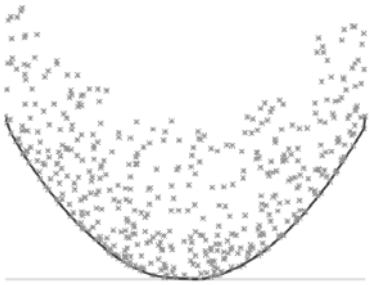
and then median of these medians is found by any sequential algorithm and the achieved value is picked as the representative "median".



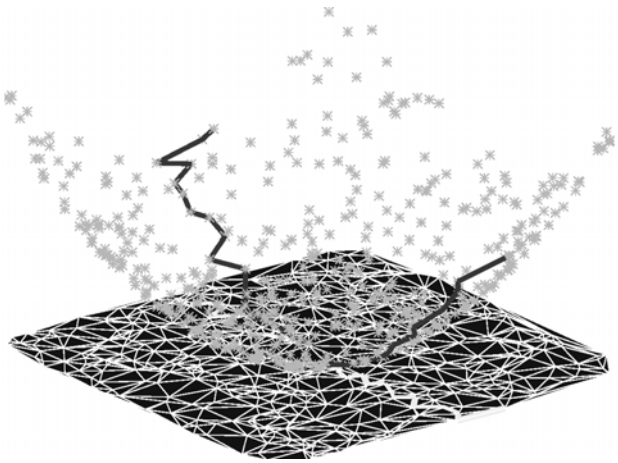
a) the median line L going through the median point q



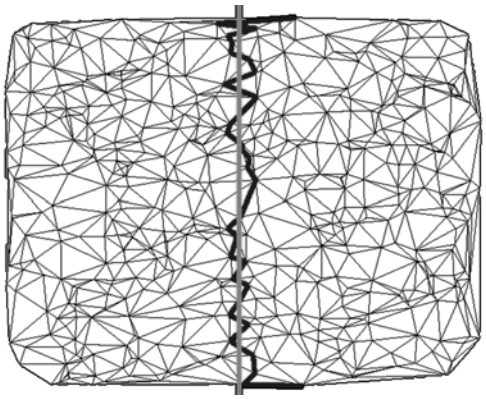
b) the projection of the point p on the vertical plane using the paraboloid centered at q – perpendicular view



c) the projected points and their lower convex hull (bold line)



d) the points and the lower convex hull (dark bold line segments) projected back onto the paraboloid and its relation to the resulting triangulation



e) the Delaunay triangulation with a median line (thin gray line) and a joint (thin black line segments)

Figure 4.6: *The construction of the Delaunay triangulation by Hardwick's approach. The images in c, d and e were adopted from [Har97]*

The main algorithm continues by the construction of a paraboloid in E^3 centered at the median q and by the construction of the projection plane yz (or xz at even levels of the recursion) containing the median-line L. Then the algorithm transforms all points onto the projection plane using the paraboloid for this purpose, i.e., the coordinates of the transformed point p' are equal to $(p_y\text{-}q_y, \ ||p\text{-}q||^2)$ where p is the original point from the input set. Let us note that at even levels we have to use x-coordinates of p and q. The projection is shown in *Figure 4.6b*.

The lower convex hull of the projected points is found by a parallel modification of simple quickhull [Pre85] – see *Figure 4.6c* and the back projection of the resulting lower convex hull

into the plane *xy* gives a set of line segments – see *Figure 4.6d*. It was proven that these line segments form a joint that has to be present in final Delaunay triangulation. This joint separates the input region with the input points into two non-convex sub-regions. Both sub-regions are simultaneously triangulated by Dwyer's sequential algorithm [Dwy86] and the Delaunay triangulation is obtained by a simple union of both local triangulations. The resulting triangulation is given in *Figure 4.6e*.

Although the algorithm by Hardwick uses a higher dimension, it cannot be included into the higher dimensional embedding category because this higher dimension is not used for the construction of the Delaunay triangulation but only for the subdivision of the set of points. In our opinion, the algorithm is quite complicated. Moreover, it is limited to E^2 only. On the other hand, it achieves a very good speed-up, e.g., about 1.8 – 4.8 for 2 – 8 PEs at SGI Power Challenge with shared memory for the uniform data sets with 128K points. There are several reasons for such a very good speed-up. First, the use of median ensures that each processor has almost the same workload. Next, the algorithm does not need the merge phase and the divide phase, i.e., the described subdivision of input points, is solved in parallel.

The just described algorithm was improved by Lee S. et al. [Lee01]. Their algorithm does not recursively subdivide the input points into two groups via a median line but it 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 evident that also a better speed-up is reached. The experiments were done at INMOS TRAM network with 32 T800 processors. Their algorithm achieves speed-up 1.36 – 12.5 for 2 – 32 PEs and uniform data. Better behavior of the algorithm is presented for cluster data; the speed-up about 16.9 for 32 PEs was measured. Let us note that an objective evaluation of this algorithm is impossible because the authors do not present the numbers of points of their data sets.

5 Incremental Insertion with Local Transformations

In previous sections, we described briefly sequential principles for the construction of the Delaunay triangulation and best-known existing parallel algorithms. In this section, we describe a sequential incremental insertion algorithm with local improvements that was chosen as a base for our parallel solution (will be discussed in next sections).

Let us remind that incremental insertion algorithms insert the points in the input set S one at a time into an already existing Delaunay triangulation. It consists of three phases: the *location* where a simplex containing the point to be inserted has to be quickly found followed by the *subdivision* of this simplex and by the *legalization* where the circum-sphere criterion is applied and if it is necessary, the local improvement techniques are used to restore the Delaunay triangulation. The algorithm for the construction of $DT(S)$ by the method of incremental insertion with local transformations is given in *Figure 5.1*.

Input: A set $S = \{ p_0, p_1, ..., p_{N-1} \}$ of N points in E^2

```
for r := 0 to m - 1 do begin
   Locate the simplex S0 ∈ DT(S) containing pr in the DAG structure;

   Subdivide S0;  //in the case where pr lies on the shared edge or face
                  //then subdivide also the appropriate adjacent simplices.

   //Legalize all new simplices
   while there exist an unchecked face F do
      if the face F violates DT criterion
               then perform local transformation;
end;
```

Figure 5.1: *Construction of the DT(S) by incremental insertion with local transformations.*

First, let us to explain why we choose an algorithm that is not the fastest one for the parallelization. It is true that the incremental insertion algorithm with local transformations has $O(N^2)$ complexity in the worst-case and, therefore, it is not worst-case optimal. Better complexity $O(N \cdot \log N)$ in the expected case can be reached if some accelerating structure, such as already mentioned the DAG, is used. However, this algorithm has many advantages over others. First, it is very simple to understand and implement. There is no significant difference in implementations of the version for E^2 and of the version for E^3. It is also relatively robust: in the case of an incorrect or inconsistent Delaunay criterion evaluation caused by numerical inaccuracy, a triangulation with two or more non-Delaunay simplices is obtained, but it is still a valid triangulation. There are no holes or mutually overlapping simplices that may result from other methods (e.g., according to our experience, incremental construction or D&C algorithm, or Bowyer-Watson's incremental insertion [Gol97]). Moreover, the algorithm can be simply modified to incorporate constraints given in the form of prescribed edges (or faces in E^3) [Vig97], to use non-Euclidian metrics [Oka92, Vig00] or weights of points. Not all input points need to be available at the beginning of computation (although the range of their coordinates is required), which can be also an advantage for some applications. As the algorithm uses a randomized order of insertion and the DAG structure for the location of simplices, it becomes almost insensitive to the type of point distributions.

5.1 Initialization

Let us have the input set S of N points. An auxiliary simplex large enough to hold all these points inside its interior is constructed. We prefer this large simplex to the convex hull (see Section 3) because it is easier and, according to our experience, more stable. One problem with this approach is how to choose the vertices of this simplex. If they are not far enough away, they may influence the empty circum-sphere tests, which may lead to the non-convex boundary of the resulting Delaunay triangulation. On the other hand, if the vertices are "too far away", it may lead to numerical instability of the algorithm.

Therefore, in our algorithm, the vertices have coordinates (K, 0), (0, K), (-K, -K) for the version in E^2 and (K, 0, 0), (0, K, 0), (0, 0, K), (-K, -K, -K) for the version in E^3. The value K is equal to the multiple of the size of the bounding box of points - see *Figure 5.2*. More detailed description is given by Žalik and Kolingerová in [Žal03].

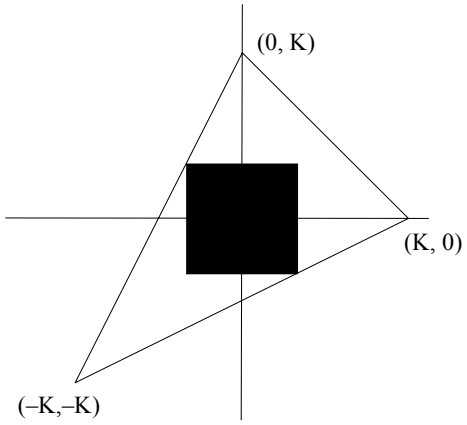


Figure 5.2: *The selection of the auxiliary simplex in E^2. The black rectangle is the bounding box.*

5.2 Location

In the location part, it is required to find a simplex to be subdivided. It can be done either with use of some hierarchical structures or without them. Let us describe two different approaches for the location and discuss its advantages and disadvantages.

5.2.1 The Directed Acyclic Graph – DAG

The Directed Acyclic Graph (DAG) is a hierarchical structure that stores the history of changes in the Delaunay triangulation and resembles tree. Each node of this graph describes one simplex; the root contains the auxiliary large simplex. If we want to find a simplex containing the current point to be inserted, we take the root in the DAG and then test the mutual position of the point and the simplices stored in the children nodes. When the relevant child with a simplex containing the point is found, the process continues until a leaf of the DAG is reached. This leaf describes a simplex that has to be subdivided. When the simplex is subdivided (in the subdivision phase), new nodes are created and joined to the node that stores the subdivided simplex. Later, in the legalization phase, more simplices are transformed in one operation, i.e., we have more input nodes. Two or more new nodes are created and they are joined to all input nodes. As the DAG structure stores the full history of the changes of the Delaunay triangulation, it consumes a lot of memory – $O(N^2)$ in the worst-case. On the other hand, the location of one point in this data structure is possible in $O(\log N)$ expected time (and it is also optimal time) and $O(N)$ worst time (worst time happens when the DAG is "totally

imbalanced", having the shape of a list – if the order of insertion of the input points is randomized, such a situation is highly improbable). More details about the structure can be found in [Ber97]. *Figure 5.3* shows the changes in the triangulation and the corresponding changes of the DAG structure. We describe the possible changes in the following text.

Let us discuss the allocation and the destruction of the DAG structure. Native solution is to allocate new nodes successively during the process, i.e., to allocate the memory for a new node when it is demanded. This strategy implies that the nodes have to be successively destructed at the end of the triangulation. Typically, it involves a recursive traversing of the DAG structure. As one node may be accessed more times via various routes (at most twice in E^2) because of local transformations, it is necessary to count the number of still not passed routes and do not destruct the node until this counter is zero.

There is another strategy. It is a bit more complicated and consumes additional memory, however, using this strategy speed-ups the algorithm significantly. According to our experiments, the number of nodes in the DAG after the insertion of all points is about 6 up to 10 times larger than is the number of points. If we allocate a continuous block of memory capable enough to hold $6 \cdot N$ nodes and provide the main algorithm with pointers on these nodes when they are demanded, the destruction of the entire DAG structure is very fast – only a few (usually up to 3) large blocks of memory are deallocated.

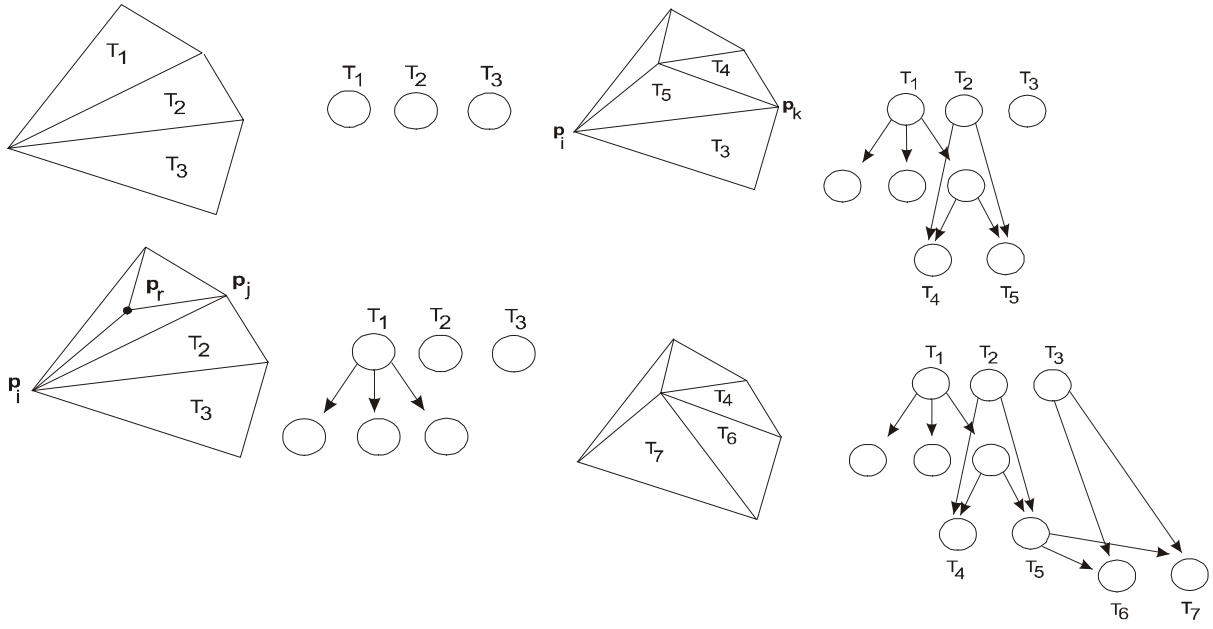


Figure 5.3: *The changes in the planar Delaunay triangulation caused by the insertion of the point p_r and the corresponding changes of the DAG structure.*

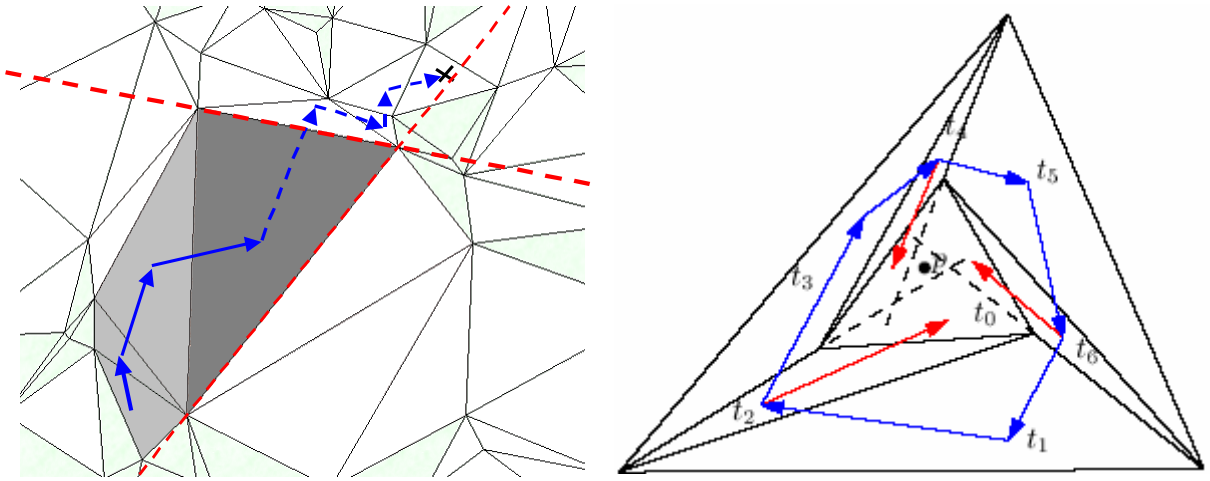
5.2.2 Remembering Stochastic Walk

Walking techniques are based on the searching of simplex to be subdivided directly in the Delaunay triangulation. Therefore, the location can take $O(N)$ time in the worst-case. Fortunately, the worst-case scenario is not very probable and the location is, usually, performed in $O(\sqrt{N})$ expected time. Let us note that under special circumstances expected time $O(N^{1/3})$ can be reached [Žal03]. Although walking approach is slower than the approach with the DAG structure, its big advantages is that it needs no additional memory, which is

appreciated in the case that we need to process a big data set. Different walking techniques are presented in [Dev01].

Let us describe *visibility* walk in E^2. The E^3 case is similar: triangles just have to be replaced by tetrahedra and edges by faces. Starting from an arbitrary triangle, the algorithm traverses through the triangulation testing the mutual position of visited triangles and the given input point until the triangle containing this point is found. For each visited triangle, it is necessary to detect an edge such that the line supporting this edge separates the triangle from the input point, which can be reduced to a single orientation test. If there is no such edge, the triangle contains the point in its interior. Otherwise, the search continues with the neighboring triangle sharing the detected common edge. *Figure 5.4a* shows an example of walk.

Unfortunately, for non-Delaunay triangulations, the walk we have just described may fall into a cycle as illustrated in *Figure 5.4b*. As the constrained Delaunay triangulations (i.e., with some prescribed edges – will be discussed in further text), which are important in practice, are also non-Delaunay, a little bit of randomness has to be introduced into the algorithm in order to avoid infinite loops. Instead of starting the detection with the first edge of the given triangle, the algorithm starts with randomly picked edge. This ensures that, if the walk enters a cycle in the triangulation, it cannot loop in this cycle forever. Another small improvement is to remember, for each visited triangle, the edge that was just crossed by the walk and do not test this edge twice. The visibility walk algorithm with these two improvements is called *remembering stochastic walk*.



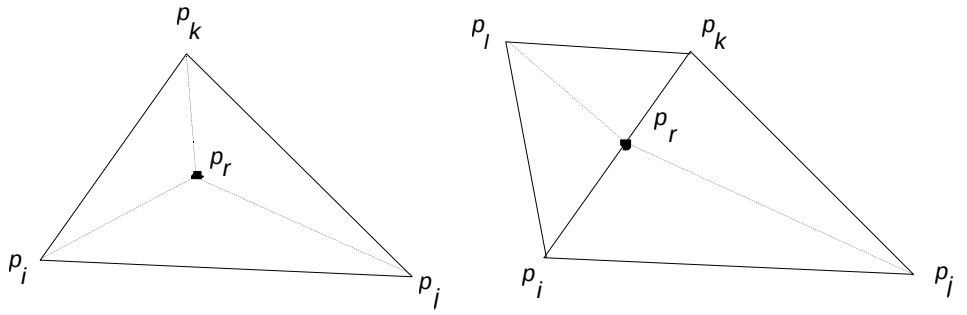
a) the path of visibility walk, the dark gray triangle is currently being tested, light gray triangles were visited in previous steps

b) an infinite cycle for the visibility walk [Dev01]

Figure 5.4 (see also Color Plates): *The visibility walk algorithm.*

5.3 Subdivision

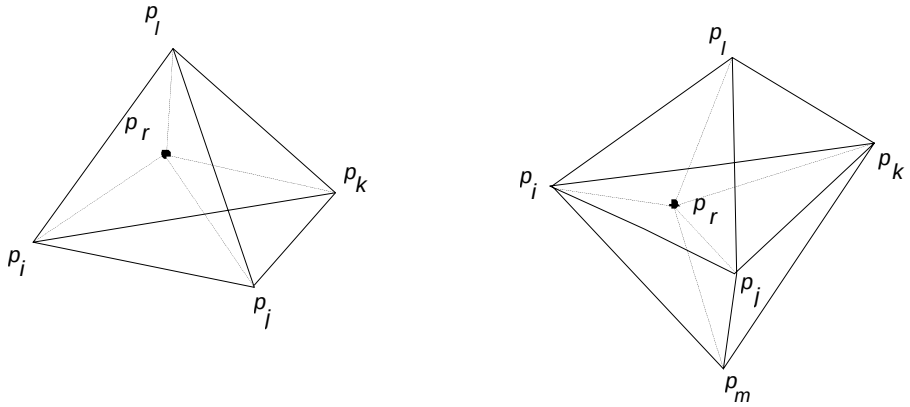
Let us suppose we have successfully found the triangle p_i, p_j, p_k (or the tetrahedron p_i, p_j, p_k, p_l in case of E^3) containing the point p_r to be inserted. There are several mutual positions of this point and the located simplex. The simplest possible configuration is that the point lies strictly inside the simplex. In this case, all vertices of the located simplex are connected with the point by an edge and the simplex is subdivided into several new simplices. These are three triangles in E^2 (see *Figure 5.5a*) and four tetrahedra in E^3 (see *Figure 5.6a*).



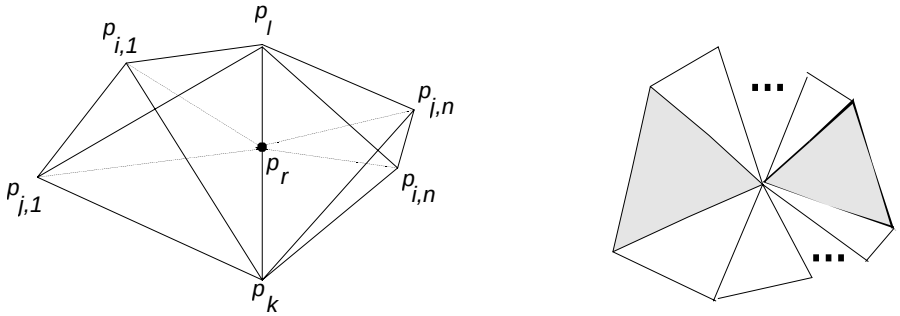
a) the point to be inserted lies strictly inside *b) the point to be inserted lies on an edge*

Figure 5.5: *Subdivision in E^2.*

Slightly more complicated situation occurs when the point to be inserted lies on an edge (for E^2 only) or on a face (for E^3 only). In both cases, it is necessary to subdivide not only the located simplex but also the adjacent simplex that shares this edge or face. It results in four new triangles (see *Figure 5.5b*) or six tetrahedra (see *Figure 5.6b*).



a) the point to be inserted lies strictly inside *b) the point to be inserted lies on a face*



c) the point to be inserted lies on an edge; only two tetrahedra are shown - see the projection to xy-plane in the right to see their position in space

Figure 5.6: *Subdivision in E^3.*

If the point to be inserted lies on an edge of the simplex in E^3, all simplices sharing this edge have to be subdivided. In the worst-case, all simplices in the triangulation will be subdivided. Each simplex is subdivided into two new simplices (tetrahedra) – see *Figure 5.6c*. Let us note

that the handing of this case is time consuming and its implementation is quite difficult. Therefore, when it is detected that the point lies on an edge, its insertion may be postponed in the hope that, as the triangulation dynamically changes, this edge will no longer exist in the triangulation when the point arrives later again. Another option is to shift slightly the point in a random direction and thus avoid insertions on edges at all. As this option introduces an inaccuracy, it is allowed only in some applications.

5.4 Legalization

After the subdivision, we have a new triangulation. However, it may not be the Delaunay one. Therefore, all outer edges (or faces in the case of E^3) of currently created simplices have to be tested whether they do not violate the empty circum-sphere criterion, i.e., whether the far point of the simplex adjacent to the new one does not lie inside the circum-sphere of this new simplex. If the condition is not fulfilled, the triangulation has to be changed by applying the local transformations. The transformation in E^2, which is shown in *Figure 5.7*, is simple: the edge is just swapped (see also *Figure 4.2*).

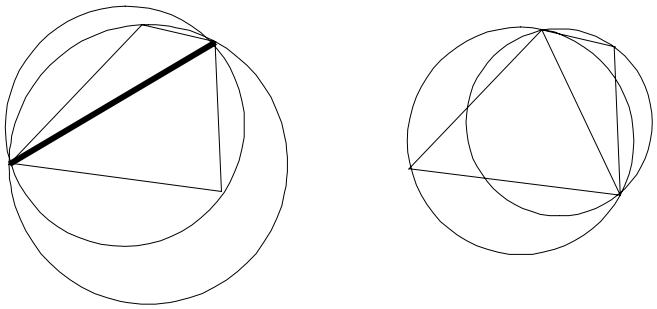
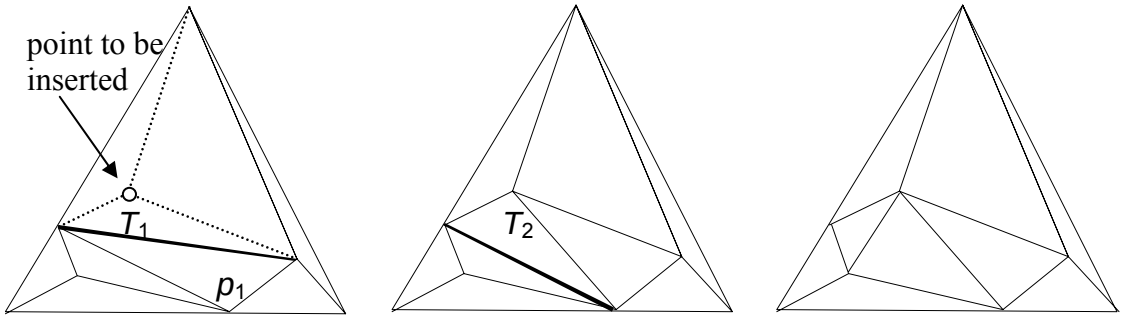


Figure 5.7: *Local transformations in E^2. The edge is swapped.*

After that, indeed, we have new outer edges (or faces) that have to be tested. *Figure 5.8* shows an example of the propagation of the local transformations in E^2. The located triangle is subdivided into three new triangles (*Figure 5.8a* – dotted line). Then, the circum-circle criterion is tested on all these new triangles. The test for the triangle T_1 fails because the far point p_1 of the adjacent triangle lies in the circum-circle of the triangle T_1. The shared edge is flipped. As the circum-circle of the just created triangle T_2 is not empty, the flipping has to continue – see *Figure 5.8b*. Finally, the Delaunay triangulation is achieved (*Figure 5.8c*).



a) subdivision *b) propagation of flips* *c) the resulting triangulation*

Figure 5.8: *The incremental insertion in E^2. Edges that should be flipped are bold.*

While in E^2 two simplices were replaced by new two simplices, the situation in E^3 is not so simple. The local transformations [Joe91] are applied on a set of two to four simplices and result in new up to four simplices. Let us have a pair of tetrahedra with vertices p_i, p_j, p_k, p_l and p_m sharing the illegal (i.e., needed to be changed) face p_i, p_j, p_k. If the line segment p_l, p_m intersects this face and does not intersect any edge of the face, then this pair of tetrahedra is replaced by three tetrahedra p_i, p_j, p_l, p_m; p_i, p_k, p_l, p_m and p_j, p_k, p_l, p_m – see *Figure 5.9*.

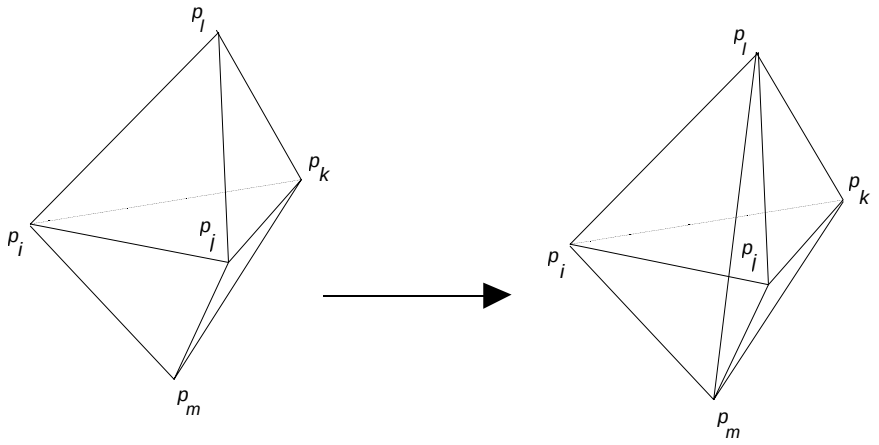


Figure 5.9: *Local transformation of two adjacent tetrahedra p_i, p_j, p_k p_l and p_i, p_j, p_k, p_m sharing illegal face p_i, p_j, p_k into three tetrahedra p_i, p_j, p_l, p_m; p_i, p_k, p_l, p_m and p_j, p_k, p_l, p_m.*

If the line segment p_l, p_m does not intersect the face p_i, p_j, p_k and the simplex p_i, p_j, p_k, p_m is available, this triple of tetrahedra is replaced by a pair of tetrahedra – see *Figure 5.10*.

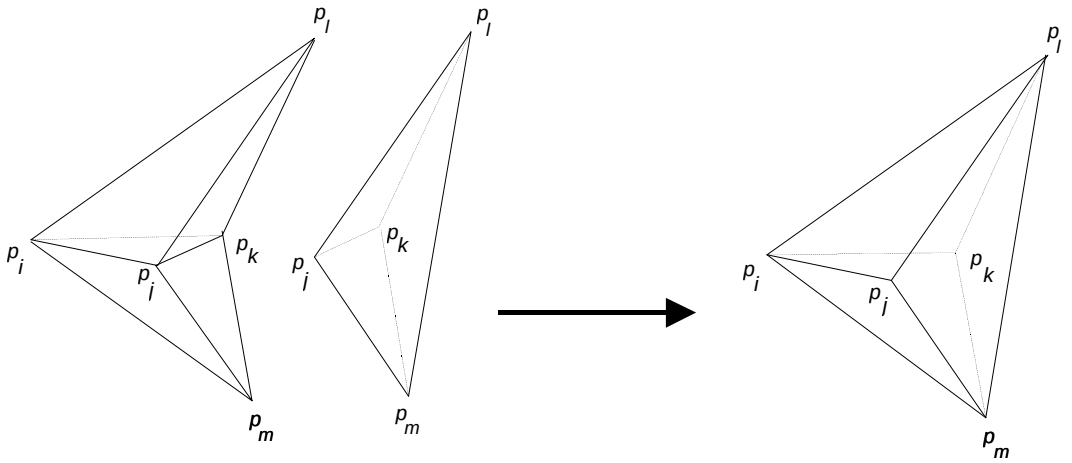
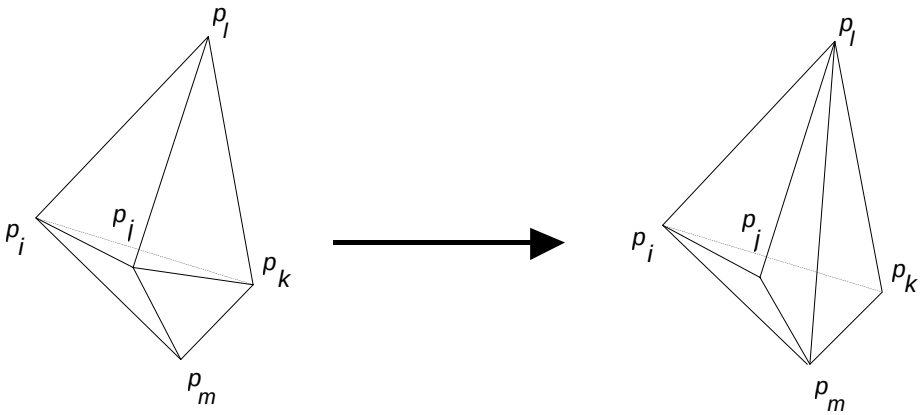


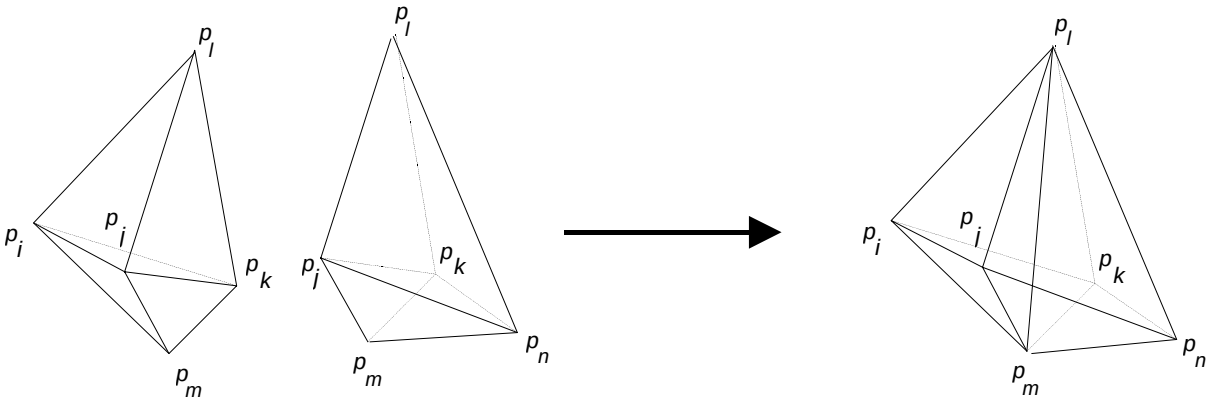
Figure 5.10: *Local transformation of three adjacent tetrahedra (tetrahedron p_j, p_k, p_l, p_m is shown separately to increase readability) p_i, p_j, p_k, p_l; p_i, p_j, p_k, p_m and p_j, p_k, p_l, p_m into a pair of tetrahedra p_i, p_j, p_l, p_m and p_i, p_k, p_l, p_m.*

A more complicated situation occurs if the line segment p_l, p_m intersects some edge of the illegal face p_i, p_j, p_k. In such a case, the faces p_j, p_k, p_l and p_j, p_k, p_m are coplanar and may or may not be shared with other two tetrahedra. If these tetrahedra do not exist, the illegal face is simply swapped and new pair of tetrahedra is created, otherwise we need also to swap the face

between the adjacent pair of tetrahedra, i.e., new four tetrahedra are generated. *Figure 5.11* shows these local transformations.



a) Local transformation of two tetrahedra into two tetrahedra p_i, p_j, p_l, p_m and p_i, p_k, p_l, p_m.



b) Local transformation of four tetrahedra into four tetrahedra p_i, p_j, p_l, p_m; p_i, p_k, p_l, p_m; p_j, p_l, p_m, p_n and p_k, p_l, p_m, p_n.

Figure 5.11: *Local transformation of tetrahedra with the coplanar faces p_j p_k p_l and p_j p_k p_m.*

Let us note that to perform a transformation, it is necessary to find all simplices adjacent to the tested one. Adding pointers on the neighbors into the data structure describing simplex is the simplest and the most efficient solution.

5.5 Finalization

Another small problem appears when the construction has been finished and the algorithm should extract the triangulation from the leaves. The most efficient solution is to have the leaves in a bidirectional list and know (or find) the head of such a list. Therefore, pointers 'Next' and 'Last' were included into the data structure. Let us note that all simplices having at least one vertex of the big auxiliary simplex have to be removed from the triangulation.

6 Parallelization Problem

Modern computer architectures allow us to compute the Delaunay triangulation in E^2 or E^3 with thousands of points by a sequential algorithm in reasonable time. However, current applications often need to work with data sets such that they cannot be computed in one piece because of the common memory size limitations or their processing consumes too many time. In such cases, a parallel algorithm is useful and welcome. We can identify two different parallelization purposes. The first one is to compute the Delaunay triangulation of set of points in as shortest time as possible without considering memory limitations. A different problem is to compute the Delaunay triangulation of very large input data sets where the final time is not as important as the memory utilization of PEs.

Quite a big set of parallel algorithms exists (see Section 4), however, they were designed in times when special parallel architectures, with hundreds of processors, dominated in the research area and thus they put stress usually on the scalability rather than on the robustness and simplicity. In the last few years, symetric multiprocessors with several processors and shared memory (especially two-processors) and clusters of workstations (especially computer networks) have become very popular due to their low prices. Many existing algorithms can be used after some modifications for these hardware architectures. However, it is a question whether the efficiency of a modified parallel algorithm is still good enough. Moreover, there is no doubt that the modified algorithm is often unnecessarily complicated for the low-degree of parallelism. This led us to the idea to develop a new algorithm that can process small data sets quickly on architectures with a limited number of processors (typically 2 or 4) and with a shared memory, and to develop a new algorithm suitable for clusters of workstations that can process large data sets. We have chosen the randomized incremental insertion algorithm described in Section 5 as a base for our parallel algorithms because of its good advantages, mainly because of its simplicity or robustness (see the previous section for more details).

In this section, we discuss general problems of parallelization of incremental insertion with local transformations common to both parallelization purposes (i.e., speed and unlimited size) such as analyzis of required synchronizations or distribution of work over processors.

6.1 Analysis of the Sequential Algorithm

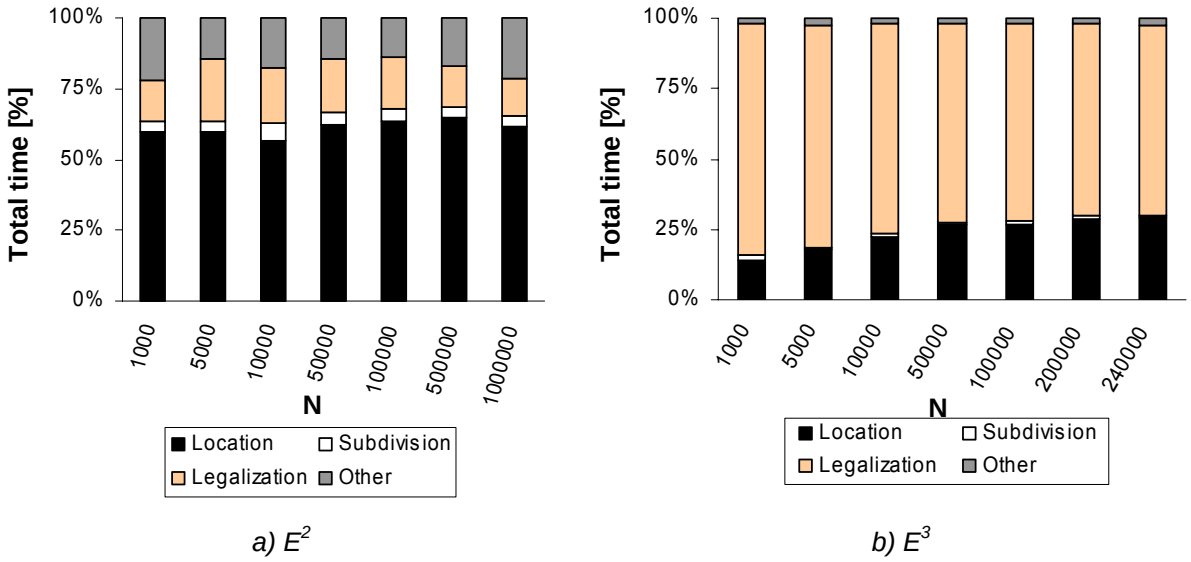
Let us remind that incremental insertion algorithms described in the previous section insert the points in the input set S one at a time into an already existing Delaunay triangulation. The order of insertion does not influence the result, i.e., it does not matter if some point p_1 is inserted before another point p_2 or vice versa. It is a very important fact because it allows us to use SPMD or MPMD paradigms (see Section 2) in order to parallelize the given algorithm.

The main idea is to let several processors to insert the points simultaneously into the Delaunay triangulation that is stored in the shared memory in the case of symmetric multiprocessors or distributed over local memories in the case of clusters of workstations. Let us note that the distribution of the Delaunay triangulation introduces several new problems such as how to ensure consistency between simplices stored on different computers. These problems will be discussed in Section 8.

The incremental insertion algorithm consists of three main phases: the location, followed by the subdivision and by the legalization. In all phases, the algorithm needs to access the data structure storing information about simplices (i.e., the Delaunay triangulation), however, in a different way. In the location, data structure is accessed for reading only in such a manner to find the simplex containing the point to be inserted. Afterwards all corresponding simplices

are subdivided and legalized, i.e., the data structure is modifed. The parallelization of the location phase will probably require different parallel programming techniques than the parallelization of remaining phases (i.e., the subdivision and the legalization).

Typical runtimes needed by various stages of the sequential algorithm with DAG hierarchical structure are in *Figure 6.1*. For remembering stochastic walk, time spent in the location phase is slightly larger. The majority of time in E^2 is consumed by the location phase (about 60 – 70%). The parts when 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 destruction of this structure, thus for the sequential part of the algorithm. The situation differs in E^3 where the most complex part is the legalization phase (about 65 – 80%) while the location phase needs up to 30%. According to this profiling, it seems to be not very probable that one parallel algorithm would work efficiently in both dimensions.

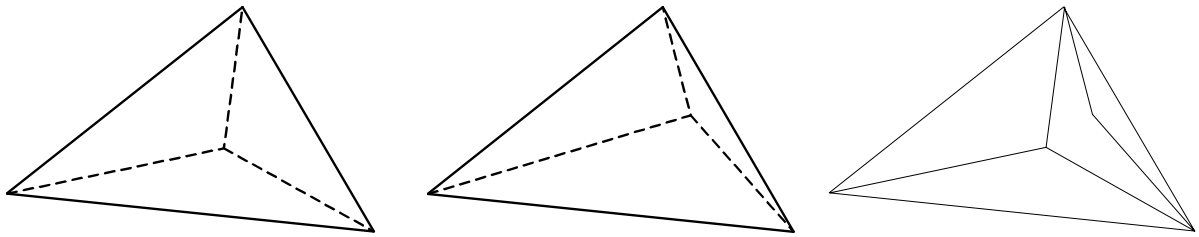


a) E^2 *b) E^3*

Figure 6.1: *Typical runtimes needed by various stages of the sequential algorithm with DAG. Uniform data sets were tested.*

Now, we discuss differences in the parallelization of phases in more detail. A simplex can be accessed simultaneously by several processors if all processors need it for read-only purpose. When any processor needs to modify it, some synchronization has to be implemented because otherwise the concurrency could produce artifacts in the resulting Delaunay triangulation or even could lead to the collapse of the program. The main reason for this behavior is that processors run in unpredictable speed. *Figure 6.2* shows one of the possible results of the unsynchronized subdivision phase when two points are to be inserted into the same triangle. Both processors performed subdivision using local data structures and started to update the common triangulation. The second processor started the update earlier than the first one, however, finished it later. This caused that the connectivity of two left most triangles updated by the second processor was overwritten by the information from the first processor.

It is quite clear that any synchronization takes additional time and an efficient parallel algorithm has to avoid synchronization as much as possible. There is also no doubt that the avoidance of synchronization in the subdivision or the legalization is impossible; we can just reduce time required for the synchronization. However, it could be done for the location.



a) subdivision to be performed by the first processor *b) subdivision to be performed by the second processor* *c) one possible result of the performed subdivisions*

Figure 6.2: *Simultaneous unsynchronized insertion in E^2.*

Let us suppose the DAG hierarchical structure used for the location. The algorithm modifies leaves only and, therefore, all parent nodes in the DAG may be tested in the location simultaneously without any synchronization. The detection whether the tested node is a leaf or not is straightforward and, therefore, we could expect that a parallel algorithm with DAG structure will be quite efficient.

For walking techniques, the situation is different because, in terms of the DAG approach, the algorithm works just with leaves. We can deal with this problem as follows. If a node (i.e., the data structure of simplex) is currently being modified, the simpex cannot be tested immediately but the processor has to wait until the modification of this node is finished. The detection whether the test can be performed without synchronization can be done, e.g., by a check of flag added into the node data structure. Indeed, operations (set, clear, check) with this flag consume some time and, therefore, the performance of proposed parallel algorithms is reduced.

Let us note that a parallel algorithm using another hierarchical structure for the location, e.g., [Dev98], should achieve efficiency lower than the same algorithm with the DAG structure, however, higher than the same algorithm exploiting a walking technique.

6.2 Subdivision of Input Points – workload

To achieve efficiency as close to the optimal efficiency as possible, we have to minimize the total time spent inactively, e.g., because of synchronization or communication between processors. This requires minimizing of the number of cases when a processor needs to access data stored remotely or currently modified by another processor. As the processor accesses, usually, only the nodes that store simplices lying closely to the position of currently inserted point, there is no doubt that the way how input points were assigned to processors significantly influences the performance of given parallel Delaunay triangulator.

For symmetric multiprocessors, all data is stored in the shared memory and, therefore, all that is needed is to reduce the amount of required synchronization. The simplest possible strategy is to subdivide points randomly into k groups (k is the number of available processor) in such a manner to ensure equal number of points in each group. Due to this randomization, it is highly improbable that two processors insert two close points simultaneously.

For clusters of workstations, the described strategy is useless because it would require a lot of communication to retrieve remotely stored data. Better seems to be a subdivision of input points into k groups in such a manner that not only the equal number of points in each group is ensured but also the bounding boxes of these groups have minimal intersection. In an ideal case, all triangles (tetrahedra) created by one processor then form just one fragment, i.e., a

continuous area (volume). A simple way is to subdivide input points into k slabs in x-coordinate. There is one serious drawback with this simple solution: for non-uniform data sets, points are not subdivided equally and, therefore, workload is imbalanced. Moreover, when k is larger, the total length of boundaries among groups is larger and slabs are very narrow, which leads to a larger number of required synchronizations. Indeed, the problem with too narrow areas could be simply handled by dividing the input points using cuts not only in x-coordinate but also in y-coordinate (and in E^3 in z-coordinate). It, however, does not solve the problem with imbalanced workload in the case of non-uniform data sets. Therefore, let us describe other two more general strategies.

Whelan [Whe85] proposed an algorithm originally used for the problem of parallel ray-tracing. Starting with bounding box of points, the algorithm recursively divides this box into groups until the number of groups equals to the number of available processors. In each step of the recursion, median of w-coordinates, where w denotes the dimension of longest side of the current box, is found and box is cut into two parts at this place – see *Figure 6.3*. As the precise computation is not needed, an approximated method, e.g., [Bat99], for median computation can be exploited. Such a solution usually requires $O(N)$ time in the worst-case. Parallel versions of approximate median computations also exist; one of them can be found in Hardwick et al. [Har97]. Whelan's approach ensures even distribution of points and, therefore, balanced workload. On the other hand, the algorithm is suitable only for $k=2^m$ processors, where m is an integer. The median computation also takes a lot of time (especially for larger data sets); it requires $O(k \cdot N)$ time in the worst-case.

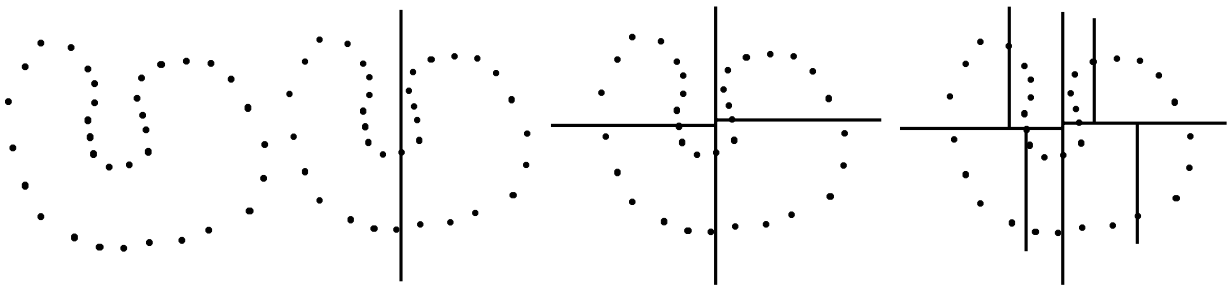


Figure 6.3: *A median based division of points in E^2.*

Mueller's approach [Mue97] developed for parallel ray-tracing overcomes disadvantages of the median based algorithm. It is also a recursive algorithm that splits the box into two parts in each step but unlike Whelan it does not need to achieve the same level of recursion everywhere. This allows exploiting an arbitrary number of processors. In the preprocessing stage, Mueller constructs a binary tree containing the plan of division as follows. Starting with k leaves evaluated by the value 1, the algorithm successively merges nodes until the root is achieved assigning to each inner node the sum of values of its children. Values in children of a node denote the ratio in which the box of this node should be split, e.g., having 5 processors, the first box will be divided into two parts in the ratio 3 to 2. *Figure 6.4* shows an example of tree construction for 5 processors.

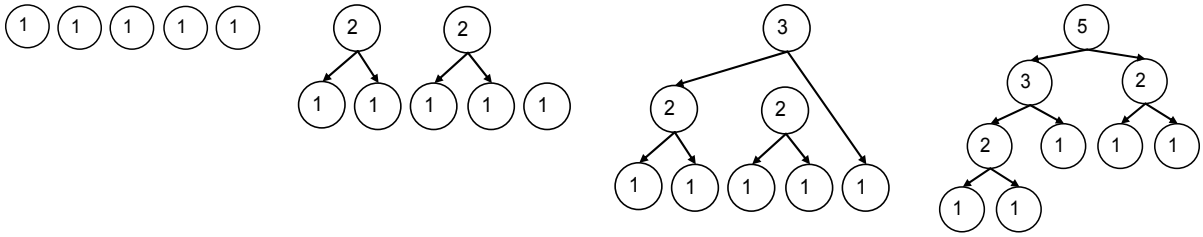
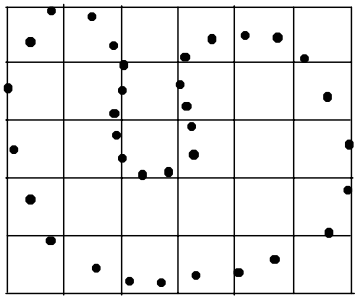


Figure 6.4: *The construction of division tree.*

Afterwards, a uniform grid covering the bounding box of points is created, each cell of this grid is evaluated according to the number of points lying in this cell and a summed-area table of the grid is computed. Summed-area table is such a table that the value of its cell at the position $[i, j, k]$ is equal to the sum of the values of the cells in the original grid at positions [0 up to i, 0 up to j, 0 up to k]. *Figure 6.5* shows an example of grid and summed-area table construction in E^2. Let us note that the summed-area table can be efficiently found in $O(R)$, where R is the total number of cells.



a) grid covering data

2	2	0	2	2	1
1	3	0	2	0	1
1	2	2	2	0	1
1	0	0	0	0	2
1	1	2	2	2	0

b) grid evaluation

2	4	4	6	8	9
3	8	8	12	14	16
4	11	13	19	21	24
5	12	14	20	22	27
6	14	18	26	30	35

c) summed-area table

Figure 6.5: *The construction of summed-area table in E^2.*

The input data set is recursively divided into k groups according to the plan introduced in the preprocessing stage of the algorithm and exploiting the summed-area table. In each step of the recursion, the dimension of the longest side of the box is picked and the appropriate position of cut in the table is found. Let us explain the process by an example in E^2 for 5 processors – see *Figure 6.6*. In the first step, we divide the summed-area table into two parts in ratio 3:2 in y-coordinate. In an ideal case, the first part of the summed-area will consist of $3/5 \cdot 409 = 245.4$ points. It is, however, impossible to find an exact position of cut to achieve this ideal case due to the error introduced by the bucketing of points into the grid. All that can be done is to find a cut that minimize the deviation from the ideal case as follows. First, the proper position of value 245.4 in the one-dimensional array 13, 42, up to 409 (i.e., the last column in the table) is found. As this array is ordered, we can use modified binary-search algorithm for this purpose. The value somewhere lies between values 247 and 278. It is necessary to decide whether the value 247 will belong to the first part or the second one. The first possibility introduces the error 245-219=26 and the second possibility the error 247-245=2. Therefore, the cut between the values 247 and 278 is created and the summed-table area updated as *Figure 6.6b* shows. In the second step, we divide the first part in ratio 1:2 in x-coordinate (the longer one). The position of the value $1/3 \cdot 247 = 83.3$ in vector 41, 89, etc. (i.e., the last row in the area) is

found. It is between values 41 and 89, minimal error is achieved if the cut is constructed behind 89. The table is subdivided and values are updated – see *Figure 6.6c*. The algorithm continues until k groups are created. *Figure 6.6d* shows the result of the algorithm. As it can be seen, the workloads are 89, 75, 83, 91 and 71, i.e., relatively evenly distributed.

Unlike Whelan's approach, Mueller does not achieve perfectly balanced workload. However, it could be easily improved by successive finding of parts of grid where cuts lie until the required precision of balance is matched. As we did not noticed significant imbalance in our experiments with this points subdivision strategy, we do not use this modification. A great advantage of the algorithm is its complexity, it needs $O(N + k \cdot R)$ in the worst-case. Moreover, not all points need to be stored in the main memory because their coordinates are not used in the algorithm. If extremely large data set is considered, Mueller's approach is, therefore, an ideal choice.

1	3	6	9	11	12	12	13	13	13
4	12	20	27	32	35	37	40	42	42
8	23	37	49	59	66	71	76	81	83
14	37	60	79	96	109	118	126	133	136
20	48	77	100	120	135	146	155	164	169
26	59	92	118	140	156	168	178	189	195
33	72	110	138	161	178	190	200	212	219
41	89	134	164	188	205	218	228	240	247
47	103	157	190	216	235	249	259	271	278
53	114	174	212	242	264	280	291	303	310
59	125	189	231	267	293	312	325	338	345
66	138	207	253	291	319	339	353	367	375
72	150	225	276	318	349	371	385	400	409

a) the summed-area table

1	3	6	9	11	12	12	13	13	13
4	12	20	27	32	35	37	40	42	42
8	23	37	49	59	66	71	76	81	83
14	37	60	79	96	109	118	126	133	136
20	48	77	100	120	135	146	155	164	169
26	59	92	118	140	156	168	178	189	195
33	72	110	138	161	178	190	200	212	219
41	89	134	164	188	205	218	228	240	247
6	14	23	26	28	30	31	31	31	31
12	25	40	48	54	59	62	63	63	63
18	36	55	67	79	88	94	97	98	98
25	49	73	89	103	114	121	125	127	128
31	61	91	112	130	144	153	157	160	162

b) after the first division

1	3	3	6	8	9	9	10	10	10
4	12	8	15	20	23	25	28	30	30
8	23	14	26	36	43	48	53	58	60
14	37	23	42	59	72	81	89	96	99
20	48	29	52	72	87	98	107	116	121
26	59	33	59	81	97	109	119	130	136
33	72	38	66	89	106	118	128	140	147
41	89	45	75	99	116	129	139	151	158
6	14	23	26	28	30	31	31	31	31
12	25	40	48	54	59	62	63	63	63
18	36	55	67	79	88	94	97	98	98
25	49	73	89	103	114	121	125	127	128
31	61	91	112	130	144	153	157	160	162

c) after the second division

1	3	3	6	2	3	3	4	4	4
4	12	8	15	5	8	10	13	15	15
8	23	14	26	10	17	22	27	32	34
14	37	23	42	17	30	39	47	54	57
20	48	29	52	20	35	46	55	64	69
26	59	33	59	22	38	50	60	71	77
33	72	38	66	23	40	52	62	74	81
41	89	45	75	24	41	54	64	76	83
6	14	23	3	5	7	8	8	8	8
12	25	40	8	14	19	22	23	23	23
18	36	55	12	24	33	39	42	43	43
25	49	73	16	30	41	48	52	54	55
31	61	91	21	39	53	62	66	69	71

d) the result

Figure 6.6: *The partitioning of summed-area table in E^2 into 5 groups.*

In our parallel approaches (see Section 7), we use a simple random subdivision or median based subdivision, while in distributed approaches (see Section 8) Mueller's approach is preferred. Comparison of different strategies can be found in Section 9 and also in [Koh04b].

7 Parallelization for Symmetric Multiprocessors

In the previous section, we defined two parallelization purposes: to compute the Delaunay triangulation in as short time as possible and to compute the Delaunay triangulation of big data sets (theoretically unlimited). In this section, we propose several parallel algorithms that fulfill the first purpose.

We have chosen symmetric multiprocessors with several processors and shared memory as a parallel platform for these algorithms because of these three reasons. First, the time needed for synchronization is lower on the architectures with shared memory than on architectures with distributed memory. Next, the parallelization for this architecture is easier for the implementation, which allows us to develop algorithms that are simple to be implemented even by a person focused on computer graphics without a deep knowledge of parallel computations. Finally, symmetric multiprocessors belong to common equipment of laboratories; thus nothing special is needed.

We start several threads (usually one per each processor) and let them to insert points simultaneously into the triangulation stored in the shared memory. Indeed, we have to prevent a thread to perform a non-consistent modification somehow. In order to achieve as short construction time as possible, we decided to use the DAG structure for the location of simplices to be subdivided – see also discussion in the previous section.

7.1 Parallel Location Phase

In the location phase of the algorithm, the DAG structure is accessed read-only in such a manner to find the simplex containing the point to be inserted. As the algorithm modifies leaves only, all parent nodes in the DAG may be tested simultaneously without any troubles. To determine which child of the currently tested node can be accessed without the synchronization, we added a parameter into each node in the DAG structure. If the *i*-th bit in the value of this 3-bits (4-bits in E^3) long parameter is set to one then the *i*-th child is a leaf and has to be accessed exclusively. As any synchronization negatively influences the efficiency of the algorithm, the non-leaf children are tested first.

Figure 7.1 explains the meaning of the parameter in E^2. For an easier understanding, let us indicate by a letter L that the first bit is set to one (i.e., the first – left – child is a leaf), similarly M for the second bit and R for the third bit. Thus, for example in the "MR node" the thread can test the triangle in the left branch but testing triangles in the middle and right branches has to be synchronized.

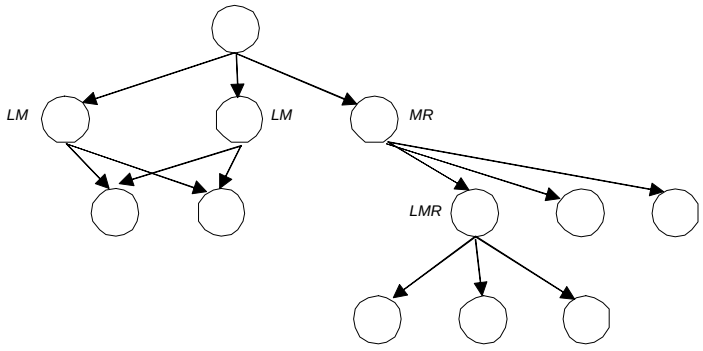


Figure 7.1: *The classification of the nodes in the DAG.*

7.2 Parallel Subdivision and Legalization Phases

In the subdivision and the legalization phases, a thread has to access one or more nodes and modify them. To avoid the collisions of the thread (in the meaning of simultaneous modification of the same node), we need to implement some synchronization mechanism. There are two possibilities of synchronization: the thread gets the exclusive access either to all leaves together or only to the currently requested node. We have identified three basic principles (they will be more discussed in the following text) according to these possibilities:

- Batch – several searching threads do the location and only one specialized thread handles the subdivision and the legalization phases.
- Pessimistic – all threads do the same work, however, the subdivision and the legalization can be done only in a critical section to ensure an exclusive access to the shared DAG.
- Optimistic – all threads do simultaneously all parts of the algorithm, if they want to modify a simplex, they need to get an exclusive access to it.

Let us note that batch and pessimistic principles are useless when some walking technique is used for the location instead of a hierarchical structure (such as the DAG).

7.3 Batch Principle and Batch Method

In the batch principle, there are several searching threads and one specialized thread. When a searching thread (producer) finishes its unsynchronized location, it puts an index of the currently inserted point into a queue in the shared memory. The specialized thread (consumer) gets the index from this queue, completes the location of the node to be subdivided, subdivides it and processes the legalization.

If the queue is empty, the specialized thread must wait, if it is full, the searching thread(s) must wait. The key issue is how long the queue should be to prevent the waiting of the threads. A long queue implies greater possibility that the unsynchronized part of the location stops for many points in the same node and the specialized thread will spend more time to complete the location. Even worse is to have a short queue because the queue would be full in a short time and the performance would decrease. According to our experiments [Koh04c], the queue length of 1024 seems to be optimal.

This method seems reasonable only for such a case when the time spent in the location is larger than 50%. Therefore, it makes no sense to consider it in E^3. According to profiling of the sequential algorithms (see Section 6), we can roughly estimate that the batch method in E^2 should achieve the highest speed-up when 3 searching threads are used at the architecture with four processors. Algorithms for searching thread(s) and the specialized thread are described in *Figure 7.2*.

7.4 Pessimistic Principle and Pessimistic Method

In this case, all threads do the same work. When a thread needs to access a leaf-node in the location phase, it enters the critical section, finishes the location on the leaf level, performs the subdivision and the legalization, and finally leaves the critical section.

The pessimistic method is simple but critical sections can be expected to limit its speedup. As usually two threads do not need to enter the critical section exactly at the same time, one thread performs the location and another one performs simultaneously the subdivision or the legalization for some time before the first thread (i.e., the thread that has just finished the location) has to start its waiting. It allows using this method even in the case when the

location phase does not consume more than 50%, i.e., pessimistic method is also available in E^3 but a substantial speed-up cannot be expected. The algorithm for inserting threads of pessimistic method is described in *Figure 7.3*. As it can be seen, this algorithm does not differ too much from the sequential algorithm given in *Figure 5.1*.

The searching thread

Input: A set $S_k = \{ p_0, p_1, ..., p_{m-1} \}$ of m points in E^2, $S_k \subset S$

```
for r := 0 to m - 1 do begin
   Locate the triangle T0 containing pr on the level of the parents
          of the leaves in the DAG structure;
   if the shared queue is full then wait;
   put {T0 , pr } into the queue
end;
```

The special thread

Input: Requests $\{T_i, p_j\}$ in the shared queue

```
while not all points inserted do begin
   if the shared queue is empty then wait;
   get { T0 , p0 }from the queue;
   Locate the triangle T1 ∈ DT(S) containing p0; //start at T0
   Subdivide T1;  // in the case where p0 lies on the shared edge, say
                  //between T1 and T2, then subdivide also T2.
   //Legalize all new triangles
   while there is an unchecked edge E do
      if the edge E violates DT criterion
               then perform local transformation
end;
```

Figure 7.2: *Parallel construction - the batch method*

The insertion thread

Input: A set $S_k = \{ p_0, p_1, ..., p_{m-1} \}$ of m points in E^2, $S_k \subset S$

```
for r := 0 to m - 1 do begin
   Locate the simplex S0 containing pr on the level of the parents
          of the leaves in the DAG structure;
   Enter a critical section;
   Locate the simplex S1 ∈ DT(S) containing pr; //start at S0
   Subdivide S1;  //in the case where pr lies on the shared edge or face
                  //then subdivide also the appropriate adjacent simplices

   //Legalize all new simplices
   while there exist an unchecked face (or edge) F do
      if the face F violates DT criterion
               then perform local transformation;
   Leave a critical section;
end;
```

Figure 7.3: *Parallel construction - the pessimistic method.*

7.5 Optimistic Principle and Optimistic Methods

Although in the worst case the insertion of the point causes a modification of the whole triangulation (i.e., all leaves are modified), changes are usually limited to several nodes of the DAG only. *Figure 7.4* shows an example of the locality of changes in the triangulation in E^2.

When a new point is inserted, only those triangles, whose circum-circles contain this point, have to be modified. In the figure, these triangles are marked by the dark gray color. No other triangles are modified.

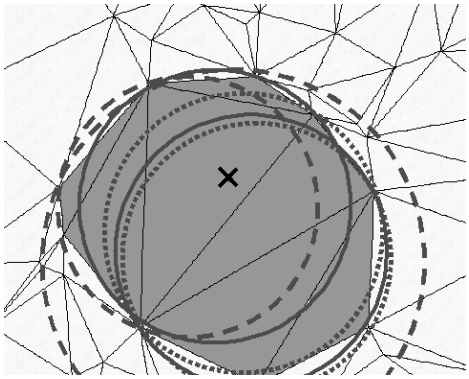


Figure 7.4: *The locality of changes in the triangulation in E^2. Inserted point is marked by a cross.*

As the order of insertion of the points is randomized, it is very probable that the set of the nodes accessed due to the insertion of the point P_i and the set of the nodes accessed due to the insertion of the point P_{i+1} are two disjunctive sets. Therefore, synchronization strategy that allocates all leaves to only one thread (like in the batch and pessimistic methods) is a kind of luxury. Let us consider a more efficient strategy. In the shared memory, sets M_t of accessed nodes ($t = 0 - k\text{-}1$, where k is number of used threads) are created. The sets are distributed among all threads. Each thread can access any set for read-only purpose and its own set for write purpose also. The work of the thread T_i can be described by the following algorithm:

1. Empty set M_i
2. Perform the location
3. Insert all nodes that are needed into the set M_i. Note: the detection of the nodes and their insertion to the set has to be done as an atomic operation.
4. If the intersection of M_i and M_j ($j = 0 - k\text{-}1$ and $j \neq i$) is not empty then wait until M_j is modified (i.e., until T_j completes the insertion of its point) and go back to 1. Note: the whole testing has to be done as an atomic operation.
5. Finish the location, perform the subdivision and the legalization and go back to 1.

The node should be accessed during the subdivision or the legalization if the currently inserted point lies inside the circum-sphere of node's simplex. It may not be an easy task to determine all such nodes at the beginning of the subdivision. Moreover, in the legalization phase it is necessary to apply the same test once again to perform flips, which would reduce the performance of the proposed parallel solution. Therefore, the set M_i is constructed successively, i.e., whenever a node (simplex) S_n is needed by the thread T_i, the thread makes the following actions:

1. If the node S_n already exists in M_i, go to 4.
2. If the node S_n exists in any concurrent set M_j, wait until M_j does not contain this node.
3. Add the node S_n into M_i.
4. Access the node S_n.

This solution introduces a problem of the deadlock caused by a mutual waiting of the threads. We have designed several methods based on the just described optimistic principle: optimistic method, burglary method and circum-circle method. They differ in the representation of the set M_i and in the strategy of the deadlock handling.

7.5.1 Optimistic Method

We added a "lock" parameter into each node. The zero value of the parameter means this node is not allocated to (locked by) any thread, non-zero value $i+1$ means it is allocated to (locked by) the thread T_i. The thread can access only nodes already locked by it. The nodes are locked in the locking routine except for the nodes newly created in the subdivision or the legalization – these nodes are automatically allocated to the creating thread T_i by the simple setting of the "lock" parameter to the value $i+1$. In the locking routine, naturally, the thread can lock the node only if it is not locked by another thread.

Let us note that the lock testing and lock setting in the routine has to be done together as an atomic operation. It can be done in a critical section, however, entering and leaving a critical section slow down the computation if we use standard techniques for the synchronization provided by the operating system, such as semaphores. Fortunately, it is not necessary to use them because low-level atomic instructions are usually available, e.g., Intel x86 provides atomic instructions such as XADD, CMPXCHG, lock INC or lock DEC which are satisfactory for our purpose – for details see [Koh04b].

If the thread cannot lock the node, it has to perform one of the following two strategies:

- *Detection strategy* – detect whether the thread should wait until the required node is 'free' or give up the insertion for the moment, undo all changes and return to the location part because the mutual waiting of the threads would cause the deadlock. This detection requires a short critical section and it could limit the performance of the algorithm.
- *Priorities strategy* – compare whether thread priority is larger than the priority of blocking thread. If the result is positive, it starts to wait, otherwise it gives up the insertion for the moment, undoes all changes, returns back to the location part and sets its priority to 1 + maximum of the current priorities of all threads. In such a strategy, the thread could often undo the changes unnecessarily.

According to our experiments (not presented here), we found that both strategies are more or less the same. Therefore, the priorities strategy is no longer considered in the next text. The simplified version of the algorithm for inserting threads is described in *Figure 7.5*.

We decided to use pseudo-active waiting: the thread repeatedly yields its short time interval until it can continue. The spared time can be used by another thread. If a thread waits only for a short time (as usually in our case), this solution leads to greater performance than the use of standard resources for synchronization supported by the operating systems, such as semaphores, etc.

Whenever the node is newly locked, the pointer to this node is added to a local array accessible just to the locking thread. According to our experiments, the fixed length of 8192 items in this array is sufficient in E^2. It is impossible to find any fixed length in E^3. Therefore, the algorithm enlarges the array dynamically during the computation whenever it finds that more than 60% of the array is full.

The thread unlocks all nodes when it completes the insertion of its current point. However, it is usually not necessary to keep the node locked for the whole insertion of the point. Therefore, we also tried to unlock the node as soon as it is detected that the node will not be

accessed again in this insertion of the point. Our experiments, however, show that such strategy is useless because it consumes more time than the original one.

The subdivision and the legalization have to be considered as an atomic operation to ensure the quality of the resulting triangulation. Therefore, the transaction mechanism has to be incorporated in this method. When a thread modifies the node, it logs the changes into a journal. If the thread has to give up the insertion and return to the location phase, it reads this transactional journal backward and undoes the changes. The journal is implemented as a static stack data structure capable to hold 4096 items in E^2 because, according to our experiments, no more than 1000 transactions are performed for any tested data set. In E^3, the number of transactions is sometimes quite big – we have found up to 200 thousands of transactions. As it is unnecessarily to use such very large stack and this value is not ensured to be a maximum, we use a dynamic solution similar to that one used for storing locked nodes.

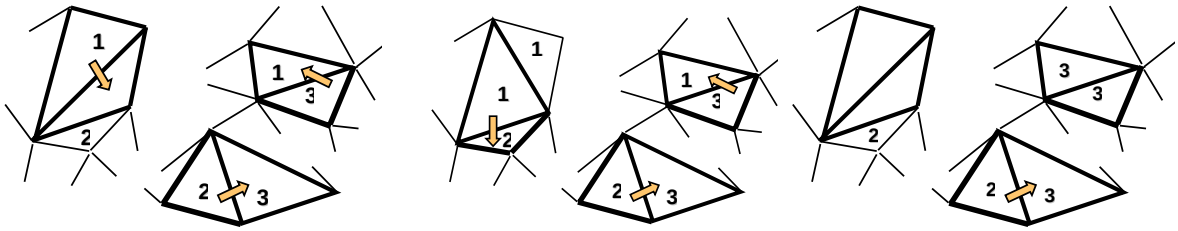
The insertion thread

Input: A set $S_k = \{ p_0, p_1, ..., p_{m-1} \}$ of m points in E^2, $S_k \subset S$

```
for r := 0 to m - 1 do begin
1: Locate the simplex S0 containing pr on the level of the parents
          of the leaves in the DAG structure;
   Lock S0;       //always succeeds
   if S0 is not on the level of the parents of the leaves then begin
      Unlock S0;
      goto 1;
   end;

   Locate the simplex S1 ∈ DT(S) containing pr; //start at S0
   Lock S1 and all its neighbors;
      //in the case where pr lies on the shared edge or face then
      //lock also the appropriate adjacent simplices and their neighbors
   if a deadlock has been detected then begin
      Unlock all simplices;
      goto 1;
   end;

   Subdivide S1;  //in the case where pr lies on the shared edge or face
                  //then subdivide also the appropriate adjacent simplices

   //Legalize all new simplices
   while there exist an unchecked face (or edge) F do
      if the face F violates DT criterion then begin
            Lock all simplices sharing the face F and all their neighbors;
            if a deadlock has been detected then begin
2:                Undo all changes;
                  Unlock all simplices;
                  goto 1;
            end;
            perform local transformation;
      end;

3: Confirm all changes; //i.e., empty transaction journal
   Unlock all simplices;
end;
```

Figure 7.5: *Parallel construction - the optimistic method (simplified version).*

Figure 7.6 shows a simplified version of the deadlock (it is caused by the mutual waiting of threads) handling in E^2. Number in a triangle identifies the thread that locked the triangle. An arrow crossing an edge means that the thread that locked the triangle where the arrow begins needs to access the triangle where the arrow ends. In *Figure 7.6a*, the thread T_3 needs to access a triangle that is currently locked by the thread T_1, therefore, it has to wait. As the thread T_2 needs a triangle locked by T_3, it has to wait as well. The last thread T_1, currently operating with the upper left triangle, gets an exclusive access to the adjacent triangle and it flips the common edge to fulfill the circum-sphere criterion. This change could violate the criterion elsewhere and, therefore, the thread needs to access another triangle. This triangle is locked, however, by the thread T_2 – *Figure 7.6b*. As the waiting would lead to deadlock, the thread T_1 gives up the insertion and unlocks all triangles. Now, the thread T_3 can continue – see *Figure 7.6c*. Meanwhile, the thread T_1 tries to insert its point once again.



a) change *b) deadlock* *c) undo*

Figure 7.6: *The deadlock handling in the optimistic method – E^2 case.*

7.5.2 Burglary Method

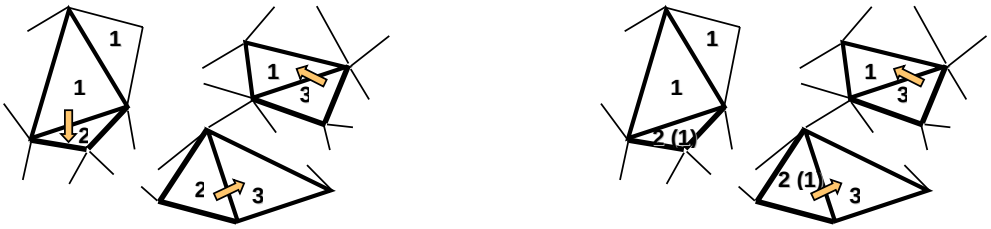
There is no doubt that the transaction mechanism (needed for deadlock handling) negatively influences the performance of the parallel solution. Our experiments [Koh04b], however, show that the probability of the occurrence of deadlocks is almost negligible and it decreases with the growing number of input points. This means that the transactions are in most cases a kind of luxury and, therefore, some strategy, which avoids the transactions, is welcome.

Burglary method is a modification of the optimistic method. It does not use the transaction mechanism, the deadlocks are handled in a different way: when a thread detects the deadlock (i.e., it cannot access the requested node and it cannot start to wait because its waiting would cause the deadlock), it grants itself a right to access all nodes locked by its counterpart and continues. As its counterpart waits and will wait at least until the offending thread unlocks its nodes, the concurrency is avoided. The algorithm for inserting threads of the burglary method differs from the algorithm given in *Figure 7.5* only in that there is no code on lines 2 and 3.

Figure 7.7 brings an example of deadlock handling in E^2. Number in a triangle identifies the thread that locked the triangle. An arrow crossing an edge means that the thread that locked the triangle where the arrow begins needs to access the triangle where the arrow ends. In *Figure 7.7a*, the thread T_3 waits for the thread T_1, the thread T_2 waits for the thread T_3 and the thread T_1 needs to access a triangle locked by the thread T_2 – i.e., a deadlock occurs. The thread T_1, which detected the deadlock, gains access to any triangle locked by the thread T_2 and continues – see *Figure 7.7b*.

When the owner of the attacked nodes finally finishes its waiting, it has to check whether its original request is still valid because, although it is not very probable, the offending thread could have modified the currently requested node(s). In such a case, the thread cancels the

request and continues its work. When a request cannot be processed, the risk of the occurrence of some non-Delaunay simplices in the resulting triangulation is increased. If the cancelled requests were logged, it would be possible to correct the triangulation sequentially in the post-processing. However, as explained before, burglary method does not use this methodology because our experiments show that the probability of incorrect triangulation is almost zero and when it happens, the number of wrong simplices is very low (in E^2 no more than 5 in 2 000 000 triangles).



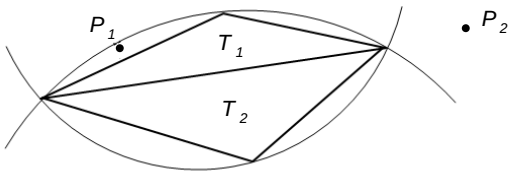
a) deadlock *b) break-in*

Figure 7.7: *The deadlock handling in the burglary method – E^2 case.*

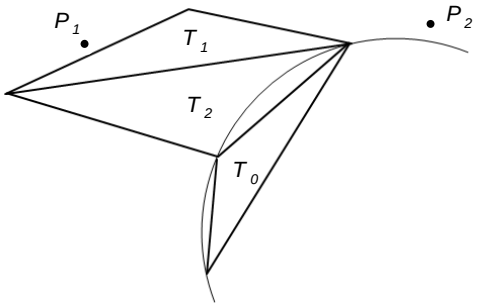
7.5.3 Circum-Circle Method

The circum-circle method is a modification of the burglary method. The decision whether to access a node or wait is solved, unlike the previous optimistic methods, by a geometric test. It can be proved that the subdivision and the legalization influence only such simplices where the input point lies in their circum-spheres – let us remind Bowyer-Watson algorithm and *Figure 7.4*. Therefore, any thread that is currently inserting such a point that lies inside the circum-sphere of some simplex will access the node of this simplex 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 simplex of this node. If the result of this test is negative, the thread has to wait, otherwise it continues. Let us note that the deadlock handling is identical to that one used in the burglary method. *Figure 7.8* shows an example of the use of just described geometric test.

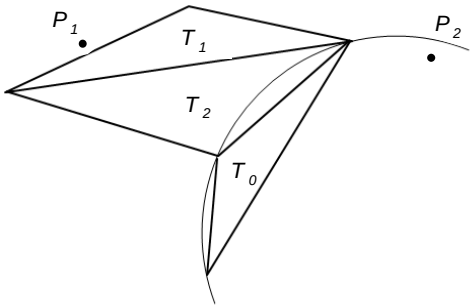
There are two serious problems with the circum-circle approach. Both negatively influence the efficiency of this method and limit its use to few processors only and, according to our experiments, to E^2 only. The first problem is that sometimes a thread waits for a long time because we are unable to determine when exactly the second thread will reach the node. Especially a large circum-sphere can block a thread too early, i.e., the thread has to wait although its concurrent thread currently works with distant simplices. The second problem is related to the circum-sphere test evaluation. This test has to be done for each accessed node. However, the complexity of this test depends linearly on the number of tested points, i.e., on the number of used threads.



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



c) P_2 lies inside the circum-circle C_0 of T_0 - synchronization is necessary

Figure 7.8: *Example of the legalization with the circum-circle method.*

In this section, we have proposed several parallel methods for the construction of the Delaunay triangulation in both E^2 and E^3 suitable for symmetric multiprocessors and we have discussed expected behavior of these methods. All proposed methods were implemented and tested. The results of our experiments are summarized in Section 9.

8 Parallelization for Clusters of Workstations

In the previous section, we focused on the development of parallel algorithms suitable for fast construction of the Delaunay triangulation. However, there exist also applications that need to deal with data sets that are impossible to be processed on symmetric multiprocessors because of the limited amount of shared memory on these architectures. A good example of such an application is the surface reconstruction based on the CRUST algorithm (see Section 10).

Cluster of workstations is a collection of independent computers, interconnected via a network, that are capable of collaborating on a task. As the number of computers in the collection can be theoretically unlimited, cluster offers theoretically unlimited computational power and storage. Therefore, it is an ideal parallel architecture for the construction of the Delaunay triangulation of big data sets, which is our goal.

We let several processors to insert points simultaneously into the triangulation distributed over local memories. There are three important issues: loading of input data, storing of output triangulation and communication during the process (including sending and receiving data) that need to be discussed.

8.1 Loading and Storing

In the case of symmetric multiprocessors, input data set is loaded into the shared memory by one processor and then all processors are able to access the entire set, i.e., to operate with all input points. However, clusters do not share memory or program execution space. For cluster of workstations, there are two possibilities. If the input data set is stored on a shared device, processors can load the entire input set into their local memories. Let us note that the shared device could be a network drive, FTP or WWW server, etc. If the input data set is stored locally and accessible only to one processor, this processor is responsible for sending the required data to all its counterparts. Both approaches have pros and cons.

The loading of input data set from the shared device is very simple; there is even no difference between loading from a local storage and a network drive. On the other hand, as the loading time grows with the number of processors, the performance of parallel algorithm drops down. Therefore, in practice, we are limited to small number of processors only. In order to overcome this drawback, the input data set can be mirrored on several shared devices and some processors then will load data from one device while other processor from another one. In the extreme, each processor has its own copy of the input data set. There are two problems. First, mirroring requires additional user effort. Next, big data, even if it is compressed, might need hundreds of megabytes on the storage medium. That is why the proposed solution is not always welcome or even possible.

In the case that data is accessible to just one processor, the problem is more complex. The straightforward solution is to let this processor, so called sender, to load the data into the local memory and then successively send it in one part to every processor (receiver). A receiving processor waits inactively until the data is available and then it immediately starts the construction of the Delaunay triangulation. It is clear that the delay between times when the first and the last contacted processor is able to start grows with the number of processors. This negatively influences the workload balance and, therefore, the performance of the parallel algorithm drops down. In practice, this approach is useful only for small clusters.

A possible improvement is to exploit multicasting, i.e., the data is sent just once but received almost simultaneously by all receivers. Multicasting, however, usually is not reliable and a message may never be received by one or more processors because of possible failures in the

network. Let us note that there is also a reliable version of multicasting [Liu04], however, it introduces a lot of overhead.

Another possibility is to split the input data set into several parts and transmit them to all processors successively. When processors receive the first part, they can start their work. This solution is quite efficient, especially if it is combined with multicasting, however, as it requires complex synchronization between processors, it is not simple to be implemented.

Let us discuss another thing related to the loading of input data set. There is no doubt that the performance of parallel algorithm increases with the decreasing amount of data to be sent via network. In the case of loading from a shared device, we usually have to transfer the whole data set. However, when the data has to be sent, we can simply employ some fast existing compress utility in order to reduce the transmitted amount. Unfortunately, losless compression is often not very helpful. As many applications do not require exact precision, it is possible to use some lossy compression algorithm.

Until now, we have assumed that all processors obtain the entire input data set. However, this kind of luxury is not always possible because data sets can be very large, e.g., Michelangelo's statue of Barbuto requires at least 4 GB (see Section 10). Better strategy is to let a processor store only points that it needs. Among these points, of course, points to be inserted by this processor belong. The remaining points usually lie near the boundary of the local part of triangulation. The simple way is, therefore, to store all the points to be inserted by adjacent processors. By the term adjacent processors we mean that they handle insertion of points lying in adjacent regions. Another way is to store remaining points on the demand, which means that when a processor needs to access coordinates of a point not stored in its local memory, it contacts the appropriate processor and downloads the requested information. As this approach consumes additional time (because of the communication), the performance of algorithm is reduced. On the other hand, it significantly spares memory occupation. Let us note that since input data set has been distributed, it is necessary to implement some mechanism available to map the index of a point into the pointer to local memory where the coordinates of this point are stored. Two mechanisms will be described later in this section.

Let us briefly discuss several possible scenarios that might occur after processors have finished the construction of the Delaunay triangulation. First, processors store the result directly on a shared device into one or more files (one file requires some synchronization) or, if there is no such device, they send the result to one processor responsible for storing. In the second case, a technique for the compression of meshes can be used in order to reduce the amount of transferred data. There are plenty of methods suitable for the compression of triangular meshes and some of them can be extended to E^3 as well. A comparison of these methods Grabner [Gra03] brings.

For some applications, e.g., for FEM or surface reconstruction, the construction of the Delaunay triangulation is only the first step and they continue with the processing of the resulting triangulation in such a manner to achieve the desired purpose. As the further process is often parallelized, it is pointless to store the Delaunay triangulation on a centralized place.

8.2 Communication between Processors

During all phases of parallel construction, it is necessary to exchange some data between processors, i.e., to communicate. In a general scheme, the processor sends a message representing a request. The message is delivered to a receiver, which processes the request and sends a message in response. In turn, the reply may trigger a further request, which leads to a subsequent reply, and so forth.

No matter whether the sender waits inactively for the response on its request or it works meanwhile on something else, any communication consumes time, which negatively influences the performance of parallel algorithm. In order to develop an efficient parallel algorithm, it is necessary to minimize the amount of required communication and synchronization (synchronization between computers implies communication).

There are several basic communication models [Liu04]. Each of them provides different level of abstraction. At the lowest level, there is the socket API, a programming interface based on pure message passing. A processor wishing to communicate with another processor must create an instance of a programming construct termed a *socket*. This socket represents a relation between these two processors, a sender and a receiver. Using a socket, processors may exchange data as follows: the sender writes a message into the socket and at the other end, the receiver reads the message from the socket. For these purposes, the socket API provides a set of basic operations such as *send*, blocking and non-blocking *receive*, etc. As the socket API is a mechanism of a low level of abstraction, the development of a complex application using this API is difficult and takes a lot of time. On the other hand, because of its low latency, the socket API may be the most appropriate for an application that calls for a fast response time or for a system with minimum resources. Let us note that the socket API is the base networking stuff available on all operating systems (including MS Windows or Unix) and its use does not need an additional installation.

On higher level of abstraction, there are two famous systems for distributed computing, Parallel Virtual Machine (PVM) and Message Passing Interface (MPI). Similarly to the socket API, both systems provide a set of operations for message transmission. These operations handle message routing and perform data conversion for incompatible architectures (e.g., conversion between little and big endian) and other tasks that are necessary for operation in a heterogeneous network environment. PVM and MPI provide routines for synchronization between processes via barriers. MPI, moreover, offers a set of advanced operations suitable for scattering data (typically vectors) over processors or their gathering, which is very useful for many applications (especially in mathematical sciences). MPI and PVM support an automatic execution of user program on each computer registered for the distributed computing, i.e., users are not required to launch applications manually like in the socket API model. One little drawback with PVM and MPI is that only programs written in C or Fortran are supported.

The previously described models work well, however, if a distributed application needs to be developed more rapidly or be developed by a person without a deep knowledge of distributed computation, models that are more abstract are desired. The Remote Procedure Call (RPC) or object oriented the Remote Method Invocation (RMI) allow programmers to build network applications using a programming construct similar to a local procedure call. When a processor, the client, needs to make a request to another one, the server, it issues a remote procedure call to the server. Similarly, as in the case of local procedure calls, this call triggers a predefined action in the procedure provided by the server. At the completion of the procedure, the server returns a result of the call to the caller (client).

Let us describe the communication between processors in more detail. The client application calls a proxy[1] procedure. This routine retrieves required parameters from the client address space and translates them as needed into some standard format for transmission over the network. Afterwards, it calls functions in the RPC library to send the request and its

[1] Proxy is an interface-specific object that packages parameters for that interface in preparation for a remote method call. A proxy runs in the address space of the sender and communicates with a corresponding stub .

parameters to the server. The RPC library at the server accepts the request and calls the server stub[2] procedure. The stub retrieves the parameters from the network buffer and converts them from the network transmission format to the format the server needs. The server stub then calls the actual remote procedure on the server. When the remote procedure completes, a similar sequence of steps returns the output data to the client. The client proxy converts data into requested format and writes it into the client memory. Afterwards, the proxy returns the result to the calling program on the client. The calling procedure continues as if the procedure had been called on the same computer.

As all proxies and stubs are created automatically by the compiler, the development of a distributed application using for the communication the model we have just described is straightforward. Moreover, the RPC middleware allows to programmers to implement their client and server applications using different programming languages (e.g., C#, C++, Java, etc.) and run them on different platforms (little vs. big endian, 32-bit vs. 64-bit, Unix vs. Microsoft) without the need to add any line of code. On the other hand, this high level of abstraction introduces big overheads and, therefore, if response time and resource consumption are a concern, it may be more appropriate to use the socket API.

Let us note that, usually, the RPC stuff is not used directly but programmers exploit CORBA or DCOM technologies [Liu04]. In our opinion, both architectures provide more or less the same functionality. Unlike the CORBA technology, the DCOM works on Microsoft platforms only. As the DCOM is a part of Microsoft Windows, it can be used immediately without any need to install additional tool. Due to the required registration of distributed programs, the DCOM is able to launch automatically the application containing the server code. Strong security incorporated to the DCOM protects remote procedures to be called by not authorized client application possibly written by a network attacker. On the other hand, to configure distributed applications to work properly is quite difficult (unlike the CORBA), especially for a person who is not familiar with networking.

8.3 Distributed Computing

As there is no shared memory on clusters, only part of the triangulation is available on one computer. In order to get the Delaunay triangulation, the consistency on boundaries of local parts must be ensured. It can be done either in the post-processing according to pure Divide & Conquer strategy or in the pre-processing similarly to the Hardwick's algorithm (see Section 4) or successively during the whole process of insertion.

The first solution (i.e., the post-processing) suffers from several serious drawbacks. First, the merging of two local parts is limited to just one processor and usually it is required that both parts are stored in the local memory of this processor, which means that we need to transmit a lot of data. In the worst-case, the whole Delaunay triangulation is present on one computer. Next, the implementation is not simple, especially if the stress is put on the efficiency or we want to avoid the necessity to have all data in one local memory.

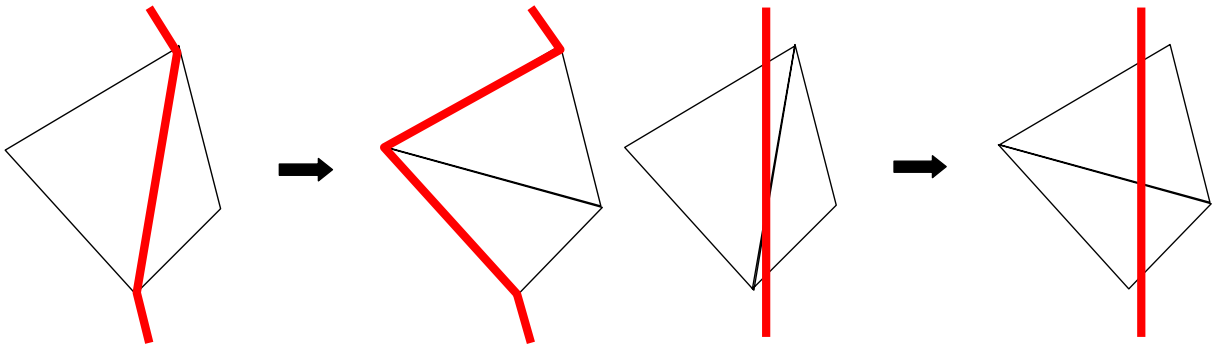
A parallel algorithm that constructs Delaunay edges or faces on boundaries in the pre-processing is supposed to be very efficient because after that each processor can triangulate its group of points without any need to communicate. On the other hand, we need to know all points in advance, thus this approach is useless for applications that construct the Delaunay triangulation from points arriving online (possibly, samples of some measurement).

[2] The stub is an interface-specific object that unpackages the sent parameters and calls the required user method. The stub runs in the receiver's address space and communicates with a corresponding proxy in the sender's address space.

Moreover, to construct the joint, it is very often required to have an access to all points, which limit the use of this approach to smaller data sets. Let us note that the incremental insertion algorithm used for local triangulation demands the incorporation of constraints, which makes it slightly more difficult to be implemented. We propose an algorithm based on this strategy in the section discussing possible extensions – see Section 11.

Finally, last option is to keep the Delaunay triangulation consistent in any time of the insertion of points. It is clear that we cannot avoid communication, which negatively influences the efficiency. On the other hand, points may arrive online and the implementation of an algorithm exploiting this approach seems to be quite simple. Boundaries between local parts of the triangulation can be either dynamic or static. In the case of dynamic boundaries, boundaries are formed by edges of triangles (or faces of tetrahedral in E^3). When a triangle having at least one edge as a part of boundary has to be subdivided or any of its edges has to be swapped, the boundary has to be updated – see *Figure 8.1a*. If workloads should retain balanced, a good heuristics is required. Let us note that there is a necessity to construct sequentially the initial triangulation in order to get initial boundaries.

Dynamic boundaries are used also by Chrisochoides et al. in their approach (see Section 4) and, therefore, we can adopt their solution for our purpose. However, there is another problem not solved by Chrisochoides. As boundaries dynamically change, some of points originally assigned to one processor might be no longer in its region and, therefore, we need to resent them to the appropriate processor. Indeed, it consumes some additional time.



a) dynamic boundaries *b) static boundaries*

Figure 8.1: *The swap operation on a boundary between two local triangulations and a possible result of this operation on the boundary shape.*

Static boundaries are simple and natural for static distribution of points over processors (see Section 6). Boundaries remain unchanged for the whole process and, therefore, no point has to be resent – see *Figure 8.1b*. Initial triangulation is also not required. In comparison to dynamic boundaries, an algorithm using static boundaries is quite simple to be implemented. On the other hand, as boundaries cross over some simplices, the access to these shared simplices has to be synchronized, which reduces the overall performance of parallel algorithm. There is about $\sqrt{N}$ shared simplices in the triangulation in E^2 and about $\sqrt[3]{N^2}$ in E^3, where N is the number of points. *Figure 8.2* acknowledges this for a uniform data set.

a) E^2 *b) E^3*

Figure 8.2: *The cut of an object containing N uniformly distributed points has approximately $\sqrt{N}$ in E^2 and $^3\sqrt{N} \cdot ^3\sqrt{N}$ in E^3 points in its vicinity.*

Due to these properties, we have decided to use static boundaries in our algorithms. The question is how to deal with operations (subdivisions or local transformations) that require an access to shared simplices. Three basic principles can be identified (they will be more discussed in the following text):

- Operation flow – the operation with its parameters (e.g., coordinates of point to be inserted) is sent to all processors sharing the required simplices and these processors perform the operation using their local copy of shared data. New shared simplices are created in local memories of contacted processors as the result of the operation.
- Data flow – required shared simplices are retrieved from their remote places of storage using the optimistic principle (see Section 7) for the synchronization, the operation is performed and modified simplices are sent to be stored back in their remote storage.
- Mixed flow – the operation with its parameters is send to the appropriate processor, this processor performs the operation and all local consequential operations and sends the resulting data to the initiator.

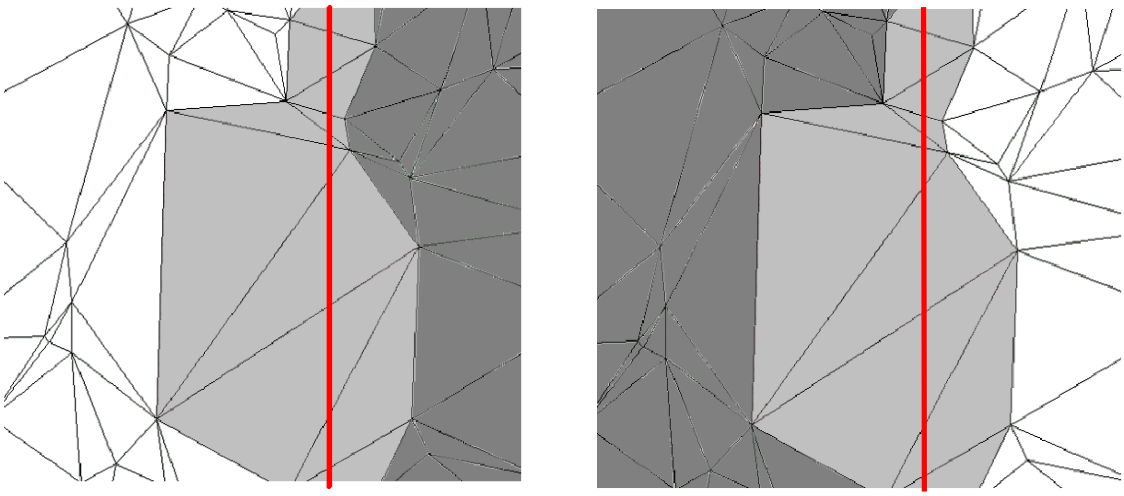
8.4 Operation flow

This approach is based on farmer-worker strategy (see Section 2). There are several workers responsible for the insertion of points into the triangulation stored in their local memories and one specialized processor (farmer), called *interface*, responsible for the loading of input data sets, the subdivision of points and the storing of constructed triangulation. The interface also provides some mechanism for the synchronization of workers.

From the point of view of any worker, say W_1, there are three kinds of simplices in the Delaunay triangulation to be considered during the insertion – see *Figure 8.3*:

- A local simplex lies fully inside the region assigned to the given worker W_1 and it is stored in the local memory of this processor. No synchronization is required to access local simplices. Let us note that local simplices are the most numerous group.
- A shared simplex overlaps at least two regions (one of them is processed by the worker W_1). It is stored redundantly in local memories of all processors responsible for these regions and also in the local memory of the interface. Simultaneous modification of shared simplices has to be avoided and, therefore, some synchronization is necessary (i.e., the interface has to be contacted).

- A remote simplex is stored in the local memory of another processor. However, when the processor W_1 modifies a shared simplex, it has to know the identification of the remote simplex adjacent to the shared one in order to update the neighborhood correctly.



a) worker W_1 *b) worker W_2*

Figure 8.3: *An example of the Delaunay triangulation in E^2 distributed over two processors (boundary is denoted by thick vertical line) – local triangles are white, shared triangles are denoted by light gray and remote triangles by dark gray.*

Each worker runs two threads. While the main thread successively inserts its points into the triangulation, the second thread handles requests sent by remote processors (i.e., other workers or the interface). Indeed, the work of both threads must be synchronized. When a worker needs to perform operation, i.e. the subdivision or the local transformation, such that it requires the modification of the content of a shared simplex, the worker contacts the interface to perform this operation in a synchronized way. Actually, it means that the worker enters a distributed critical section. The interface enters a local critical section, finds the appropriate shared simplex and performs the operation. For each new simplex constructed during the operation, the interface then determines which processors share this simplex. After that, the interface contacts all workers (including the originator of this synchronized operation) that share any simplex modified during the operation sending them information about the operation. These workers receive the message sent by the interface in the second thread, perform the operation and acknowledge the change to the interface. From the description of the operation flow we gave so far, it is clear that shared simplices are processed and stored redundantly. Let us note that as the run of the main and the second thread must be synchronized (we have to avoid simultaneous modification of the same simplex), a critical section is used for this purpose.

When all contacted processors performed the operation, the interface proceeds with the consequential operations, i.e., with the legalization of shared simplices. Indeed, it introduces another communication because any operation with a shared simplex must be processed by every worker sharing this simplex. When these operations are finished, the interface leaves a critical section and sends an acknowledgment to the worker that initiated the processing of the synchronized operation. After receiving this message, the initiator continues with further legalization and then it proceeds with the insertion of next point. The simplified version of the operation flow algorithm is described in *Figure 8.4*.

Worker - the insertion (main) thread

Input: A set $S_k = \{ p_0, p_1, ..., p_{m-1} \}$ of m points in E^2, $S_k \subset S$

```
for r := 0 to m - 1 do begin
   Locate the simplex S0 containing pr in the DAG structure;
   if S0 or any of its neighbors is shared then begin
      Send the operation (subdivision) to the interface;
      Wait for the reply;          //i.e., until the operation is finished
      Enter a critical section;
   end else begin
      Enter a critical section;
      Subdivide S0;
   end;

   //Legalize all new simplices
   while there is an unchecked face (or edge) F  do begin
            Leave a critical section;
            if the face F violates DT criterion then begin
                  if any of participating simplices is shared then begin
                        Send the operation (swap) to the interface;
                        Wait for the reply;
                  end else begin
                        Enter a critical section;
                        Perform local transformation;
                        Leave a critical section;
                  end;
            end;
            Enter a critical section;
      end;
   end;
   Leave a critical section;
end;
```

Worker - the receiving (second) thread

```
while not all points inserted do begin
   Receive the operation from the interface;
   Perform the subdivision or the local transformation;

   //Legalization
   for each outer face (or edge) F of new created simplices do
      mark F as unchecked;
   Send reply to the interface;
end;
```

Interface - main thread

```
while not all points inserted do begin
   Receive the operation from the worker W1;
   Enter a critical section;
   Perform the subdivision or the local transformation;
   Send operation to every worker sharing any of modified simplices;
   Wait for replies;    //from all workers
   Leave a critical section;
   Send reply to the worker W1;
end;
```

Figure 8.4: *Parallel construction - the operation flow (simplified version).*

Let us now describe the operation flow in detail. The first problem to be discussed is how to uniquely identify simplices in the triangulation and exploit this identification to retain the connectivity between simplices. As the triangulation is distributed, we cannot use 32-bits pointers to identify simplices like in the parallel solution. Let us, therefore, assign an integer value to every simplex. Positive values are used for local simplices, negative values for shared. By checking the highest bit of the simplex identification, processor easily determines whether the simplex is local or shared and according to the outcome of this test, it decides how the operation (e.g., subdivision) should be handled. While an assignment of value to a shared simplex is done in a synchronized way (i.e., via the interface), a local simplex is evaluated by its proper worker independently, i.e. without any communication. This, however, means that it is necessary to add some information to identify local simplices in the triangulation. Such information is, naturally, the processor identifier.

We could either use these pairs of identifiers (i.e., processor ID and simplex ID) anywhere in the algorithm or proceed with traditional pointers and use simplices identifiers only in cases when pointers cannot be exploited. As the use of pointers is much faster and pointers are suitable for local simplices, which are the most numerous group in the triangulation, the second option is preferable. Therefore, identifiers are used only by the interface, which allows to it to retain the connectivity of the triangulation.

Figure 8.5 shows a part of triangulation in E^2 and its appropriate data structure. As remote nodes (and triangles) are not physically stored in the local memory, shared simplices on workers cannot refer to these nodes using pointers and, therefore, an invalid (not null) value is used. In figure, this value is denoted by letter *R*. The invalid value informs workers that they need to contact the interface in order to retain the connectivity between the given shared simplex and the remote one or to perform an operation with these simplices (e.g., the swap).

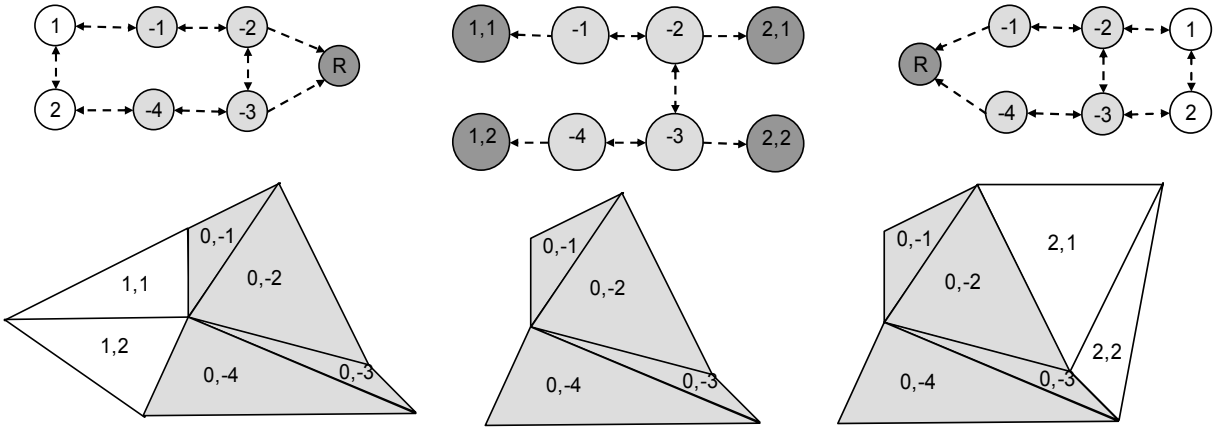


Figure 8.5: *An example of the Delaunay triangulation in E^2 and its appropriate data structure (upper images). Local triangles are white and shared triangles are light gray. Remote triangles are denoted by dark gray. Dashed arrows between nodes show the connectivity. While pointers are used on workers, the interface uses triangles identification – a pair of integers in triangles (and nodes).*

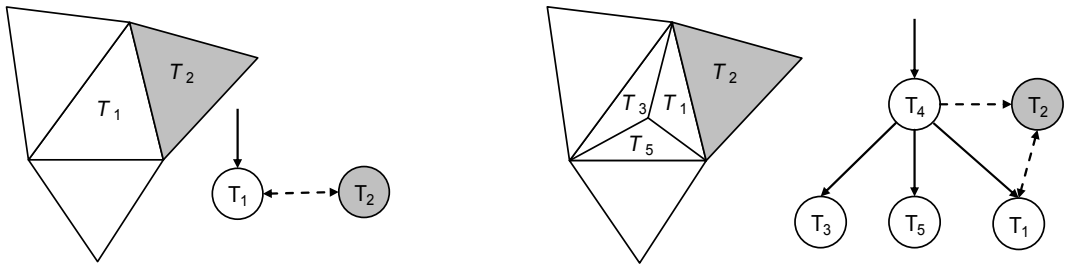
Now, it is time to describe the operation flow algorithm step by step. At the beginning of computation, it distributes the input points over the working processors (see Section 6). It is clear that a processor needs to know coordinates of its points and also coordinates of points lying outside of its area but belonging to vertices of simplices shared by this processor.

Unfortunately, it is impossible to predict which remote points (i.e., points processed by another processor) will be used. We can, therefore, either store the whole input set in the memory or send coordinates of remote points successively during the computation and store them into some additional data structure, which is of course slower but allows us to deal with larger data sets. Let us note that a hash table suits for this purpose well. We can use the highest bit in the index of point to determine whether the given point is remote, i.e., coordinates are stored in the hash table, or local one, i.e., its coordinates are elsewhere. The use of the highest bit allows us also to apply an efficient test on a simplex to find out whether this simplex is shared (at least one of its vertices is a remote point) or local.

After the distribution of points, the interface constructs the first auxiliary simplex, sends it to all processors and they start with insertion of their points. The location of the simplex to be subdivided is identical to the location in the sequential algorithm, i.e., no synchronization is needed (unlike the parallelization for symmetric multiprocessor), and no matter whether some hierarchical structure to speed-up the location is used, e.g., the DAG, or some walking technique is employed. This is because there is only one thread (on one processor) operating with the local part of triangulation at one time.

Now, we describe how different operations are handled. For easier understanding, we focus on E^2 case. Let us assume that the processor W_1 wants to insert a point into the triangulation and it already somehow managed to find the triangle to be subdivided. If the triangle and all neighbors of this triangle are local, the subdivision operation is processed without any synchronization because it does not influence the workspace of another processor. If the triangle is local but there is just one shared neighbor, we can handle this case also without any communication by reusing of the identification of the triangle to be subdivided for the new triangle adjacent to the shared one as follows – see *Figure 8.6*. After the triangle T_1 is subdivided into three triangles T_3, T_4 and T_5, the connectivity between these new triangles and the triangles adjacent to the triangle T_1 has to be updated. The triangle T_2 is shared and, therefore, the processor has to contact the interface to change the reference on T_1 stored in the node of T_2 to reference on T_4. If we, however, swap identifiers of triangles T_1 and T_4, no communication is required because, although the triangle T_1 is different, the connectivity has not been modified.

In the case that there are at least two shared neighbors, the communication cannot be avoided and the processor has to contact the interface and send it identifications of the shared triangle, previous local triangle and new local triangle. The interface finds the data structure for the shared triangle using a lookup table (in an ideal case, a perfect hash is exploited) and updates the information.



a) before the subdivision *b) after the subdivision*

Figure 8.6: *The operation of subdivision of local triangle with just one shared neighbor (denoted by light gray) and appropriate changes in the DAG structure. Dashed arrows show the connectivity between triangles, solid arrows history of changes (i.e., the parent triangle and its children).*

If the triangle to be subdivided is the shared one, the situation is more complex because the subdivision must be handled in a synchronous way. The processor, therefore, sends the identification of the shared triangle and information about the point to be inserted to the interface. Let us note that if the whole input set is loaded on every processor, it is sufficient to send only index of this point, otherwise its coordinates must be specified in the message. The interface enters a critical section, finds the shared triangle, subdivides it and assigns identifiers of negative values to new triangles. For each new triangle, it then determines which processors share the triangle. This is done by comparing the geometric position of the triangle and the bounding boxes of every processor. *Figure 8.7* shows different cases that must be detected. If a new triangle is not shared, i.e., it lies fully inside one bounding box only, it is discarded, otherwise identifiers of processors are stored into the data structure of the triangle. Assuming that processors are labeled by integers forming an arithmetic progression with difference one, we store their identification in a bit-array, where the *i*-th bit is one if the *i*-th processor shares the triangle. If the number of processor is limited to some small value, e.g., 32, the binary array can be superseded by an integer.

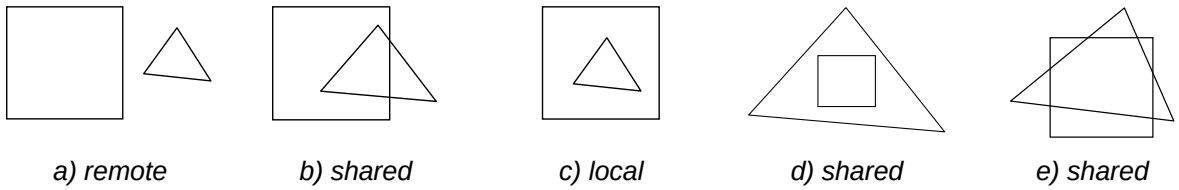


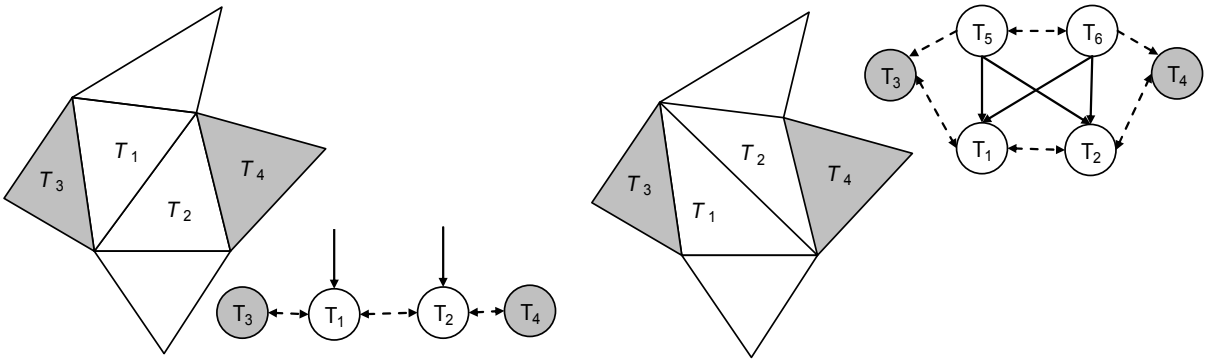
Figure 8.7: *Different mutual positions of a triangle and a bounding box.*

After the interface finishes the subdivision, it contacts all processors (including the originator of this synchronized operation) that share the subdivided triangle sending them its identifier, the point to be inserted and also some additional information such as identifiers of new triangles. As the main thread could be suspended (e.g., because the processor waits for the response from the interface), the message is received in the second thread. The main thread, however, may perform some local operation and we have to avoid simultaneous modification of the triangulation. Therefore run of both threads, i.e., the main thread and the second thread, must be synchronized. We use a critical section that is entered in the main thread at the beginning of the algorithm, left and immediately reentered in each step of the legalization, left also before the processor contacts the interface and reentered after the synchronized operation is performed and, finally, left when all points were inserted. When the message from the interface is received (in the second thread), the processor enters the critical section, performs the operation, leaves the critical section and acknowledges the change to the interface. Let us note that remote triangles are not constructed. As the main thread enters and leaves this critical section regularly, the time spent by the second thread inactively is not significant.

When all processors performed the operation, the interface proceeds with the legalization of shared triangles and after that it leaves the critical section and sends a message to the originator informing that the operation has been finished. Let us note that any synchronized operation consumes a lot of time and, therefore, an inactive waiting for its completion harms the efficiency of the algorithm. It seems reasonable to start several insertion threads (similarly as in the pessimistic method for symmetric multiprocessors) and when a thread has to contact the interface, another thread is released and it continues with insertion of its point. This approach, indeed, requires some additional time for maintenance of these insertion threads

and some time for their synchronization and, therefore, it is an open question how many threads should be used. We can expect that to employ more than four threads is useless.

The legalization offers more cases to be dealt with. If both triangles and all their neighbors are local or there is just one shared neighbor or there are just two shared neighbors not sharing a vertex, e.g., positioned as in *Figure 8.8*, the operation is performed locally exploiting the trick with the swap of identifiers as in the subdivision. If local triangles to be legalized have more than two shared neighbors or their neighbors contain the same vertex, this trick does not help and the interface has to be contacted to update the connectivity.



a) before the legalization *b) after the legalization*

Figure 8.8: *The operation of legalization of local triangles with just one shared neighbor (denoted by light gray) and appropriate changes in the DAG structure. Dashed arrows show the connectivity between triangles, solid arrows history of changes (i.e., the parent triangle and its children).*

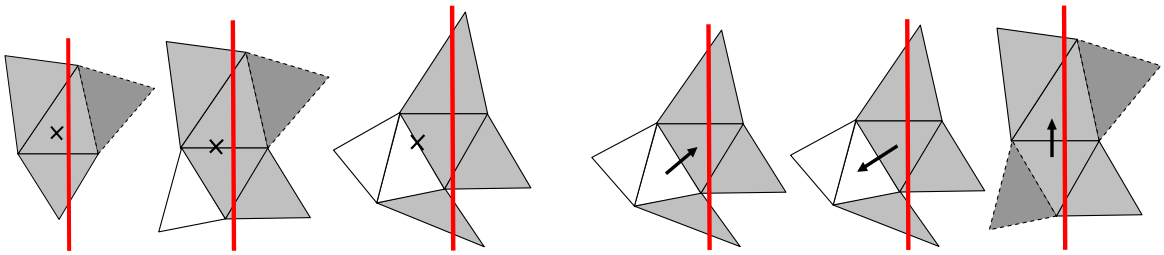
When the first triangle, i.e., the triangle from which the wave of legalization comes, is local and the second one is shared, the operation has to be performed, similarly to the subdivision of a shared triangle, in a synchronized way. Thus the processor contacts the interface, the interface performs the local transformation (i.e., swap), contacts all processors sharing at least one of triangles sending them all required information and when these processors complete the operation, the interface continues with the legalization of newly created triangles and when the process stops, the originator of the synchronized request continues with its work.

During the legalization on the interface, other two cases might appear: either both triangles are shared or the first one is shared and the second one is local. The first case is handled similarly to the previously described one. It is clear that if the second triangle is local, the legalization cannot proceed on the interface because the information about the local triangle is missing and, therefore, the interface contacts the processor having the triangle inside its bounding box. This processor enters a critical section, inserts sent identifications of triangles into a local queue of postponed swaps and leaves the critical section. The interface then proceeds with the legalization elsewhere.

A processor processes postponed swaps stored in the queue when it finishes the insertion of current point. As the triangulation could change, it is necessary to check whether the operation is still valid, i.e., both triangles exist. If the outcome of this test is positive, the processor contacts the interface providing it by information about vertices of local triangle in order to perform the synchronized operation.

Figure 8.9 summarizes all cases that require synchronized processing. Let us note that the subdivision stage when the point to be inserted lies on an edge is handled similarly to the

local transformation between shared and local triangles – indeed, the information about the point to be inserted must be added into the message.



a) the subdivision *b) the legalization*

Figure 8.9: *Synchronized operations in E^2, the point to be inserted is denoted by a cross, the direction of legalization wave by an arrow, white triangles are local, light gray shared and dark gray triangles inaccessible to the processor originating the request.*

The extension of synchronized operations to E^3 is straightforward, however, the implementation is quite complex because we need to differentiate many cases, which is caused by the fact that each kind of transformation considered in the sequential algorithm introduces several different cases for a parallel algorithm. The insertion of a point on a common edge of shared and remote tetrahedra seems to be extremely difficult. These implementation problems significantly harm usefulness of the described algorithm.

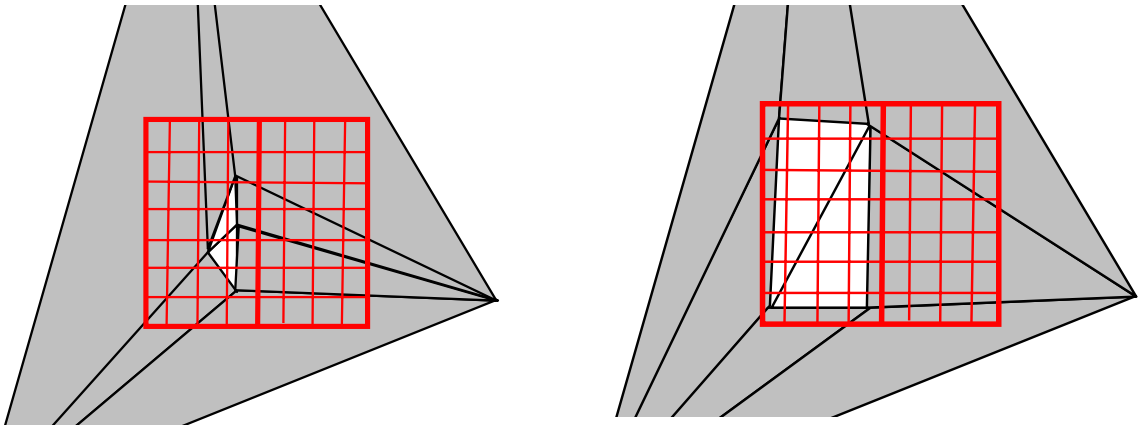
When the insertion finishes, each working processor extracts its local simplices and the interface extracts its shared simplices. The interface collects all extracted parts and their union forms the requested Delaunay triangulation.

The operation flow that we have just described suffers from several problems. First, as the number of operations to be handled in a synchronized way grows with the increasing number of processors and the interface can process only one request at a time, the performance of the algorithm drops down very quickly. Let us propose an improvement of the described algorithm that could significantly reduce the number of requests. The first local simplex is constructed when a processor finished insertion of the first three (or, in E^3, four) points into the triangulation. As points are inserted in a random order (reasons are described in Section 5), the area covered by this first simplex may be small and, therefore, the probability that the next point lies inside this local simplex is low, i.e., synchronization will be probably needed. Using the grid from Mueller's algorithm for points subdivision (see Section 6), we can find better starting points in corners of the bounding box – see *Figure 8.10*. As the first local simplex is now larger, we reduced the number of insertions into shared simplices, i.e., the number of requests is decreased and, therefore, the efficiency grows.

As the construction of the first local simplex must be done in a synchronized way, it should be faster to let the interface prepare a primary triangulation of points lying in corners and send this triangulation to every processor.

Despite all these improvements, the interface is still a bottleneck and, therefore, the overall number of processors is limited to a small number. Indeed, some functionality of the interface, e.g., the subdivision of points, could be moved onto working processors, which could bring a better scalability of the system. However, it would also make the implementation more complex. Fortunately, if we focus with the operation flow on E^2 case only, we do not need to employ large number of processors because, in practice, the largest data set needed to be

processed contain about 20 millions of points, which can be handled by 5 processors using a walking technique (the DAG would require up to 16 processors).



a) random *b) in corners*

Figure 8.10: *The influence of selection of the first four points on the size of area covered by local triangles in E^2, white triangles are local, light gray shared.*

8.5 Data flow

The data flow approach is an extension of the optimistic method described in Section 7 for clusters of workstations. Each processor can access any simplex in the Delaunay triangulation for any purpose. At the beginning of computation, the first processor, master, constructs the auxiliary big simplex and sends its identification to all other processors, workers. Unlike the operation flow, every simplex is stored only on one computer and, therefore, it is possible to fully identify a simplex by a pair $\{ID, ptr\}$, where ID is an integer uniquely identifying the processor in whose local memory is the simplex stored and, indeed, ptr is the traditional 32-bits local pointer to this storage. We supersede all 32-bits pointers used in data structures implemented in the original sequential program by this 64-bits long full identification.

Afterwards, each processor starts the insertion of its points into the triangulation. Whenever the processor W_1 needs to operate with the content of a remote simplex (i.e., to access the data of a node describing this simplex), it sends a request specifying the full identification of simplex to the processor W_2, which stores this simplex, and it waits for the reply. When W_2 processor receives a request for a simplex, it simply extracts the location of the simplex in its local memory from the sent identification and sends the data of this simplex in a response. There is no doubt that the extraction is much faster than the searching for simplex in a hash table used in the operation flow. After the processor W_1 receives data, it performs the operation. If the operation involves a modification of the simplex, the processor sends the modified data back to W_2 and the processor W_2 updates the content of the simplex. Let us note that as concurrent modification of a simplex by two or more processors has to be avoided, we need, indeed, to implement some synchronization mechanism.

Any communication takes some time and, therefore, it affects significantly the overall performance of the algorithm. In order to improve the efficiency, it is necessary to reduce the amount of data to be transferred per one communication and the amount of required communication. As a processor accesses the same simplex several times during the insertion of a point, it is useful to store the retrieved remote simplex into a temporary storage, cache, and send the data of requested simplex only if the cached version is outdated. Then, when

the processor W_1 wants to access a remote simplex, it specifies the version currently available to it in the request. If the simplex stored in the local memory of W_2 has been updated since the processor W_1 obtained its version (or the simplex has not been ever cached), the data of the required simplex is sent in response, otherwise the request is just acknowledged (by a message of zero length). Let us note that since the processor accessed the remote simplex for the modification purpose, the cached version is always the current one until the insertion of its current point is completed because it cannot be simultaneously being modified by another processor. This means that we can avoid any communication. A general scheme of the data flow approach with caching is given in *Figure 8.11*.

It is quite clear that the cache size is limited and, therefore, it might not be capable to hold all remote simplices required during the computation. If the cache is full, then before a new simplex can be stored in, one of simplices currently in the cache must be removed. A simple and very popular basic replacement policy is Least-Recently-Used (LRU) that always removes the simplex that was accessed least recently. An implementation of the LRU policy usually exploits double linked list data structure sorted by timestamps of accesses to simplices in the ascending order. We slightly modified this policy in such a manner that it is not allowed to remove simplex that has been accessed for the read or modification purposes since the insertion of the current point started. If the cache is full and no simplex can be removed, then the cache is temporally enlarged.

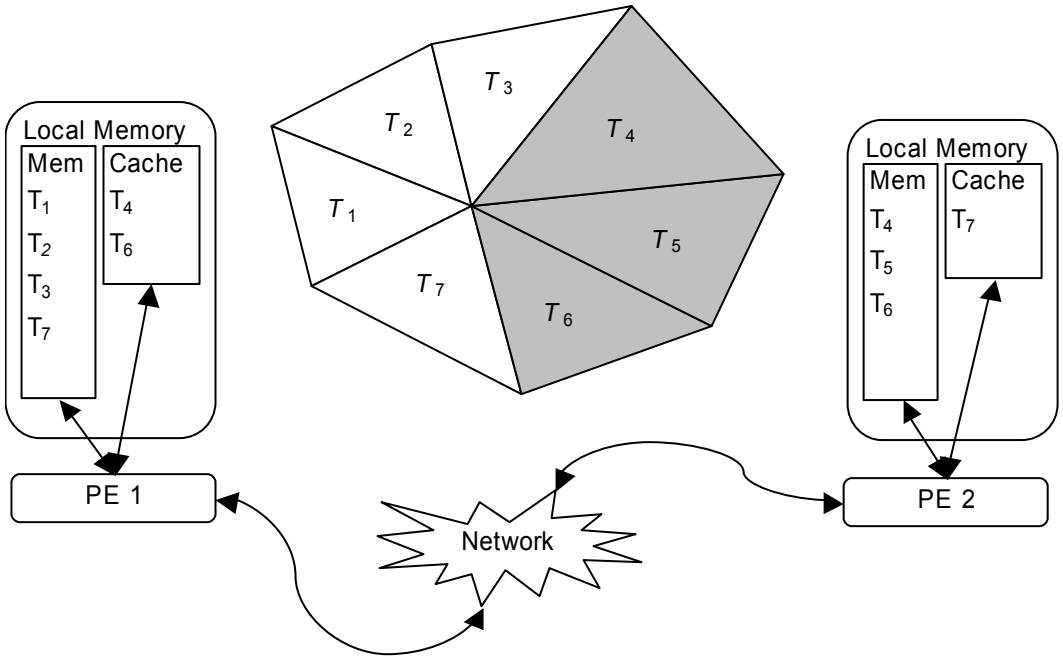


Figure 8.11: *An example of a general scheme of the data flow approach for a triangulation of seven simplices distributed over two processors.*

Figure 8.12 demonstrates the behavior of caching. Let us suppose to have a cache currently capable to hold just three items. We want to insert a point into the triangle T_7 in the triangulation from *Figure 8.11*. To complete the subdivision, the processor needs to access already cached triangle T_6, which causes the move of the pointer to T_6 into the head of the LRU list – see *Figure 8.12a*. In the legalization, the processor needs to operate with neighbors of the triangle T_6. First, it accesses the triangle T_9 and places this triangle into the cache.

Indeed, the LRU list describing the organization of the cache is updated. Now, the cache is full and, therefore, the processor removes the least recently used triangle T_4 from the cache and uses the newly free entry for the triangle T_5 – see *Figure 8.12b*. As the legalization proceeds, it is necessary to access triangles T_{10} and T_{11}. All triangles in the cache were accessed during the insertion of the current point and, therefore, we cannot remove any triangle from the cache. The current cache size is increased and triangles T_{10} and T_{11} are placed in. When the processor completes the insertion of its point, there remain 5 triangles in the cache – see *Figure 8.12c*. If the current cache size is beyond the allowed limit, the cache is shortened destroying entries occupied by least recently used triangles.

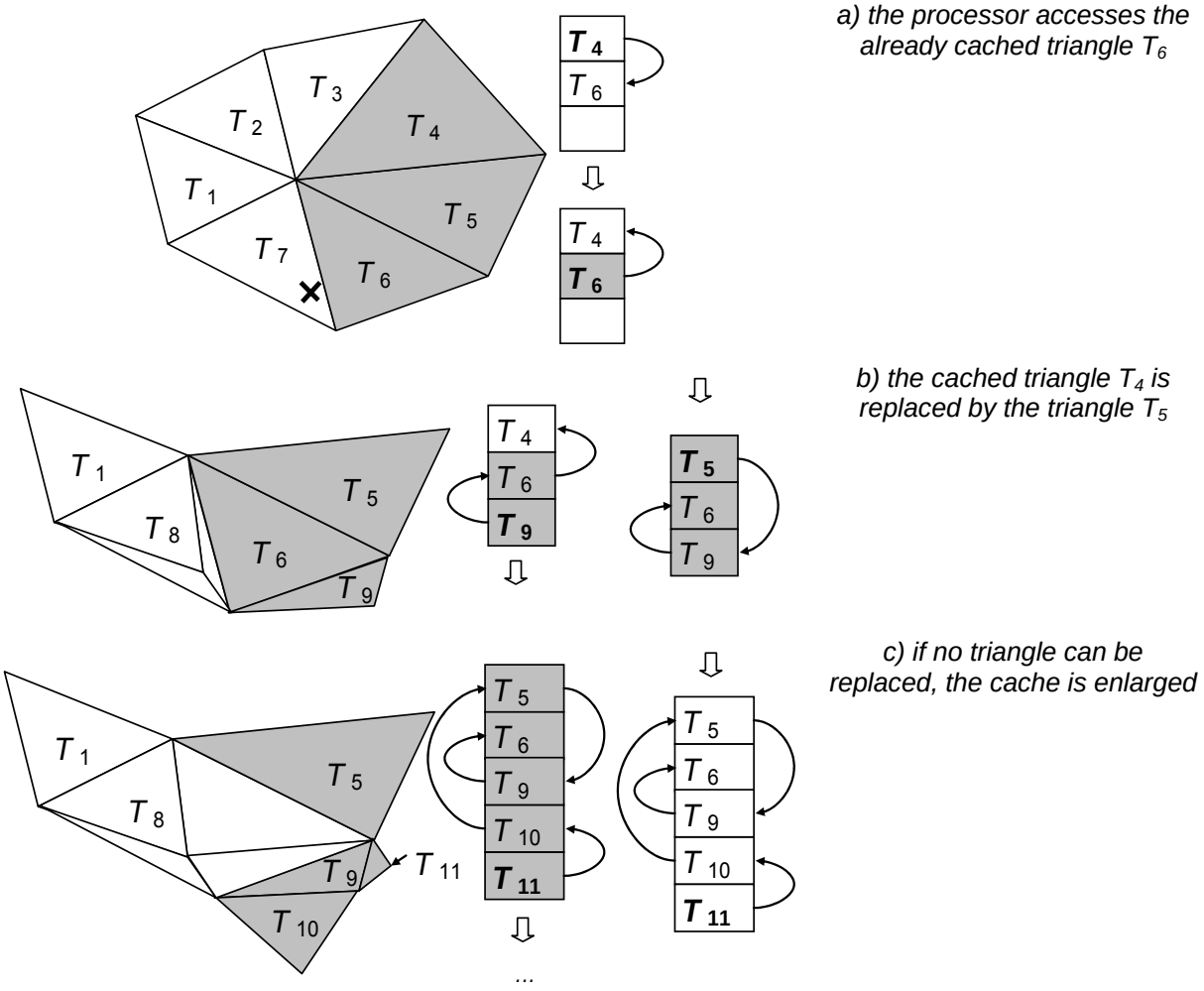


Figure 8.12: *An example of the behavior of caching policy during the insertion of a point. Changes in the triangulations are shown in the left, changes in the cache organization in the right. Bold text denotes head of the LRU list, shaded slots in the cache contain triangles that have been accessed since the insertion started.*

Although the performance is greatly enhanced due to the caching strategy, the communication still stands for a serious bottleneck. Let us, therefore, propose another improvement. When a processor needs to access a simplex, it will likely require an access to at least one adjacent simplex in a near future. A precise detection of simplices that are going to be accessed is not an easy task and it is quite time consuming. As the number of adjacent simplices, i.e., neighbors, is small, let us to retrieve them all in one communication. Although only some of

them will be accessed, the efficiency of the proposed algorithm, according to our experiments, is greatly improved. This is because the actual time needed for the transferring of data via 100 Mbs Ethernet network is negligible in comparison with the time consumed in routines for sending and receiving of the request (even if the low-level fast socket API is used). It would be also possible to retrieve neighbors of neighbors in the same message or we can go even further and retrieve neighbors of these simplices. As the time needed for the detection of simplices to be retrieved grows with the increasing level of recursion and, indeed, the time for actual transferring grows proportionally to the number of bytes, it is a question, how many simplices should be requested in one message.

We have just given a general overview of the data flow approach. Now, let us describe the data flow in detail. It is quite clear that a simplex can be accessed simultaneously by all processors that need this simplex for the read-only purpose. Simultaneous modification is not possible and, therefore, when a processor wants to modify a simplex, it has to contact the owner of this simplex, i.e., the processor that stores the simplex, and get an exclusive access to the simplex. As it was already discussed in Section 7 (see the optimistic principle), this can be done with a use either of the Delaunay empty circum-sphere property, or of memory locks. The first option, according to our results (see Section 9), does not work in E^3 efficiently and, therefore, we decided to exploit the second one. Let us remind the approach of memory locks. A "lock" parameter is added into each simplex. The zero value of the parameter means that the simplex is unlocked and non-zero value j+1 means that it is locked by the processor W_j. When the processor W_k receives from the processor W_i a request to get an exclusive access to some simplex, it has to lock this simplex for W_i. If the simplex has been already locked for another processor W_j, the processor starts a new thread and waits (in this thread) until the required simplex is 'free'. Afterwards, the simplex is locked for W_i, a reply is sent to W_i and the thread terminates its work. Let us note that a processor unlocks all its simplices when it completes the insertion of its current point. According to our experience, the successive unlocking of simplices during the insertion is useless.

It is necessary to avoid mutual waiting of processors, i.e., the deadlock. Therefore, whenever the processor cannot lock a simplex, it has to perform some check in order to decide whether it can start waiting. We can identify two different check strategies. In the *priorities strategy*, the processor can proceed with waiting only if its priority is larger than the priority of blocking processor. This test is very simple. Moreover, if priorities are static, i.e., they do not change during the computation, the processor does not need to communicate. On the other hand, this strategy is too cautious and, therefore, the processor often undoes the changes unnecessarily, which could increase the number of communications. According to our experience, better choice is the *detection strategy* where the processor checks directly whether the waiting would cause the deadlock. This detection requires always some communication. A local “waiting-for” variable was added into the address space of every processor. The zero value of this variable means that the processor does not wait and non-zero value i+1 means that it waits for the processor W_i. Let us suppose the processor W_k has found out that the simplex required for the modification purpose by the processor W_i is already locked by the processor W_j. It sends, therefore, a message informing about the existence of blocking processor W_j to W_i. The processor W_i sets the “waiting-for” variable to j+1 and acknowledges the message. Afterwards, W_k. sends a query message to the processor W_j. When a processor receives the query message, it has to check its “waiting-for” variable. If the value is not zero and does not equal to i+1, the query cannot be evaluated by the processor and, therefore, it is forwarded to the appropriate processor denoted by this value. Otherwise, the processor sends a reply to W_k. informing it that the deadlock has been, the value is i+1, or has not been, the value is zero, detected. If the processor W_k receives the negative outcome of detection, it waits

inactively until the simplex is not locked, then it sends a message to W_i in order to reset its "waiting-for" to zero and repeat the try to lock the simplex for the processor W_i.

In the case that the deadlock has been detected, a message about the failure is sent and the run of the waiting thread on W_k is terminated. The processor W_i has to reset its "waiting-for" to zero, undo all changes performed in the subdivision or the legalization and return back to the location phase. Therefore, the transaction mechanism has to be incorporated into the data flow. When a processor wants to access a simplex for the modification purpose first time, so called *shadow copy* of the whole content of this simplex is created. If the processor has to give up the insertion, the content of simplex is restored using the data from its shadow copy. Memory allocated for shadow copies is released, i.e., shadow copies are destroyed, after the processor completes the insertion of its current point. A shadow copy is also created when the simplex should be updated by sent data but the processor currently accesses this simplex in another thread running on this processor.

This means that shadow copies always store already confirmed and consistent version of triangulation. The processor sends the shadow copy of simplex instead of its current content because this content might be being modified at the time of simplex retrieval and, therefore, the current data may be inconsistent. Let us note that if an inconsistent data were retrieved and cached, it could seriously affects the quality of the resulting triangulation or it could even lead to a program crash (especially, in cases that the performed changes had to been undone). It is clear that when the simplex is to be retrieved for the modification purpose, there is no shadow copy of this simplex or the copy contains the current data.

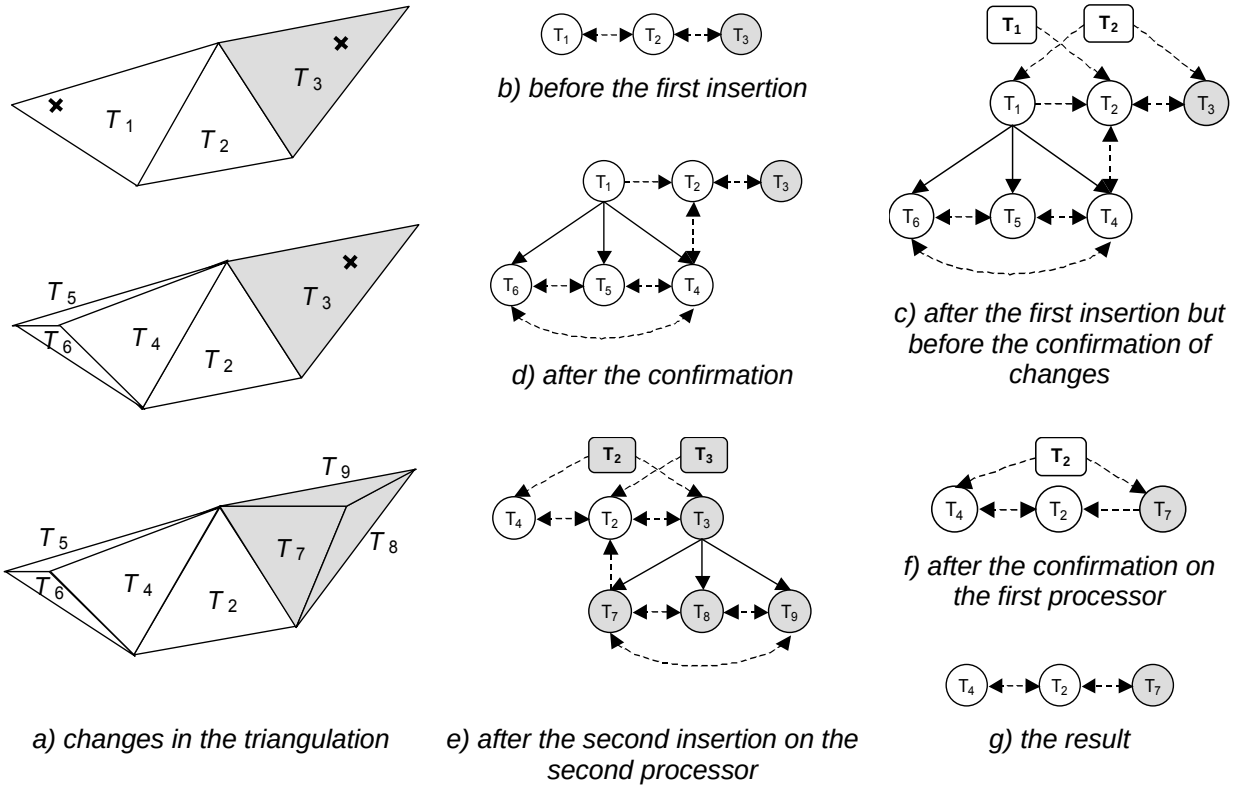


Figure 8.13: *The insertion of two points into the triangulation by two processors (their simplices are denoted by white and gray colors) and the appropriate changes in the DAG data structure (dashed arrows denote the connectivity between triangles, solid arrows history of changes); shadow copies are in rounded rectangles – see text.*

Figure 8.13 shows an example of the use of shadow copies. For easier understanding, the example is given in E^2. There are two processors, say W_1 and W_2, each one wants to insert a point into the triangulation. The processor W_1 stores all white triangles (e.g., T_1 and T_2) and the processor W_2 stores any gray triangle (e.g., T_3) – see *Figure 8.13a, b*. After successful detection of the triangle T_1 to be subdivided, the processor W_1 locks triangles T_1 and T_2, creates new triangles T_4, T_5 and T_6, creates shadow copies of triangles T_1 and T_2, and, finally, modifies the content of T_1 and T_2 to reflect the performed changes – see *Figure 8.13c*.

Now, the second processor, W_2, performs the location of triangle that contains its point. Let us suppose that during this detection it requires an access to the triangle T_2. It, therefore, contacts the processor W_1 and the contacted processor W_1 sends the shadow copy of T_2 in a response. Although the sent content version of the triangle is not the current one, it is sufficient for the testing. When the processor W_2 locates the triangle T_3 to be subdivided, it has to lock triangles T_3 and T_2. The triangle T_2 is remote one and, therefore, the processor has to contact the processor W_1. As the triangle T_2 is currently locked, the processor W_2, however, must wait until the triangle is free (i.e., until the insertion is completed). When the processor W_1 finishes the insertion, shadow copies are destroyed, the triangle T_2 is locked for the processor W_2 and the current version of the triangle is sent to this processor – see *Figure 8.13d*.

The processor W_2 performs the subdivision (see *Figure 8.13e*) and sends the new version of the triangle T_2 back to its owner (i.e., W_1) to be stored. Let us suppose that the processor W_1 meanwhile began to insert next point and has accessed this triangle for read-only purpose. Therefore, the sent data is not used to update the triangle T_2 immediately but a shadow copy with this data is created and when the processor W_1 finishes the insertion of its current point or it is going to modify the triangle T_2, this shadow copy is used to update the content of the triangle – see *Figure 8.13e, f*.

Let us discuss the efficiency of the proposed data flow approach. There is no doubt that the performance rapidly decreases with the growing number of communications, i.e., with the number of accesses to remote simplices. If we subdivide points with a respect to their geometry, e.g., by Mueller's algorithm (see Section 6), it is highly probable (because of the locality of operations) that a processor will access in the subdivision and the legalization only simplices lying fully inside its region and simplices lying in adjacent regions near boundaries of this region. As a processor constructs new simplices in its local memory, we expect that the majority of accessed simplices are local and, therefore, there is no need to communicate, which is, indeed, time consuming. *Figure 8.14* shows an example of position of local simplices in the triangulation.

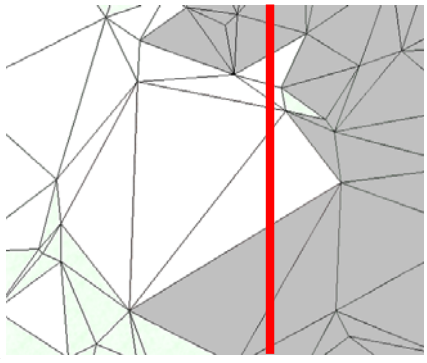


Figure 8.14: *The Delaunay triangulation in E^2 distributed over two processors. White triangles are stored in the local memory of the first processor, light gray are stored in the local memory of the second processor.*

In the first steps of insertion, the triangulation consists of several large simplices that cross over boundaries. These simplices are remote at least for one processor, i.e., this processor has to communicate in order to access them. As they are large, it is quite probable that a point to be inserted lies inside some of them. Therefore, it seems reasonable to apply the same trick as in the operation flow and let the interface to prepare a primary triangulation of points lying in corners and send this triangulation to every processor.

If a hierarchical structure is used to speed-up the location, there are, however, still many nodes on upper levels of the structure describing remote simplices and, therefore, the communication is unavoidable. This is definitely true for the DAG – see *Figure 8.15*. If the first processor wants to insert the point denoted by a cross into the triangulation, it will have to retrieve 6 remote simplices and test them to locate the simplex, local simplex, to be subdivided. One possible solution to this problem is to keep top part of the hierarchical structure always in the cache. The key issue is where to cut the structure. If the cut is too high, the number of communications will be large and the performance of the algorithm quite low. On the other hand, if the cut is too low, many nodes will have to be duplicated, which limits the size of input data set that can be processed.

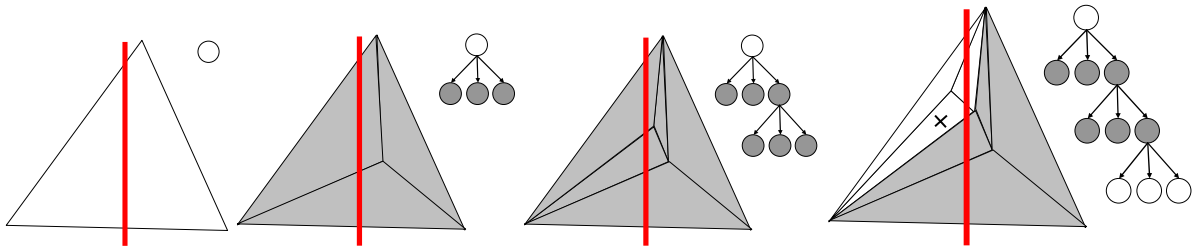


Figure 8.15: *An example of the insertion of first 3 points into the triangulation distributed over two processors (the boundary is denoted by a thick vertical line) and the appropriate changes in the DAG structure on "white" processor. White nodes denote local triangles, gray denote triangles stored remotely, i.e., in the local memory of "gray" processor.*

Therefore, a better strategy could be to avoid the use of hierarchical structures and exploit some walking technique, which, moreover, allows us to handle larger data sets on fewer computers. It is possible to adopt the uniform grid constructed in the points subdivision stage to speed-up the location. In each cell of the grid, we can store the pointer on data structure of some simplex (at the beginning it is the first auxiliary simplex) that will be used for all points lying inside this cell as a starting simplex for the walking algorithm. When a new simplex is created, the pointer on the simplex data structure is stored to every cell containing at least one vertex of this simplex. Using this approach, we can expect that (especially for uniformly sampled points) the walking technique will need to test only few simplices. Another location strategy based on a uniform grid was proposed by Žalik et al. in [Žal03].

Let us remind that the changes performed during the insertion of a point are confirmed and simplices unlocked when this insertion is completed. It might cause a communication between processors. Although a processor requires, in the worst-case, to access all simplices in the triangulation during the insertion, in practice, only few simplices are accessed. As the communication is time consuming, the algorithm could run faster if the confirmation of changes is postponed until several points are inserted. It is a question, how many points should be inserted in one transaction. If this number is too big, the processor locks large areas and, therefore, there is a bigger probability that another processor has to wait inactively or

even give up the insertion due to a deadlock. Moreover, as larger amount of memory is reserved for the cache, the processor may have not enough memory to complete its work.

The advantage of the described data flow approach is its simplicity – it can be easily implemented in E^2 and in E^3 as well. On the other hand, it requires, despite all improvements, a huge amount of communication and, therefore, the performance is limited. As there is no centralized computer, we can expect better scalability than in the case of the operation flow but lower efficiency because of the intensive communication effort.

As the data flow is an extension of the optimistic method described in Section 7.5, the simplified version of data flow algorithm is similar to the algorithm of the optimistic method given in *Figure 7.5*. The main difference is that when a simplex is to be locked, it is necessary, to detect whether the simplex is stored locally or not. If the outcome of this test is positive, the local locking routine is called. Otherwise, the appropriate processor is contacted and this processor calls the locking routine. Actually, this means that for remote simplices the routine is called remotely. The same policy applies for the confirmation of changes performed during the insertion.

8.5.1 Virtual Shared Memory (VSM) Manager

The data flow approach can be easily adopted for parallelization of various applications. This led us to the idea to develop an application independent software layer that provides universal routines for the manipulation with data, no matter whether the data is stored locally or remotely, and means for synchronization between processors. As this layer actually simulates the shared memory, we call it Virtual Shared Memory (VSM) manager. A schema of the distributed computing under the VSM is shown in *Figure 8.16*.

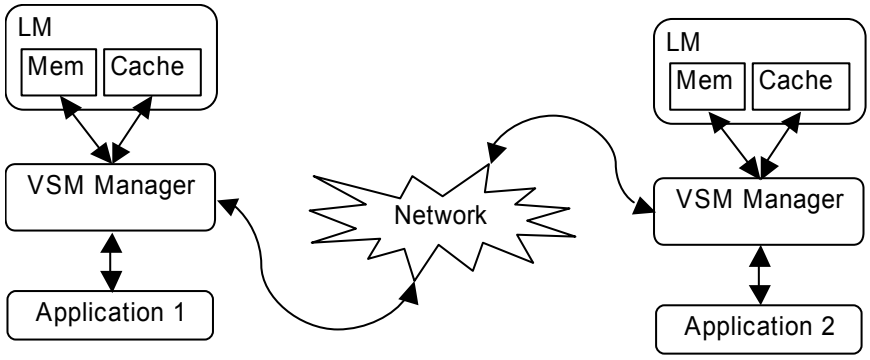


Figure 8.16: *A distributed computing using the VSM on two computers, LM is the local memory.*

The idea to simplify the parallelization for parallel computers with distributed memory by the use of a hardware or software solution that simulates the shared memory is very old, it was proposed by Kai Li in his PhD thesis from 1986 [Kai86], but it has became popular in recent few years (see also DSM in Section 2.1). While the first works were mainly intended for the massive parallelism, recent solutions are dedicated to clusters of workstations. They differ in the level of abstraction that provide. Some of them allow users to continue to operate with traditional 32-bits pointers limiting them to access at most 2GB of total memory. Such systems are typically used by applications that are quite time consuming but do not require a big amount of memory. Other solutions overcome this drawback by introducing a generalized 64-bits long pointer, which either requires unnecessarily complicated changes in user source codes or the use of specialized compiler shipped within the solution (e.g.,

commercial systems Linda or Paradise). In his PhD thesis [Bil98], Bilas gives a nice survey of virtual shared memory systems and discusses problems with their performance.

The main part of our approach is the VSM library (briefly the VSM). It is an object library dynamically or statically linked to a user application. From the distributed point of view, there is a necessity to have two programs: master and worker. In the system, there is just one running instance of master program and any number of running instances of worker program. One instance runs, typically, on one computer. It is possible to start more instances on one computer but it degrades, except for the case that the user application introduces an intensive synchronization, the performance of computation. The master program is responsible for the initialization of computation, which usually involves displaying of a setup dialog allowing the user to set parameters for the calculation, e.g., the name of data set, thresholds for filtering, etc. The worker program is dedicated for the calculation only and it cannot interact with the user. Very often master and worker programs are written according to the farmer-worker programming model (see Section 2). However, it is not a condition and both programs may implement the same algorithm, e.g., both participate to the construction of $DT(S)$.

The master program is executed manually by the user and it calls the master initiation routine of VSM passing the minimal and maximal number of computers to be used for the computation and the name of configuration file as arguments. From the specified file, the VSM reads UNC or IP addresses of computers available in the cluster. For each such an address, the VSM attempts to contact a special utility, the VSM admin, running on the background of the appropriate computer and sends it the worker program (including all required binary and initialization files). Usually, the VSM admin is launched automatically by the operating system when it starts and terminated when it shutdowns. The utility stores the sent files onto the local disk and executes the worker program.

When the worker program starts, it calls the VSM worker initialization routine. This routine starts a listening thread responsible for receiving of all incoming requests and waits until the computation can start. The utility sends a confirmation to the computer from which the request came. When a sufficient number of workers, i.e., instances of worker program, run, the VSM master initiation routine assigns a unique number identification, starting from zero, to every instance. The master is always denoted by the zero value. Afterwards, it sends a message containing all addresses and identifications to every worker. Each worker establishes the connection with other workers. Now, everything is prepared for the computation. Initiation routines finish and the execution of user code proceed.

At the beginning of computation, the application (both master and worker) calls the VSM to register data structures that should be distributed over computers. During the registration, it specifies a user unique identification of data type, the size of this type, references (pointers) used in this type and a maximal number of elements of this data type that can be created in the local memory. For example, in the Delaunay triangulation, we register the data structure describing a simplex, i.e., the node.

Typically, the application also calls the VSM to create synchronization primitives. Barriers, critical sections, manual-reset and auto-reset events are supported. A barrier is used to synchronize work of a group of processors. When a processor reaches the barrier, it has to wait until all its counterparts reach this barrier. Barriers are an ideal choice for an algorithm divided into stages where one stage must be finished before the algorithm can proceed with next stage, e.g., in the construction of the Delaunay triangulation, the insertion of all points must complete before the resulting triangulation can be stored. As critical sections and events are notoriously known synchronization primitives, we omit their description. Let us, however, explain the difference between manual-reset and auto-reset events. When a manual-reset

event is signaled, it remains in this state until it is manually set to non-signaled state. Usually, all processors waiting for the event are released. An auto-reset event is automatically reset to the non-signaled state after a single waiting processor has been released.

The VSM offers the use of so-called atoms. An atom is an indivisible piece of data, usually very small, that is stored in the local memory of one computer but it is accessible from any computer. By the term indivisible, we mean that the application cannot read individual bytes of atom but it has to read it as a whole into some local temporary storage and access data in this storage. The same rule is applied for the modification of atom. Typically, the application exploits atoms to exchange small pieces of data such as values of counters, names of input data sets, settings of computation, etc. An access to an atom is considered always as an atomic operation, i.e., only one processor may operate with the same atom at one moment. Besides standard read and write operations, the VSM supports also more sophisticated operations, e.g., the operation that modifies the atom only if its value matches a given mask, which is useful for low overhead synchronization.

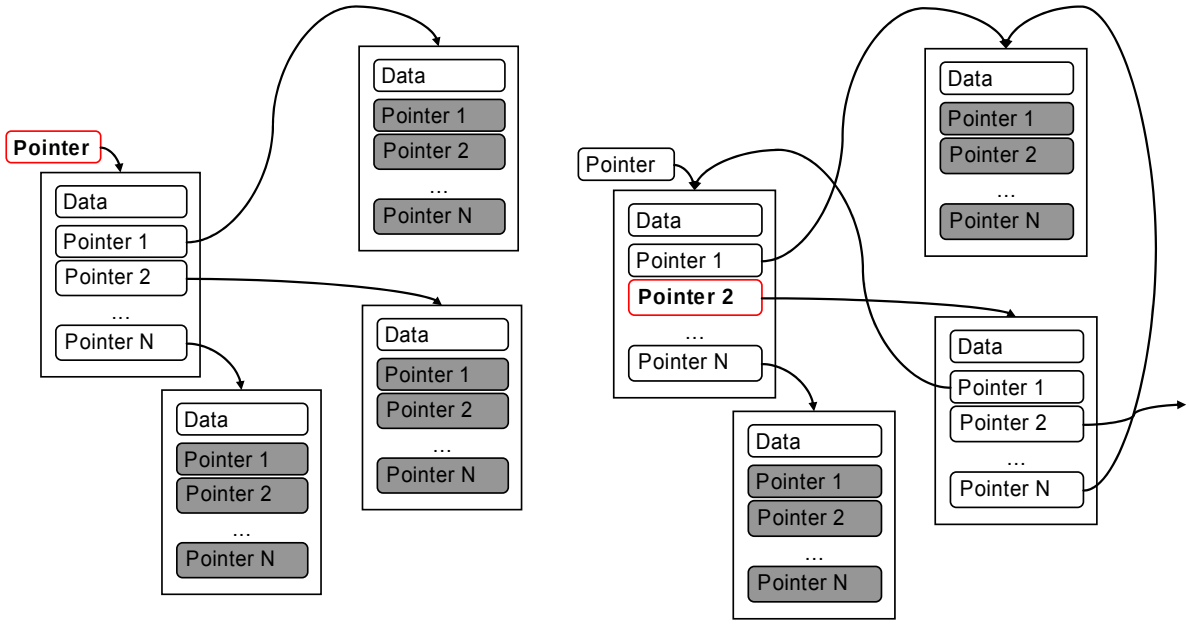
Whenever the application needs to allocate a new element of the registered data type, instead of calling standard allocation routine, e.g., *new* or *malloc*, it calls the *Add* operation provided by the VSM for this purpose. A traditional pointer, 32-bits long, is returned to the application. What is very important is that the VSM allows to programmers to continue using of standard pointers, which means that the content of an element can be read and modified in the same way as in the original sequential program and, therefore, there is no need to change source code. Internally, the element, indeed, is identified by a 64-bits number. The VSM operation to translate local pointer to this identifier is available. If the application operates with some hierarchical structure, the master program, usually, sends the identifier of root node to every worker using an atom and a manual-reset event. In our case, it is the first constructed simplex.

When the application wants to operate with the content of data element for read-only purpose, it has to call the *Get* operation passing the full identification or, if the element is stored locally or it is cached, the traditional pointer to the element as an argument. If it is necessary, the VSM retrieves the data from the appropriate owner and places it into the cache. Afterwards, it retrieves all refereed elements and updates 32-bits pointers in the structure to point to proper place. *Figure 8.17* shows an example of two steps of successive traversal of a hierarchical structure. After the first call of the *Get* operation, passing the pointer on the root element as an argument, the application can access the entire root element (i.e., its data and any pointer stored in this element) and the data in any refereed element. Pointers in refereed elements, however, might be invalid (in figure, they are denoted by gray color). In the following step, the application moves to the element refereed by the second pointer (i.e., “Pointer 2”) and performs another call passing the proper pointer as an argument.

Let us note that once elements have been retrieved in the *Get* operation, they are valid and cannot be removed from the cache until the application calls the *CancelUpdate* or the *Update* operation (will be explained later).

If the application needs to modify the element, it calls the *Edit* operation. As simultaneous modification must be avoided, the VSM in this operation attempts to get for the caller an exclusive access to this element, i.e., to lock the element. If the access cannot be granted, the application is suspended until the blocking application finishes modification of the element. In the case that the waiting would cause a deadlock, the *Edit* fails and the calling application has to return in the control flow to the stable state and to call *CancelUpdate* operation. The VSM then undoes all not confirmed changes and unlocks all elements. If no deadlock is detected, the application has to call from time to time the *Update* operation to confirm the changes and to unlock elements. Both operations also reorganize the cache and release

memory that is no longer needed. After their call, any pointer provided to the application by the VSM is invalid and, therefore, it is needed to call either the *Get* or the *Edit* again.



a) after the call of Get(Pointer) *b) after the call of Get(Pointer.Pointer2)*

Figure 8.17: *An example of the successive update of 32-bits pointers used in the application. Valid content is placed in white boxes, invalid in gray. Bold text denotes pointer that is used as an argument for the Get operation.*

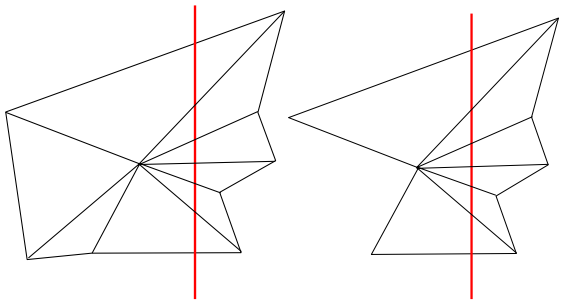
The VSM offers similar functionality for arrays of elements. An item in the array is accessed via pointer that is returned from the *Get* or the *Edit* operations called, naturally, with an index to the array as an argument. Using the indexer operator supported by almost any object oriented programming language, arrays in the original sequential code by objects can be superseded with minimal effort.

The advantage of the proposed VSM approach is the simplicity of its programming interface, which opens a possibility to parallelize a sequential algorithm even by a person not focused on distributed computing. Primarily, 32-bits pointers are supported; their translations to 64-bits long identifiers and vice versa are for the user transparent. It minimizes the number of changes to be done in order to parallelize the algorithm. On the other hand, the provided higher level of abstraction introduces quite a big overhead, especially for hierarchical structures, e.g., the DAG. It harms the efficiency of algorithm parallelized with the VSM.

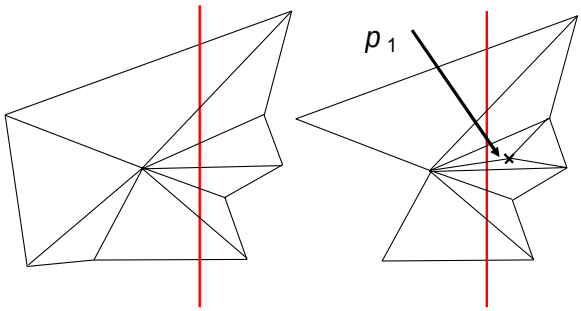
8.6 Mixed flow

Mixed flow combines the operation flow and the data flow approaches. Unlike the operation flow, it does duplicate neither the storage nor the computational effort. In comparison to the data flow, processors cannot operate with any simplex and the amount of data to be transferred is reduced. A simplex can be modified only by its supervising processor, which is the processor having the majority of its vertices inside its region or, in the case that we cannot determine the majority, the processor owning the vertex with the highest index. If a processor wants to modify a simplex that it does not supervise, it sends the operation to the appropriate supervisor and proceeds with another point. When a processor receives the operation, it enters

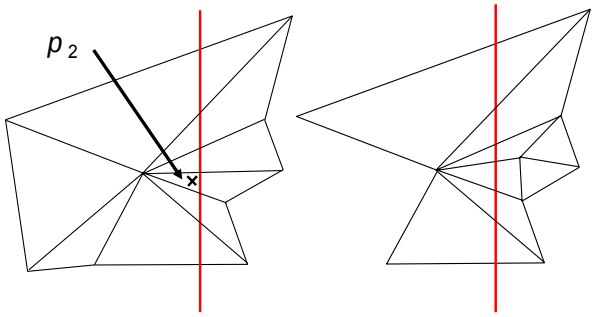
the request into a queue of postponed operations. Operations stored in the queue are processed when the processor completes the insertion of the current point. As the processing is postponed, the synchronization is not needed (unlike the operation flow). On the other hand, the triangulation could change and, therefore, some walking technique has to be used for subdivide operations in order to find the proper simplex. When the operation is completed, the processor starts the legalization of its simplices. After the legalization, the processor detects all simplices that have changed and sends them to the initiating processor in one message. The processor merges the sent part of the triangulation with its local triangulation, checks the boundary of sent part whether they fulfill the Delaunay criterion and, if the outcome is negative, it performs the legalization.



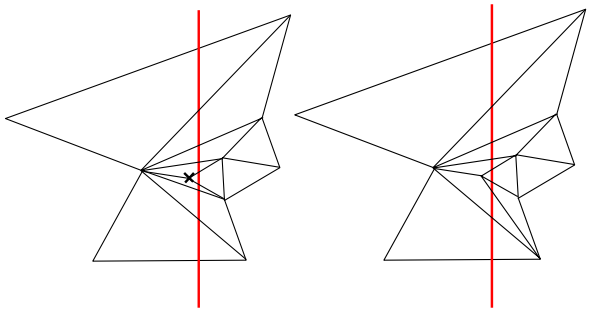
a) local triangulations at the common boundary



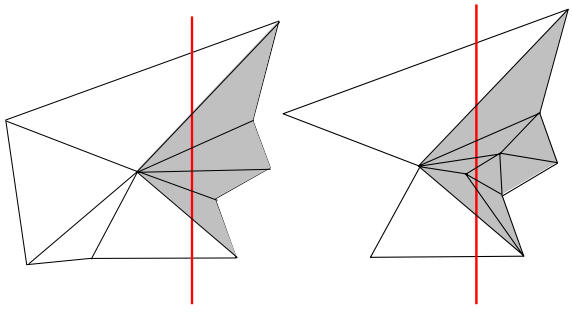
b) the second processor inserts the point p_1, local triangulation on the first processor is not changed



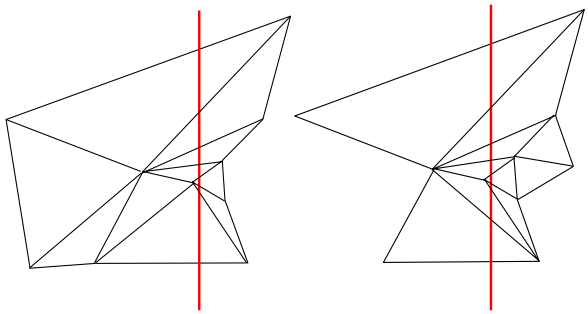
c) the first processor wants to insert the point p_2, the second processor completes the insertion of the point p_1



d) the second processor performs the sent operation, local triangulation on the first processor is not changed



e) local triangulations before the update, gray denotes the area to be updated



f) local triangulations after the update and consequential legalizations

Figure 8.18: *An example of the insertion of two points lying near the boundary in E^2.*

Figure 8.18 shows an example of the insertion of two points near the boundary of regions in E^2. The second processor wants to insert the point p_1 into the triangulation. It finds the appropriate triangle to be subdivided. As the triangle is supervised by the processor, the subdivision is performed locally – see *Figure 8.18b*. At the same time, the first processor wants to insert the point p_2. It would require a modification of triangle that is supervised by the second processor and, therefore, it sends the operation to its counterpart. Meanwhile the second processor finishes the insertion of p_1 and continues with the processing of the sent operation and its consequential operations – see *Figure 8.18c, d*. The modified part of the triangulation is transmitted and the first processor updates its local triangulation and proceeds with the legalization – see *Figure 8.18e, f*.

In the mixed flow, there is a significant decrease of required communication between processors and of the number of synchronizations. Therefore, we can expect higher efficiency that in the operation flow or the data flow approaches. On the other hand, the implementation is not simple (especially, the consistency restore) – it is comparable with the operation flow. The extension into E^3 is possible but extremely difficult (more than in the case of the operation flow) due to the numerous cases of legalization.

In this section, we propose three different strategies for the parallelization of the construction of the Delaunay triangulation for clusters of workstations. While the operation flow and the mixed flow are suitable for E^2 only, the data flow approach is general and can be simply used in both dimensions. Moreover, the data flow strategy, unlike the operation flow and the mixed flow, can be adopted for many other problems. On the other hand, its performance is supposed to be much worse than the performance of the operation flow. As the processing of large data sets is demanded especially in E^3, we focus on the data flow approach in next text.

9 Experiments and Results

Parallel and distributed solutions for the construction of the Delaunay triangulation were implemented in C++ under Microsoft Visual Studio.NET 2002 or Microsoft Visual Studio.NET 2005 Beta 1 using serial incremental algorithm implemented in Delphi 6. If not mentioned otherwise, we assume that the DAG structure is used to speed-up the location. In the operation flow, the interface subdivides input points and sends them to workers. The DCOM was used for the communication between processors – see Sections 8.2. In the data flow, points are duplicated on every computer and the low level Socket API was exploited.

As we, unfortunately, have only very limited accesses (approximately one per half a year) to any suitable symmetric multiprocessor with more than two processors, the presented results for parallel version are a collection of different experiments at various computers. Main tests were done on the following machines:

- Dell Precision 410 (2x Intel Pentium III 500 MHz, cache 512KB, 1GB RAM) with Microsoft Windows XP Professional operating system,
- Dell Power Edge 6400 (4x Intel Pentium III Xeon 550MHz, cache 1MB, 4GB RAM) with Microsoft Windows XP Advanced Server,
- 64-bits Dell Power Edge 7150 (4x Intel Itanium, cache 4MB, 800MHz, 2GB RAM) with Microsoft Windows XP Advanced Server,
- Unisys ES5000 (8x Intel Pentium III Xeon, cache 2MB, 700MHz, 2GB RAM) with Microsoft Windows 2000 DataCenter operating system,
- Dell Power Edge 8450 (8x Pentium III, cache 2MB, 550 MHz, 2GB RAM) with Microsoft Windows 2000 Server.

For additional tests, Shalla (2x Celeron 533 MHz, cache 128KB, 512MB RAM) with Microsoft Windows 2000 Professional operating system was used. For the distributed solution, tests were done on these homogenous clusters of workstations:

- 10x HP Workstation xw3100 (2x Intel Pentium IV 2.8 GHz, 2GB RAM) interconnected via 100Mb Ethernet with Microsoft Windows XP Professional operating system,
- 20x HP Compaq EVO D310 (Celeron 1.7GHz, 512 MB RAM) interconnected via 100Mb Ethernet with Microsoft Windows XP Professional operating system.

In our experiments, we used two kinds of testing data. The first group consists of artificially generated points with various distributions, such as grid, uniform, gauss, cluster, arc and sphere. The points were generated in a unit square. Points for grid distribution lie in a regular orthogonal grid. In the uniform distribution, the coordinates of points are chosen at random. Cluster distribution is formed by several groups of normally distributed points. Points for arc distribution converge to an arc. Sphere distribution provides points on the surface of a sphere. The arc and sphere were especially useful in testing the robustness of both serial and parallel implementation because these data contain many cases that are singular for Delaunay triangulation (e.g., 5 or more points laying on a sphere in E^3). Examples of tested data sets are shown in *Figure 9.1*. For experiments on symmetric multiprocessors, the tested number of input points, i.e., data size N, was chosen between 1K and 1M in E^2 and 1K and 250K in E^3. The largest tested data sets consume about 1GB of memory. Distributed programs were tested on data sets up to 4M in E^2 and 1.5M in E^3.

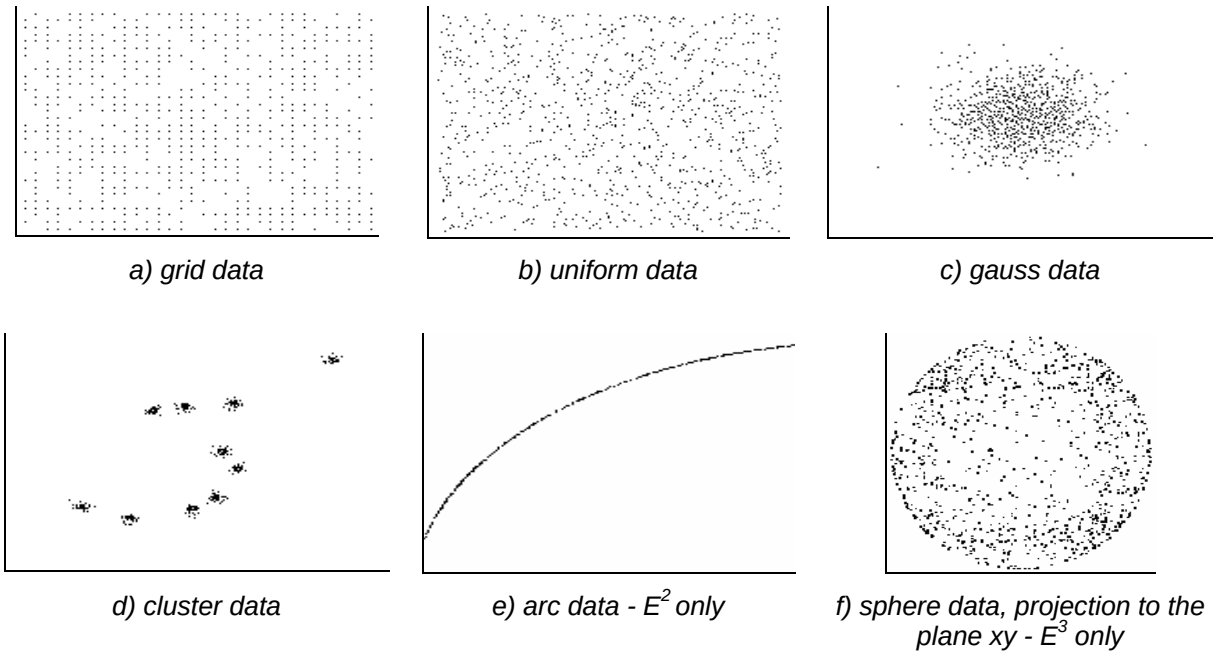


Figure 9.1: *Examples of tested distributions of the input points.*

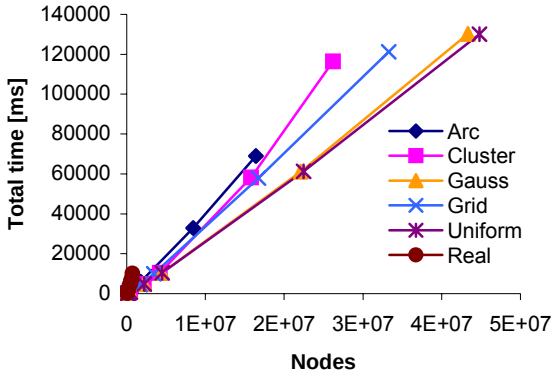
Besides the artificial data sets, we tested real data sets of terrain models (e.g., Crater Lake with 100 001 points) and some popular surface models (e.g., whale with 52 635 points, CTMayo with 98 869 or bell with 213 373). Most of these models were obtained from the Stanford repository [Sta99] and Žalik database [Žal99]. 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 in this way we eliminate singular cases. The difference between the results of both functions, however, is insignificant. The efficiency is computed as the speed-up divided by the number of used processing elements.

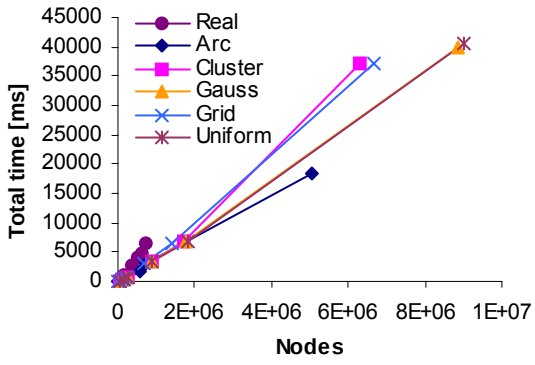
The distribution influences the types of local transformations and the linearity and the growth of the number of the required local transformations, i.e., it affects the total time. The different number of local transformations for various types of data also means different number of nodes in the DAG structure. If we plot the dependence of the total runtime on the size of the DAG structure for both sequential and parallel algorithms, these functions of the dependencies behave very similarly in almost all cases – see *Figure 9.2*. The different rate of the growth of the dependency can be noticed in E^3 for grid data sets. The explanation is as follows. The transformations of two tetrahedra to two or four to four need to lock more nodes than other types of transformations and, therefore, they consume more time. However, 'two-two' and 'four-four' local transformations are rare for other than grid distribution where they have on average about 20% of all transformations.

The just described characteristic of the dependency of runtime on the size of the DAG allows us to estimate the parallel behavior of our parallel algorithms for any data distribution if we know the parallel behavior for another data distribution and sequential behavior of both these data distribution are available. This possibility was exploited to limit number of experiments on symmetric multiprocessors with more processors (to which we have only a limited access) and we have chosen uniform data sets as representative.

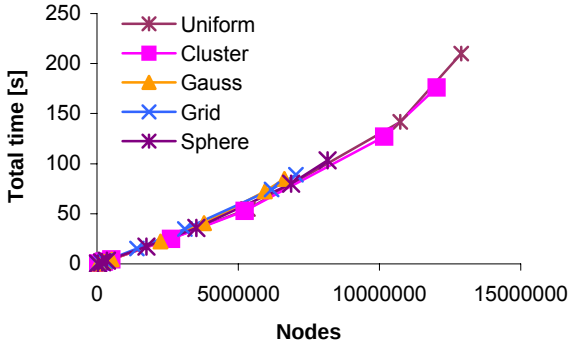
In the next text, we assume (if not specified otherwise) that in the case of parallel versions for symmetric multiprocessors, points are subdivided randomly into k groups (k is the number of used threads) in such a manner to ensure equal number of points in each group. Let us further assume that in the case of distributed versions for clusters of workstations we use a modified Mueller's algorithm described in Section 6. As two insertions of m points of the same point distribution take almost the same time, no dynamic load balancing is necessary and, in the distributed case also unwanted.



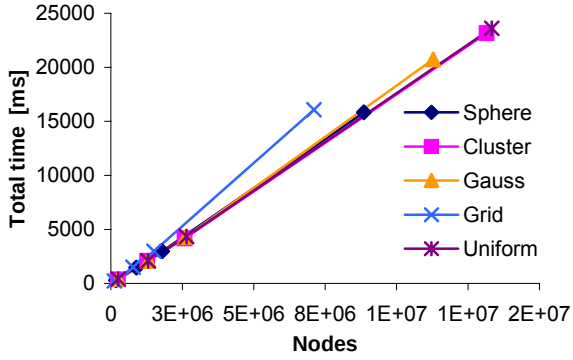
a) sequential algorithm – E^2, Dell Precision 410



b) parallel algorithm (optimistic method) – E^2, Shalla (2 PEs)



c) sequential algorithm – E^3, Shalla (2 PEs)



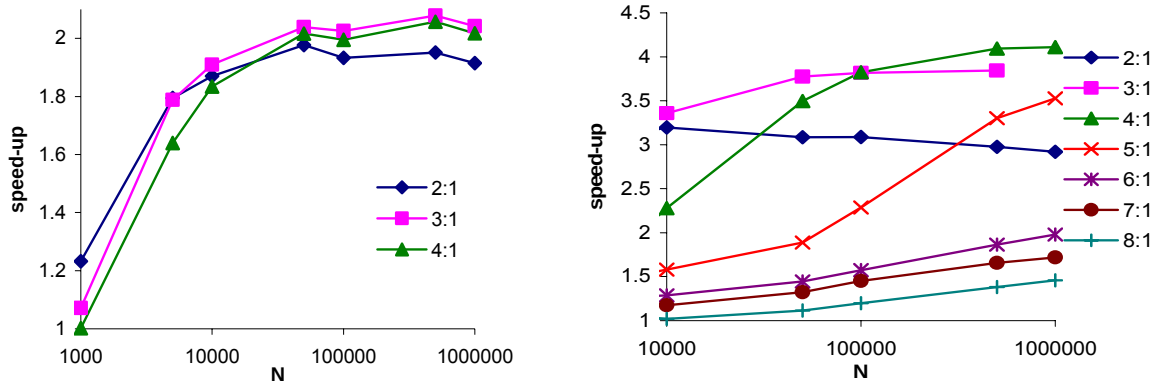
d) parallel algorithm (optimistic method) – E^3, Shalla (2 PEs)

Figure 9.2: *The dependency of the total time on the number of nodes in the DAG structure.*

9.1 The Results of Batch Method

Figure 9.3 shows the achieved speed-up for batch method for uniform data sets. The speed-up is mainly influenced by the number of input points. 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 decrease slowly.



a) Dell Precision 410 (2 PEs) *b) Dell Power Edge 6400 (4 PEs)*

Figure 9.3: *Speed-up of batch method when 2 - 8 searching threads were used.*

Let us discuss this behavior. The algorithm has to process one subdivision and zero or more, say L, swaps after the location. 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 they are local and thus L is limited to a small number in average. It means that the time needed for the legalization is also independent of the size of the data set. While expected complexity of both parts is $O(1)$, expected complexity of the location is $O(\log N)$ and, therefore, time needed to locate a triangle is significantly dependent on the data size. How does this fact influence speed-up? When the data set is too 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 down with growing data size and, therefore, we can notice a quick increase in speed-up. When the searching threads are unable to insert the point into the queue in time, the queue becomes empty and the specialized thread has to wait. As this situation happens rarely and, moreover, in such a case, only one thread has to wait, the performance decreases very slowly.

It is clear that the optimal efficiency is reached when no thread has to wait and just one element to be processed is in the shared queue. Let us assume the PRAM computational model, i.e., the model of the architecture with an unlimited number of processors, common shared memory and unrealistically cheap synchronization between processors. For a given data set, the sequential algorithm spent the total time t_1 in the location and the total time t_2 in the subdivision and the legalization. In such a case, to achieve highest efficiency for this given data set, we have to use k searching threads, where k is equal to the $round(t_1 / t_2)$. If the result of t_1 / t_2 is an integer, the achieved efficiency is optimal.

The number of available processors is, however, limited to PE_{max} in the real architectures. Let us assume that k+1 is the total number of all threads (searching + specialized threads) for a given data set and, moreover, for a given architecture, k+1 > PE_{max}. Which strategy is better: to let k+1 threads run and admit sharing of the processor time among more threads or use at most PE_{max} threads and admit waiting of threads in the algorithm? According to our experiments, the waiting of the threads has greater negative impact on the efficiency if k+1 is not 'too different' of PE_{max}. *Figure 9.3a* shows a good example.

There is no configuration (i.e., number of used searching threads) ideal for all data sizes. Therefore, we recommend a modification of the proposed batch method in such a manner to get an improved batch method that automatically estimates the number of threads that should be used for the location according to the number of input points. According to *Figure 9.3b*, the best configuration seems to be 3:1 (i.e., 3 searching threads and 1 specialized thread) and 4:1 for larger data sets (up to 1 000 000). Let us note that it corresponds to the results of analysis of the sequential algorithm as it was presented in Section 6. Use of five searching threads comes to consideration for data sets with more than approximately 1.5 millions of points. Although we did not perform an experiment on a multiprocessor with more than 4 PEs, we cannot expect that the batch method would achieve a good speed-up on such computer because the location part in E^2 consumes only 75% of the total time for common data sizes (i.e., up to one million). Probably, there exist such large data sets that would feed more PEs, however, computation of such data sets using the DAG structure needs more memory than is available on any 32-bits computer. Therefore, this method seems to be scalable only up to 4 PEs for common data sizes and, moreover, usable only in E^2 because the location in E^3 consumes less than 50% of the total time.

Let us now discuss the possibility of super-linear speed-up recognizable in almost all graphs presented in this paper. The super-linear speed-up means that the speed-up is bigger than the number of used processors. This situation generally may appear; two main reasons are in [Sie94]. The problem was analyzed in detail by Sun et al. [Sun95]. Typically, it is caused by a more efficient use of the memory cache and the caches of the processors. A memory cache influences also the time needed for processing data in the algorithm. For example, the data loaded into the cache for the thread T_0 are needed also for threads T_1 and T_2 that access them when they are still in the cache, thus they do not need additional time for their loading. This case is 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. Effect of the processors caches is, especially in batch method, notable. There is no doubt that the code stored in the cache runs faster than the code stored in the RAM. The 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. As for the sequential algorithm, it persists there for some time and then it is partially replaced by code needed for the subdivision or legalization, it means that the RAM has to be often accessed. However, in the batch method, searching threads do only the location, therefore, their code can be in the caches for longer time and so the location version takes a noticeably shorter time. *Figure 9.4* brings a confirmation of the cache-effect. We compared speed-up achieved at Shalla (cache 128 KB), Dell Precision 410 (cache 512 KB) and Dell Power Edge 6400 (cache 1 MB) for the batch method. To ensure comparable environment, we limited the run of the algorithm to two processors in case of PE 6400.

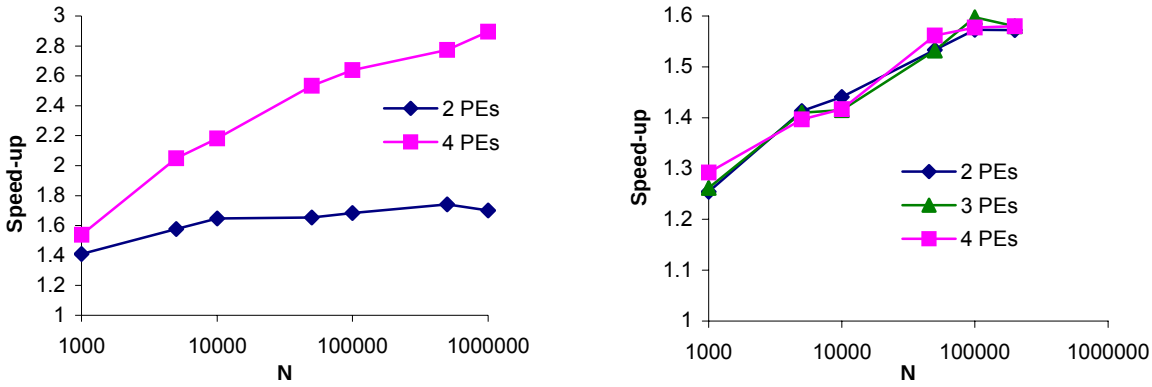
N	Speed-up		
	Shalla 128 KB	**P 410 512 KB**	**PE 6400 1024 KB**
1 000	1.132	1.072	1.642
5 000	1.698	1.787	2.218
10 000	1.606	1.910	2.391
50 000	1.695	2.039	2.342
100 000	1.749	2.025	2.422
500 000	1.745	2.079	2.399

Figure 9.4: *Influence of cache-effect on speed-up at three different computers for the batch method with the configuration 3:1 (run limited to 2 PEs).*

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

9.2 The Results of Pessimistic Method

Figure 9.5 shows achieved speed-up for pessimistic method for uniform data sets. Speed-up increases with growing number of input points. The optimal efficiency of this method is reached when a thread enters a critical section without waiting. This condition is equivalent to the condition of optimal efficiency in batch method and, therefore, the results in E^2 are similar. As the location in E^3 takes only about 30%, threads always have to wait. Therefore, the use of more than 2 PEs in E^3 makes no sense.



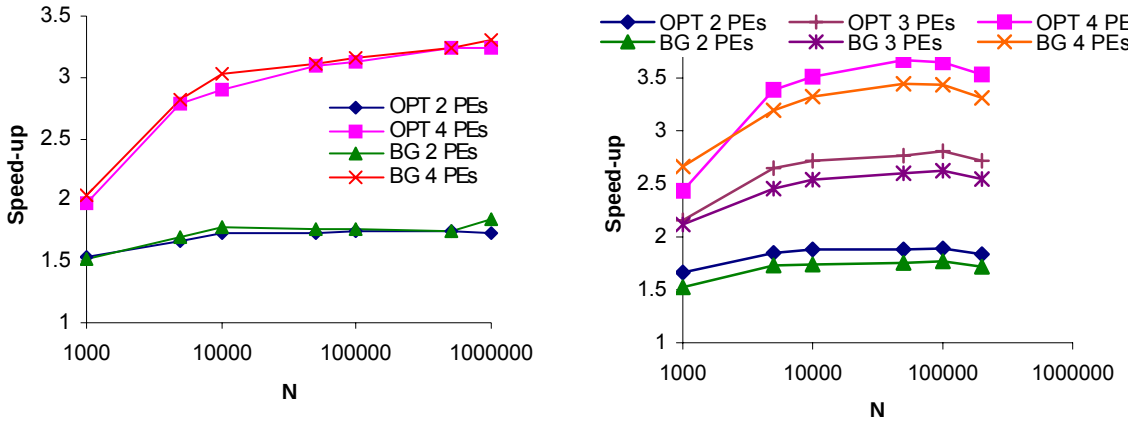
a) E^2 *b) E^3*

Figure 9.5: *Speed-up of pessimistic method - Dell Power Edge 7150 (64 bits, 4 PEs).*

9.3 The Results of Optimistic Method and Burglary Method

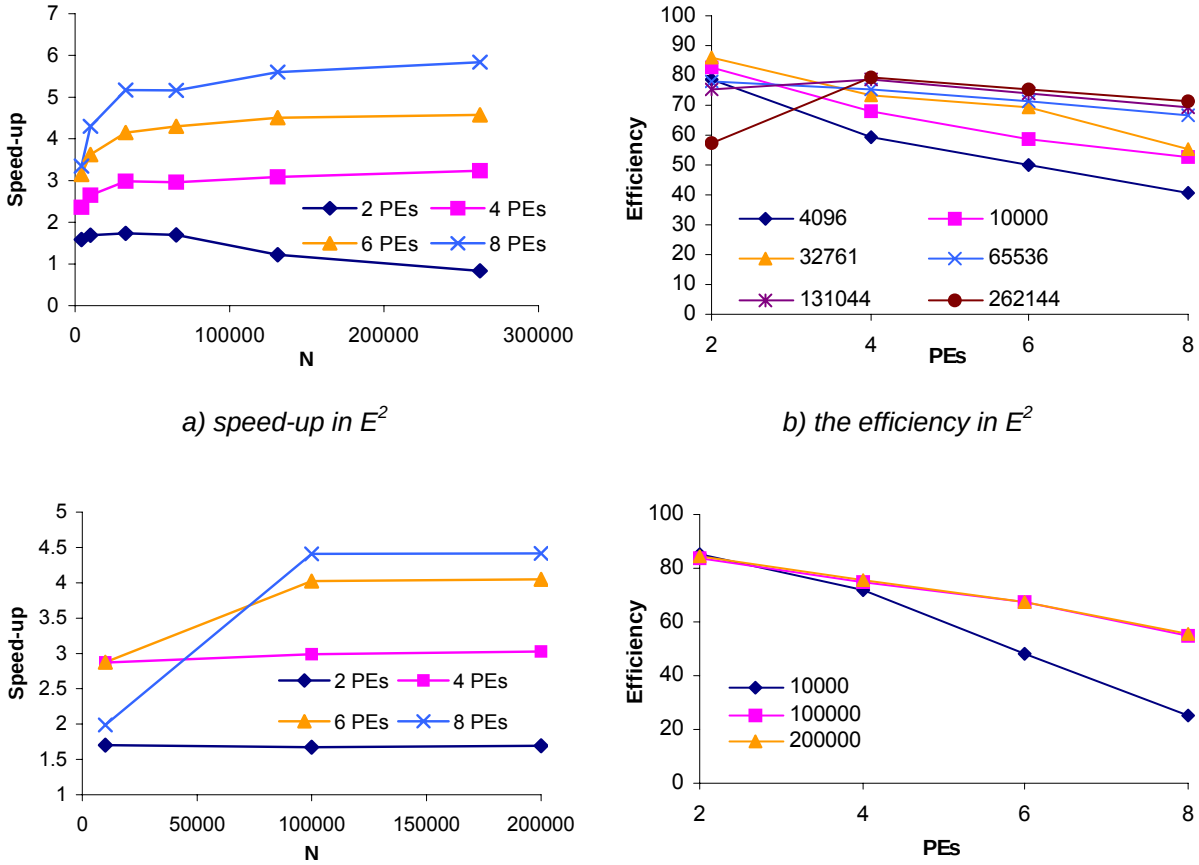
Figure 9.6 shows the achieved speed-up for optimistic method and burglary method for uniform data sets. Speed-up increases with the growing number of input points and with the growing number of used threads. In burglary method, the transactions are avoided. It has, indeed, a positive impact on the total time needed for the construction. On the other side, burglary method requires more complicated locking routine than optimistic method and it introduces some additional tests necessary for the decision whether to continue in the legalization or not. It negatively influences the efficiency. Although number of the transactions required per one insertion in E^2 is similar to number of the transactions required per one insertion in E^3, the numbers of calling locking routine are different – methods in E^3 need several times more callings. That is why burglary method achieves slightly higher speed-up than optimistic method in E^2 and lower speed-up in E^3.

According to our previous experiments [Kol02, Koh03b], both methods should be scalable at least up to 8 PEs. As we have no longer access to an appropriate multiprocessor, let us present the results of experiments with older versions of our algorithm. Since that time, we have reduced the number of required synchronization and, therefore, one can expect better results using the current version. *Figure 9.7* acknowledges the scalability of the optimistic method.



a) E^2 *b) E^3*

Figure 9.6: *Speed-up of optimistic method (OPT) and burglary method (BG) - Dell Power Edge 7150 (64 bits, 4 PEs).*



a) speed-up in E^2 *b) the efficiency in E^2*

c) speed-up in E^3 *d) the efficiency in E^3*

Figure 9.7: *Speed-up and the efficiency of the optimistic method for grid data sets. The results are valid for older versions only. The experiments ran on Dell Power Edge 8450 (8 PEs) for E^2 and on UniSys ES5000 for E^3.*

9.4 The Results of Circum-Circle Method

Our preliminary experiments revealed that circum-circle method is useless in E^3 because the threads have to wait very often. The reason is that circum-spheres occupy a large part of the space. *Figure 9.8* presents the achieved speed-up for circum-circle method for uniform data sets in E^2. Speed-up grows with the growing number of input points but the efficiency of the algorithm quickly decreases with the increasing number of used threads. The reasons for the decrease are two: the complexity of the geometric tests and the possibility that a point inserted by a concurrent thread lies inside the larger circum-circle (constructed for narrow triangles) grows with the number of used threads.

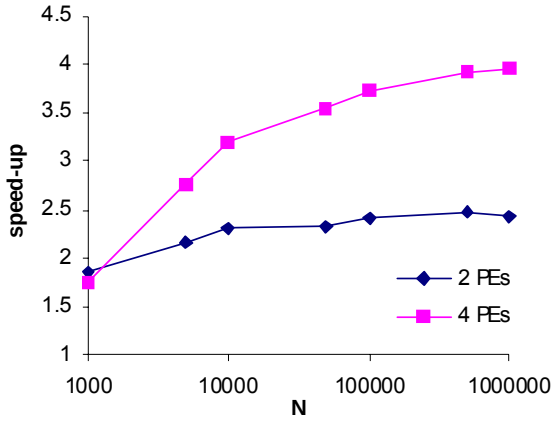
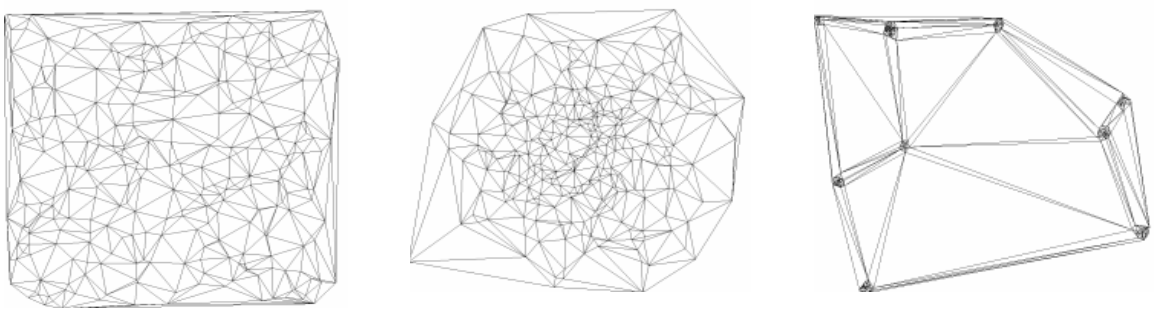


Figure 9.8: *Speed-up of circle-circum method - Dell Power Edge 6400.*

9.5 Experiments with Different Data Point Distributions

Let us discuss the influence of various distributions on the run of our methods. *Figure 9.9* shows samples of Delaunay triangulations for different data point distributions in E^2.



a) uniform data set *b) gauss data set* *c) cluster data set*

Figure 9.9: *Examples of triangulations of different types of data point distribution in E^2.*

Figure 9.10 demonstrates the efficiency of optimistic method in E^2 for different data point distributions. As we can see, the differences in speed-up among the distributions are almost insignificant – up to 12% (on average the difference does not exceed 8%). Batch, pessimistic and burglary methods behave similarly (see [Kol03, Koh04c]).

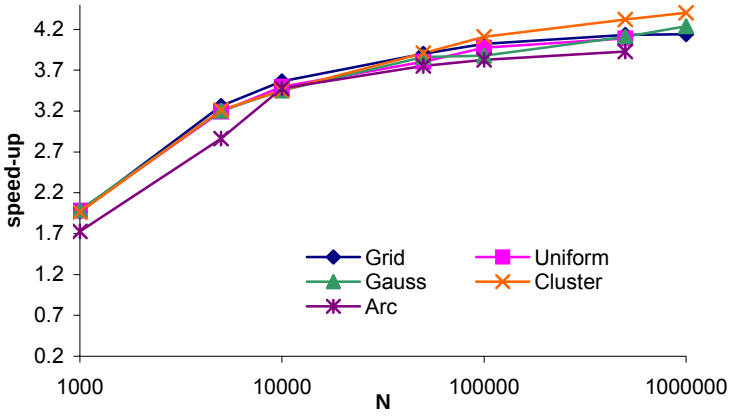


Figure 9.10: *Speed-up of the optimistic method in E^2 for different types of data point distribution - Dell Power Edge 6400 (4 PEs).*

A slightly different situation occurs when circum-circle method is tested – see *Figure 9.11*. It seems that the method of circum-circle is useless for arc data sets and the most efficient for cluster data sets. The reason for such behaviour is directly related to the previously described problem of circum-circles of narrow triangles. In arc data sets, many triangles are narrow, thus many circum-circles are huge and the probability that a thread has to wait is quite high. *Figure 9.12* shows an example of the Delaunay triangulation of arc data set.

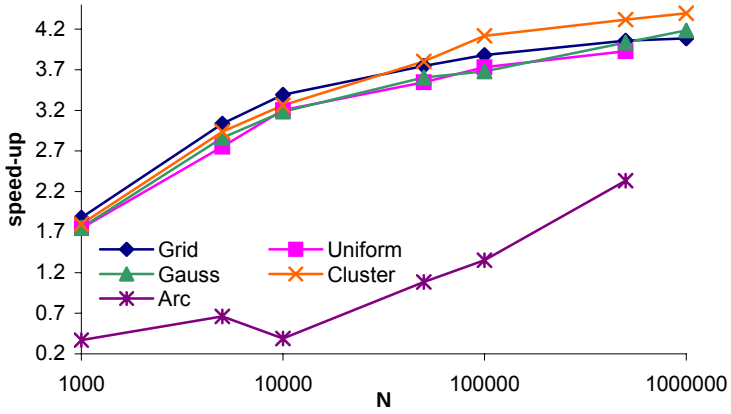
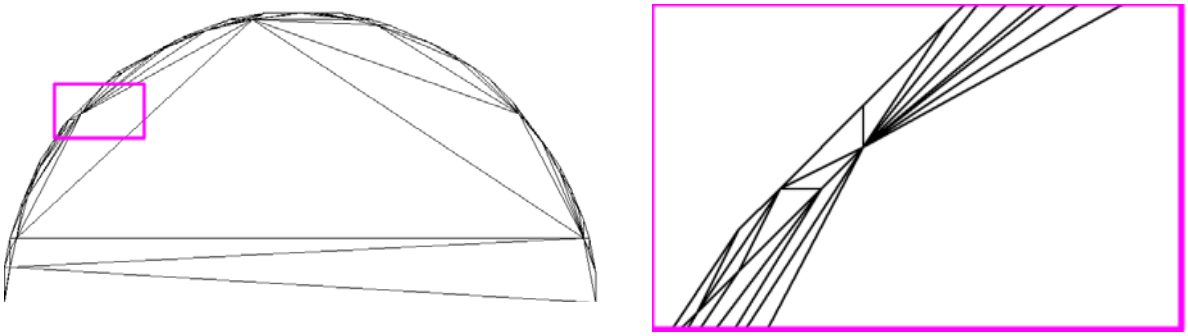


Figure 9.11: *Speed-up of the circum-circle method for different data point distributions - Dell Power Edge 6400 (4 PEs).*

On the other hand, when dealing with cluster data sets, the possibility grows slower than when dealing with other distributions. It is caused by the fact that insertion of a point into one cluster rarely influences another cluster, thus if the thread T_0 works with a cluster A and the thread T_1 works with a 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.



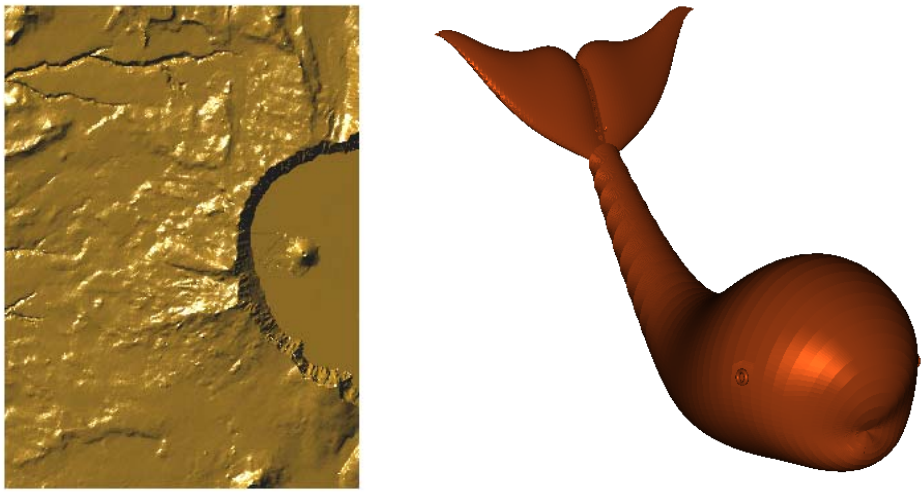
a) the entire Delaunay triangulation *b) the detail of selected area*

Figure 9.12: *An example of the Delaunay triangulation of 100 points lying on arc.*

Pessimistic, optimistic and burglary method in E^3 achieve similar results for all tested data point distributions with one exception: optimistic and burglary methods for grid data sets achieve lower speed-up than for any other data sets. It is caused by the necessity to lock more nodes in the subdivision and/or the legalization (see the beginning of Section 9). However, the grid points are not a typical input - if the user prefers a structured mesh, he will probably not use the Delaunay triangulation.

9.6 Experiments with Real Data Sets

Besides artificial data, we tested real data sets from [Sta99, Žal99] – see *Figure 9.13*. The experiments were done under the same conditions as were applied on the generated data sets. There is usually no significant difference between the results of experiments with real data sets and uniform data sets when corresponding numbers of input points are compared.



a) E^2 *b)* E^3

Figure 9.13: *Examples of triangulations of real data sets.*

Figure 9.13a shows an example of real data set in E^2. It is a digital elevation model of Crater Lake, USA containing 100 001 points. Batch method (configuration 3:1) reached speed-up 2.01 at Dell Precision 410 with 2 PEs and 4.40 at Dell Power Edge 6400 with 4 PEs. Optimistic method reached speed-up 2.50, 4.33, burglary method 2.50, 4.44 and circum-circle method 2.62, 4.09 at Dell PE 6400 with 2, 4 threads. Let us note that the model was rendered using the MVE visualization package [MVE].

Figure 9.13b shows an example of real data set in E^3. It is a model of a whale containing 52 635 points. Pessimistic method reached speed-up 1.54, optimistic method 1.95, 2.72, 3.51 and burglary method 1.75, 2.53, N/A at Dell Power Edge 7150 with 2, 3, 4 threads. Let us note that the final model in *Figure 6.26b* was obtained from the resulting triangulation by the method publicated in [Var05] – see also Section 10.

9.7 Experiments with Different Points Subdivision Strategies

In the previously described results, we assumed that points are subdivided randomly into k groups (k is the number of used threads) in such a manner to ensure equal number of points in each group. This strategy is reasonable for methods based on the batch or the pessimistic principle because the subdivision and the legalization in these methods are performed by one thread only. For optimistic methods, however, better strategy could be a subdivision of input points into k groups in such a manner that not only the equal number of points in each group is ensured but also the bounding boxes of these groups have minimal intersection.

The simplest way is to subdivide input points into k slabs in x-coordinate using modified median cut algorithm or Mueller's algorithm – these were described in Section 6. According to our experiments [Kol02, Koh03b, Koh04c], however, the additional time required by a more sophisticated points subdivision is not counterbalanced in the computation and, therefore, the use of such possibilities brings worse results. The explanation is simple: the number of cases when a thread has to wait is, especially for larger data sets, negligible, i.e., the total time spent in the waiting is lower than the additional time.

9.8 The Results of Operation Flow

In the parallelization for clusters of workstations, unlike the parallelization for symmetric multiprocessors, the total time needed for the computation is not as important as the size of input sets that we are able to process. *Figure* 9.14 shows the minimal amount of memory required for the DAG structure in the dependency on the size of uniform data sets in E^2. It can be seen that the requirements grow almost linearly. Let us note that for other kinds of input data, it behaves similarly. As a program running under Microsoft Windows platform on 32-bits computer, widely used equipment, may take up to 2GB of memory, it is impossible to process data sets containing at least 3 millions of points on one computer. As it is necessary to store also coordinates of points and, usually, some miscellaneous data structures, the memory consumption is, in practice, larger.

N	DAG		N	DAG	
	Nodes	Size [MB]		Nodes	Size [MB]
5 000	44 734	3	1 000 000	8 956 681	598
10 000	89 286	6	2 000 000	17 935 024	1 197
50 000	449 012	30	3 000 000	26 902 836	1 796
100 000	899 293	60	3 500 000	31 386 289	2 095
500 000	4 489 882	300	4 000 000	35 870 048	2 395

Figure 9.14: *The size of the DAG structure in the dependency on the number of points in E^2 (uniform data sets were tested).*

Figure 9.15 shows the total time required to process smaller uniform data sets on various number of computers using the operation flow without any improvement. Let us note that we could not perform an experiment on two computers for a data set with one million of points because the total amount of memory available on these computers (i.e., about 800 MB) for the computation of such data set was insufficient.

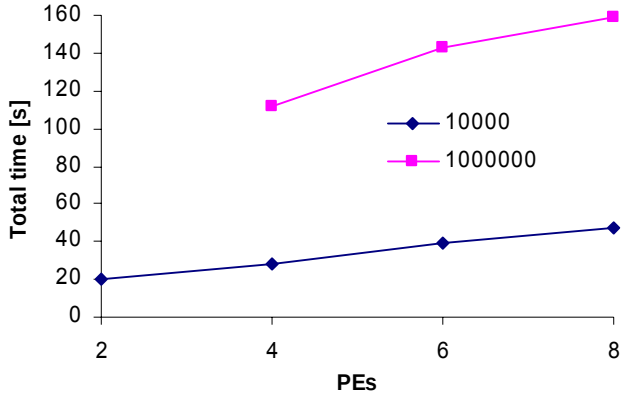


Figure 9.15: *The scalability of the operation flow method without any improvement in E^2 – tested uniform data sets on cluster of HP Compaq EVO D310.*

Surprisingly, the time increases with the growing number of employed processors and in the case of small data set with 100 000 points, it increases almost linearly. The reason is quite simple. When a processor needs to operate with a shared simplex, it has to communicate and some synchronization between processors is unavoidable. The amount of shared simplices increases proportionally to the number of processors. An operation with a shared simplex has to be performed by every computer sharing this simplex and by the interface. Therefore, the total time spent by the processing of such operations grows with the number of computers employed for the computation. As the processor that invoked the operation with a shared simplex is suspended and has to wait inactively until the operation is finished (see Section 8), operations with shared simplices are performed actually sequentially. As the total time required by the processing of an operation with a shared simplex is much larger than the total time required by the processing of an operation with a local simplex, efficiency of the algorithm rapidly decreases as we use more computers. *Figure 9.16* shows the number of synchronized operations, i.e., operations with shared simplices, and their influence on the runtime of the algorithm in the dependency on the number of processors.

N	PEs	Total Time [s]	Synchronized operations		
			Count	Time	Time / PE [s]
100 000	2	20.3	7982	4.5 / 22.2%	2.3 / 11.1%
100 000	4	28.2	21277	12.5 / 44.3%	3.1 / 11.1%
100 000	6	38.8	40908	24.9 / 64.2%	4.2 / 10.7%
100 000	8	46.8	64791	40.2 / 85.9%	5.0 / 10.7%
1 000 000	4	111.7	107446	67.3 / 60.3%	16.8 / 15.1%
1 000 000	6	143.3	169843	107.7 / 75.2%	18.0 / 12.5%
1 000 000	8	159.1	214122	155.6 / 97.8%	19.5 / 12.2%

Figure 9.16: *The amount of synchronized operations (operations with shared simplices) in E^2 in the dependency on the number of processors and the total time required for the processing of these operations in comparison with the total time consumed for the computation. Uniform data sets on cluster of HP Compaq EVO D310 were tested.*

It can be seen that each processor spends more than 10% of the total construction time inactively by waiting for the completion of some synchronized operation. With larger number of processors, the total time consumed by the interface grows – see also *Figure 9.17*. As this time is an overhead, no wonder that the algorithm is not scalable.

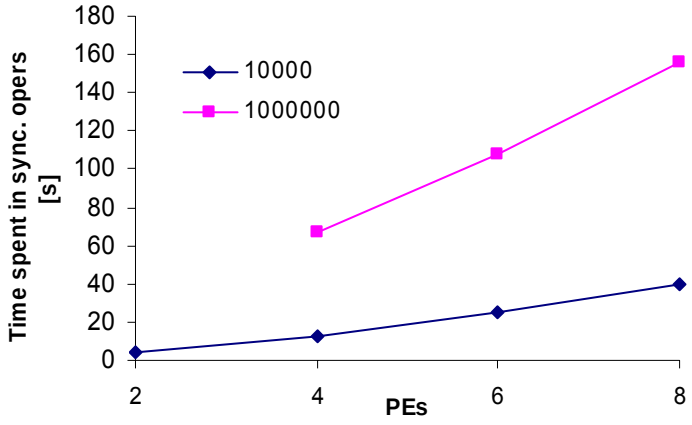


Figure 9.17: *The total time spent by the interface to handle synchronized operations in the dependency on the number of working processors – tested uniform data sets on cluster of HP Compaq EVO D310.*

First what can be done is to reduce the number of synchronized operations that are required just for retaining the connectivity by reusing old identifications for new simplices (see Section 8). This, according to our further experiments, however, does not bring a substantial improvement because the total time spent in these operations is negligible (about 0.3%). To decrease the time spent inactively by the waiting, we tried to run more insertion threads on one computer. As only one thread may run at one time, some mechanism to synchronize the work of threads is necessary. Before a thread starts to wait for the completion of a synchronized operation, it releases another thread and the released thread proceeds with the insertion of its point. *Figure 9.18* shows the total time required for the processing of uniform data sets up to four millions of points on 8x HP Compaq EVO D310 for various number of threads used for the insertion.

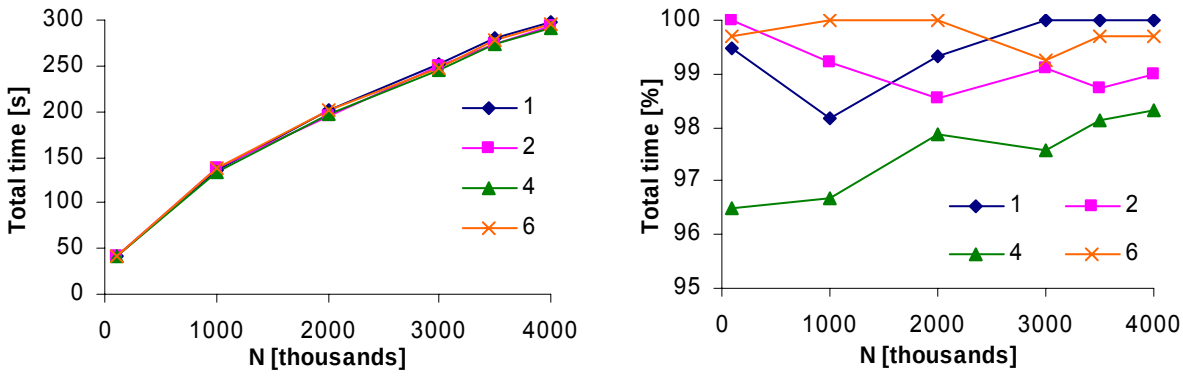


Figure 9.18: *The total time required for the construction of the Delaunay triangulation in E^2 of uniform data sets on 8x HP Compaq EVO D310 for 1, 2, 4 and 6 insertion threads.*

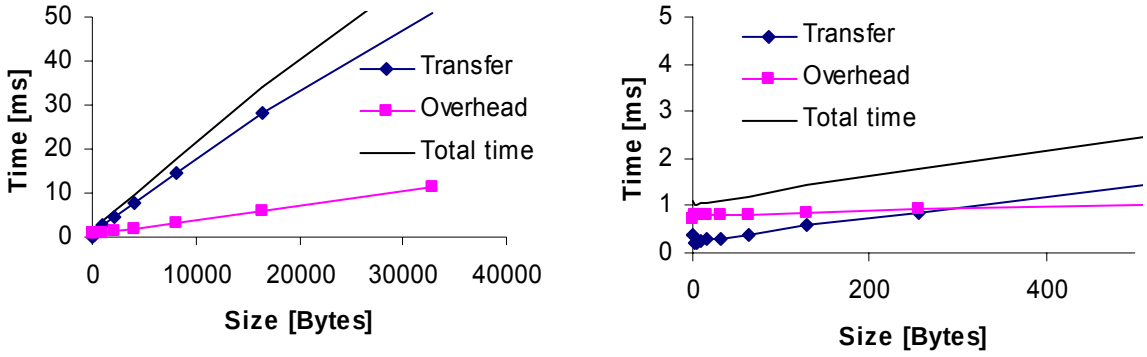
Despite our expectation, the overhead introduced by the synchronization of threads is hardly counterbalanced in the computation and, therefore, the use of more threads speed-ups the construction only by about two percents if four insertion threads are used (see right graph in *Figure 9.18*). For different number of threads, results are worse. Let us point out that the total required time shows a logarithmical trend. This is caused by the fact that the number of local simplices grows with the number of input points faster than the number of shared simplices and, therefore, the efficiency increases. That is also the reason why the difference between times achieved by algorithms with one and four insertion threads is for a four millions data set lower than for smaller data sets.

If we let the interface construct the primary triangulation, as it was proposed in Section 8.4, it would bring additional speed-up. Unfortunately, this improvement is not implemented at present. According to our preliminary experiments, however, it seems that the speed-up might reach the value 1.4, which could partially eliminate the problem with scalability.

No matter what we do, the necessity to duplicate the computational effort stands for a serious bottleneck and, therefore, we cannot expect that the operation flow method would offer also big efficiency besides the possibility to process large data sets. As the extension of the operation flow into E^3 is quite difficult due to the big number of cases that must be considered and the results from E^2 version are not definitely positive, we decided not to use the operation flow, and prefer the data flow, for E^3 case.

9.9 The Results of Data Flow

We used the data flow for the parallel construction of Delaunay triangulation in E^3. Keeping the problem with scalability of the operation flow in mind, we decided to exploit the low level socket API for the communication in the data flow (unlike the slow DCOM that was used in our implementation of the operation flow) because a larger amount of communication than in the operation flow has to be expected. The data flow also requires transmitting of more data. *Figure 9.19* shows the dependence of the total time spent for the sending of data via the socket API on the size of this data. The total time consists of two contributions. First, it is an overhead introduced by the socket API. Next, it is the time required for the transfer of data itself. The measurement was done on 10Mb Ethernet. It can be seen that for smaller sizes (up to about 256 bytes), the socket API consumes larger time than is the transferring time. Let us note that if we consider also an overhead introduced by the VSM routines or 100Mb Ethernet is available, the size of data for which the overhead time is still larger than the transfer time is larger. Therefore, the algorithm runs faster if more simplices are retrieved in one communication, even if only some of them will be accessed. According to our experiments, if a simplex is to be accessed in the subdivision or the legalization phase, an adjacent simplex will be also accessed with the probability of 40% in a near future. If the confirmation of changes is performed once per insertion of 10 points, the probability increases to 50%. As the probability is quite low, the ideal number of simplices to be retrieved in one communication is also low. We found out that to retrieve a simplex with simplices in its neighborhood of three levels of recursion (i.e., neighbors, neighbors of neighbors and neighbors of neighbors of neighbors) is optimal.



a) up to 512 Bytes

b) up to 32K

Figure 9.19: *The total time required for the sending of data via the socket API in the dependence on data size, Intel Celeron 533 MHz, 10Mb Ethernet.*

Despite this improvement, the first experiments, when we used the DAG structure to speed-up the location, gave desperate results. The processing on a uniform data set with 25 000 points on two computers interconnected via 100 Mb Ethernet consumed about 6 minutes. Although the same amount of points was assigned to both processors, we found out that the first processor might finish up to one minute earlier than the second one. We tried, therefore, to incorporate dynamic load balancing: when the processor is idle, it sends a request to its counterpart and the contacted processor sends some of its points in a response. As the insertion of these points requires access to remote simplices, the communication effort grows. While the dynamic load balancing sometimes helps and the algorithm runs faster, sometimes this growth is killing – we needed even about 11 minutes to handle the considered data set. Therefore, we decided not to resent points.

Our further investigation showed that the imbalance is caused by the different number of accesses to remote simplices stored in non-leaves levels of the DAG structure. Since the time when the master processor sent the full identification of the auxiliary simplex, it, typically, has inserted several points into the triangulation, i.e., several levels in the DAG have been constructed before it receives the first retrieval request from the second processor. Therefore, while top levels of the DAG are local for one processor, for another processor these levels are remote and must be retrieved. We tried to construct sequentially some primary triangulation and sent it with its appropriate DAG structure to every processor in order to avoid remote accesses to top levels of the structure. The performance of the algorithm was enhanced.

Furthermore, we investigated the influence of various insertion strategies on the number of remote simplices that must be accessed. First, we tried to insert points in a random order into the triangulation. Next, we tried to insert points cell by cell, i.e., all points lying in a cell of the grid used for Mueller's algorithm for the subdivision of points are inserted and then the algorithm proceeds with the insertion of points from next (adjacent) cell. Finally, points are inserted so that in each step, a point from the current cell is inserted and then the algorithm continues with next cell, i.e., this cell becomes the current one. All these strategies were tested for the DAG location approach and the remembering stochastic walking technique. The results of this experiment are given in *Figure 9.20*. Although the test was done for small data set, it shows the tendency of the data flow: the best choice is to use the walking technique.

Insertion order	Location technique	Remote simplices [%]
random	DAG	45.5
grid cells, cell at once	DAG	53.0
grid cells, cell successively	DAG	33.7
random	walking	2.8
grid cells, cell once	walking	2.6
grid cells, cell successively	walking	3.7
8 points from corner cells, then random	walking	2.2

Figure 9.20: *The amount of remote simplices that have to be accessed to insert 1000 of uniformly distributed points by two processors into the triangulation in E^3.*

Using the walking approach, the workload is quite balanced and we were able to process the data set with 25 000 points in about 4 minutes. Let us note that points laying in corner cells of the grid were inserted first, the rest points were inserted in a random order (see Section 8.4). The walking technique offers also one important property: it does not need any additional memory. *Figure 9.21* compares the memory requirements (in nodes) of both location techniques for uniform data sets. As it can be seen, there is about 6 times more tetrahedra in the resulting Delaunay triangulation than points. If the DAG hierarchical structure is exploited, the algorithm consumes about 8 times more nodes than in the walking technique.

uniform data sets			real data sets		
N	DAG	walking	N	DAG	walking
1 000	47 907	6 330	2 905	147 380	19 010
10 000	522 015	66 483	9 199	493 611	62 403
100 000	5 355 671	672 138	22 625	1 281 827	156 155
200 000	10 732 525	1 346 669	98 869	4 995 099	649 403
240 000	12 889 278	1 616 453	213 373	19 732 714	1 517 812

Figure 9.20: *The amount of nodes constructed by the DAG and the walking location techniques.*

Figure 9.21 shows the total time required for the processing of uniform data sets up to one million of points on 8x HP Workstation xw3100 (using the walking technique). As it can be seen, the approach is not scalable. We have identified several reasons for such a behavior.

First, when a processor wants to get an exclusive access to a simplex that is already locked for another processor, the deadlock detection must be performed. When the number of used processors increases, the number of waiting processors, naturally, grows. This means that the number of processors that must be contacted in order to determine whether the processor is allowed to start the waiting also grows. The total time required for the deadlock detection, therefore, is linearly proportional on the number of processors and this, indeed, harms the efficiency of algorithm.

Next, when a remote simplex and simplices in its neighborhood are to be retrieved, the number of owners of these simplices, typically, grows with the increasing number of processors. This means that the total time consumed by the algorithm in order to retrieve remote simplices grows as more and more processors are used.

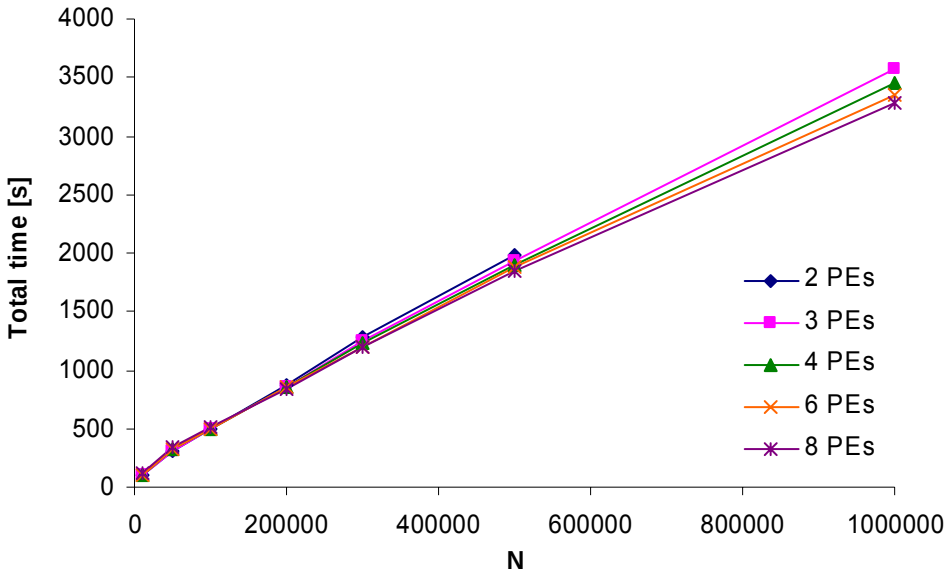


Figure 9.21: *The total time required for the construction of the Delaunay triangulation in E^3 of uniform data sets on 8x HP Workstation xw3100 – the walking technique was exploited.*

Last, however, the most serious reason is that the retrieval of remote simplices is too expensive to be counterbalanced by operations with local simplices. According to our experiments, any communication requires at least 7.5 ms, however, the local insertion of a point (i.e., no remote simplex is necessary) takes about 0.2 ms only. This means that at least 55 points has to be inserted locally to counterbalance one communication. We have found out further that a processor accesses about 75 different simplices during the insertion of a point. As it was already presented in Section 8, if the Delaunay triangulation is distributed over two processors, there is about $\sqrt[3]{T^2}$ remote simplices where T is the total number of tetrahedra. It is quite clear that this amount increases with the growing number of processors (e.g., it is $2 \cdot \sqrt[3]{T^2}$ for 4 PEs). Therefore, a simplex is the remote one at least with the probability $p_1 = 1 / \sqrt[3]{T^2}$. A processor accesses a remote simplex during the insertion of a point with the probability $p_2 = 75 \cdot p_1$, which means that it inserts $1 / p_2$ points locally per one insertion when it has to communicate in order to retrieve a remote simplex. Let us consider the construction of the Delaunay triangulation of a data set with one millions points using two processors. As the triangulation contains about 6 millions of tetrahedra (see *Figure 9.20*), the expected number of remote simplices is 185. Therefore, the processor accesses a remote simplex during the insertion of a point with the probability $p_2 = 0.2\%$. By a simple calculation, we get that the processor has to communicate in order to retrieve a remote simplex once per 500 local insertions. On the first sight it does not seem as a bad result, however, we have to point out that the operation with a remote simplex involves at least three communications (one is required for the retrieval of the simplex for read-only purpose, one for write purpose and one for its update). Therefore, the processor communicates approximately once per 150 local insertions, which is only about three times larger than the minimal number of local insertion needed to counterbalance the communication effort (i.e., 55). Without any doubt, if more processors are used or a smaller data set is processed, this value decreases. This means that the communication effort is never counterbalanced in the computation and that is why the algorithm is not scalable. It is very unfortunate that this problem had never crossed our mind until the solution was implemented and tested.

Let us consider several possible improvements to the proposed data flow approach. First what can be done is to start several insertion threads on every processor. When a thread requires to communicate, run of another thread is released and it proceeds with the insertion of a new point. We already used this strategy in the operation flow, however, without any significant improvement and, therefore, there is no reason to believe that it would bring speed-up of the data flow approach. As simplices are locked, this solution might even achieve an opposite effect because it increases the probability of deadlock, i.e., the number of cases when a processor has to undo its work.

Whenever a processor completes the insertion of a point (or insertion of few points) and performs the confirmation of changes, the data of a modified remote simplex is sent to the appropriate owner of this simplex. Afterwards, if the processor needs to access the same simplex, it needs to communicate to check whether its cached version is the current one. Very often, especially in the early phase of the insertion, a remote simplex is checked several times with the same result: the caller has the current version. This led us to an idea to let simplices to migrate, i.e., whenever a processor gets an exclusive access to the simplex, it becomes its owner and performs any change of this simplex without a necessity to communicate as long as its ownership lasts. It is clear that when a processor wants to access a remote simplex, it may need to contact several processors before it reaches the current owner. Moreover, for each simplex, this solution would require to store its current address. We tried, therefore, another approach. Let us define a *page* as a continuous block of memory. Under MS Windows platforms, this block is usually 4096 bytes long. A simplex is stored in just one page; there are about 45 simplices per one page (one simplex needs about 90 bytes). When a remote simplex is to be accessed, the whole page is retrieved. If a simplex is going to be modified, the ownership of its appropriate page is changed, i.e., the page migrates.

Our preliminary experiments with pages, however, show that this approach is even slower than the original one. The reason is that it is impossible to store adjacent simplices on the same page and, therefore, a processor has to get an exclusive access to many pages, i.e., to change their ownership, in order to insert a point. This leads to the rapid increase of deadlocks in the computation, i.e., cases when a processor has to undo its work are frequent. A cure to this problem could be to let several processors to modify the same page and to decide whether a collision has occurred (i.e., one byte in a page has been modified by more processors) during the confirmation of performed changes. There is no doubt that the implementation of this strategy is quite complex and there is no insurance that it would bring the improvement.

9.10 Comparisons and Summary

We have described several parallel methods for the construction of the Delaunay triangulation based on randomized incremental insertion with local improvements. We focused on two different goals. The first one was to speed-up the computation of the Delaunay triangulation and, the second one, to process large data sets. In order to achieve the first goal, we designed several methods for multiprocessors with shared memory and limited number of processors, typically 2 or 4. The implementation of the developed methods is available as a part of the Modular Visualization Environment [MVE] – a software package developed at our institute.

According to our experiments, we found the methods almost insensitive to the point distribution of input data set. They differ in the complexity of implementation, scalability and efficiency (considered for the optimal number of processors). Batch method is limited to few processors only and usable in E^2 only, however, it is very efficient and simple to be implemented. Pessimistic method is the simplest method to be implemented but inefficient. Optimistic method needs more complicated implementation but this method is efficient and

scalable. Burglary method is also complicated, however, very efficient in E^2 (not that efficient in E^3) and scalable. Circum-circle method is very efficient in E^2 when 2 threads are used.

The simplicity, scalability and average efficiency of all methods are summarized in *Figure 9.22* – higher number means that a method is more simple, scalable or efficient. The grades were assigned by our best personal opinion and experience, especially in the case of the simplicity criterion. According to these results, it is quite clear that each method has its pros and cons and, therefore, a researcher should use a method that is the most suitable for their purpose. If a general solution is required, optimistic method seems to be the best choice.

Method	E^2			E^3		
	Simplicity	Scalability	Efficiency	Simplicity	Scalability	Efficiency
Batch	2	1	3	N/A	N/A	N/A
Pessimistic	3	2	0	3	0	0
Optimistic	0	3	2	0	3	2
Burglary	1	3	3	1	3	1
Circum-circle	2	0	1	N/A	N/A	N/A

Figure 9.22: *Comparison of all parallel methods (higher value means higher evaluation).*

The comparison of the best speed-ups achieved by optimistic method and speed-up achieved by other existing parallel algorithms is given in *Figure 9.23*. As far as we know, there are no published results of any parallel algorithm based on the incremental insertion principle. Therefore, we choose the Hardwick's algorithm [Har97], DeWall and InCode algorithms [Cig93] for the comparison. These algorithms were described in Section 4. All our methods ran at Dell Power Edge 7150, the Hardwick's algorithm at SGI Power Challenge, DeWall and InCode algorithms ran at nCUBE 2 system model 6410. As the experiments ran at different machine and speed-up evaluation differs, it is impossible to say without any doubt whether one method is more or less efficient than another one. Let us, however, claim that the results achieved by our optimistic method are comparable.

PEs	2D		3D		
	OPT	Har–Ble	OPT	DeWall	InCode
2	1.75	1.82	1.66	1.70	1.79
4	3.25	3.33	3.67	2.46	3.11
8	5.71[1]	5.88	4.42[2]	3.05	5.31

[1] From previous results (at present we have no access to multiprocessor with 8 PEs and, therefore, we are unable to repeat the experiment), expected similar
[2] From previous results (at present we have no access to multiprocessor with 8 PEs and, therefore, we are unable to repeat the experiment), expected higher

Figure 9.23: *Comparison of best speed-ups achieved by optimistic method (OPT), Hardwick's algorithm [Har97] (Har-Ble), DeWall and Incode algorithms [Cig93] for uniform data sets.*

For larger data sets, two different parallel algorithms suitable for clusters of workstations were implemented. Although they are able to process data sets of theoretically unlimited sizes, in practice, we are limited to data sets with several millions of points because the processing of larger data sets would consume too much time. The main reason of huge time consumption is the inefficiency of algorithms. There are several options how to improve the efficiency and scalability but these require further research and we are not quite sure whether they would bring the desired improvement. According to the achieved results, we decided not

invest any development effort for the implementation of the mixed flow proposed in Section 8 because, although this approach requires less amount of communication than the operation and data flow approaches, the mixed flow still needs to communicate too often to achieve scalable results. An algorithm that does not need to communicate during the insertion is, therefore, probably the only choice if a good efficiency and scalability is required besides the possibility to handle larger data sets. We propose such an algorithm in Section 11.

Although the current solution is inefficient, the possibility to process larger data sets itself is still a good result of our research. In the next section, we show an application exploiting our data flow approach.

10 Surface Reconstruction

Surface reconstruction is a common problem in computer graphics. Given a set of points sampled from some surface, i.e., points in E^3, a triangle mesh interpolating or approximating these points has to be obtained. The importance of the task has grown in the last years as the scanning devices became cheaper and more applicable. Many algorithms based on various approaches, e.g., warping, incremental construction and spatial subdivision, have been developed for this problem. It is not a goal of this thesis to give a detailed description of the surface reconstruction problem. A survey of existing approaches and their comparison can be found in the PhD thesis by Varnuška [Var05].

Let us focus on spatial subdivision methods. The basic property of the methods based on spatial subdivision is that the boundary hull (convex hull, box around points, etc.) of the point input set is divided into independent areas. A typical example is the division by a regular grid, adaptive by an octree or an tetrahedronization. The Delaunay triangulation is very popular because, as it can be proved, the surface reconstructed from the given points set is the sub-graph of the $DT(S)$ and, therefore, it is much easier to find the proper surface using the Delaunay triangulation than in the case that another spatial subdivision is used. One of the first algorithms exploiting the Delaunay triangulation was proposed by Boissonat [Boi84]. It removes successively outer tetrahedra from the triangulation until all points lie on the surface. The rest of tetrahedra then compose the volume of the object. Amenta et al. [Ame02] proposed the CRUST algorithm. Again, the first step is the computation of the Delaunay triangulation. By the dualization, the Voronoi diagram of the point set is obtained and some specific information from the Voronoi diagram is used for the selection of triangles supposed to be on the surface from the Delaunay triangulation. The CRUST produces nice results for sufficiently sampled data of the closed smooth objects. If an object contains sharp edges, it is undersampled or the sampling introduced some noise, then some artifacts, e.g., holes or overlapping triangles, may appear in the reconstructed surface. Varnuška [Var05] suggests several improvements to the CRUST algorithm to handle such problematic data sets.

Let us describe the algorithm by Varnuška. First, if it is necessary, the amount of noise present in the point set is reduced using a technique based on the approximation of the normal vectors in every point and on the estimation of surface location. Then, the Delaunay triangulation in E^3 is computed. For each point, tetrahedra that are incident with this point are found, the appropriate Voronoi cell is constructed (the centers of circum-spheres of these tetrahedra form the Voronoi vertices of the cell) and the normal vector and so-called poles for the given point are computed from the cell. Afterwards, each tetrahedron face is tested using the information about normals and poles of its vertices whether it could be on the surface or not. The surface formed by the selected faces may not be a manifold; it may contain fans of overlapping triangle, holes etc. Therefore, some additional steps are required to get the final surface. In these steps, incorrectly reconstructed places are detected and repaired. Details can be found in the thesis [Var05].

A typical real data set, e.g., points scanned on a statue [Mich], contains several millions of points. A surface reconstruction of such a data set demands a lot of memory. This is definitely true for algorithms based on the spatial subdivision and especially for the CRUST. For each input point, at least 124B are needed in order to store all required information such as coordinates of this point, coordinates of its poles and the normal vector. A tetrahedron in the Delaunay tetrahedronization takes 54B to store its vertices, pointers to adjacent tetrahedra, circum-sphere, volume, etc. Finally, 62B are consumed to store triangle vertices, pointers to adjacent triangles, normal and information for surface extraction. According to our

experiments (see Section 9), there are about 6 times more tetrahedra than points in the triangulation, which means that the final surface might consists of up to 12 times more triangles (usually two faces of tetrahedron lies on surface) . The processing of a data set with one million of points, therefore, requires at least 850 MB of memory. If the double-precision arithmetic is preferred, these memory requirements increase. Let us also note that if a hierarchical structure, e.g., the DAG, is exploited to speed-up the construction of the Delaunay triangulation, the CRUST consumes at least 3 times larger amount of memory. As the current sequential implementation by Varnuška trades memory for speed, it is able to handle data sets only up to 250 000 points on a computer with 1GB of memory. In practice, it is, however, necessary to deal with much larger data sets. *Figure 10.1* shows a few examples of large real data sets.

a) Lucy, size: 14M points, estimated memory requirement: approx.10GB

b) St. Matthew, size: 186M points, estimated memory requirement: approx. 150GB

c) Barbuto, size: 350M points, estimated memory requirement: approx. 300GB

Figure 10.1: *Examples of large real data sets from the Stanford repository [Mich, Sta99].*

As an application running under MS Windows operating system on 32-bits computer can use at most 2 GB, typical real data sets cannot be processed in an unmodified version in one-step on one computer. We could use some kind of compressed data representation or out-of-core technique. Such an approach, however, would slow down the processing significantly. Therefore, first possibility how to deal with large data sets (i.e., one million or more points) is to reduce the number of points and reconstruct the surface from this smaller set. This means that the result is only an approximation. Despite this, some reduction technique is often used. An interesting idea can be found in paper by Carr et al. [Car03]. Authors use radial basis functions on clusters of points to construct an analytical patch that is then sampled with lower frequency than was the original one to get new points to be used in the reconstruction process. As radial basis functions generate a smooth patch, the approach performs also the reduction of noise. On the other hand, the reconstructed surface does not interpolate the original input points, which is not always allowed (e.g., for the manufacturing).

Varnuška proposed another approach. For each point, it finds such a point that is nearest to the given one (the so-called nearest neighbor) and merges these two points together. The

position of the merged point can be either the same as is the position of one of these points or somewhere, usually in the middle, on the line segment connecting these points. In the first case, the surface interpolates the selected group of points. *Figure 10.2* shows an example of the merging of two points. Finding of pairs of points can be done with expected quadratic complexity by a brute-force. Better strategy is to exploit the grid bucketing technique (used also for the distribution of points over processors – see Section 6), which offers linear complexity in the expected case. Let us note that the merging process has to be repeated until the number of points is lower than some defined maximum. As it is, indeed, necessary to store coordinates of all points, this reduction approach is useful up to about 130 millions of points. If the information about which two points were merged (including original coordinates of these points) is retained, it is possible to insert removed points back to the surface using vertex split technique [Hop96] and so improve the result. However, it is possible for smaller triangulations only (up to four millions).

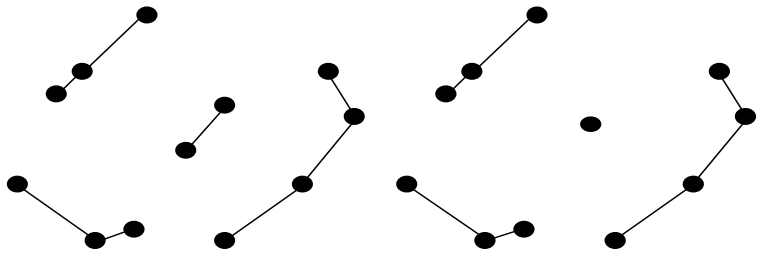


Figure 10.2: *Merging of a pair of two nearest points. Pairs of nearest points are connected by lines.*

If the object to be reconstructed contains sharp edges or the ratio of the original size and the reduced size is big, this solution may produce an incorrect surface (because of the accumulation of error introduced in the iterative decimation of points). If the reconstructed surface should be accurate, e.g., because it is used in the manufacturing process, it is a serious problem and, therefore, another approach is requested.

Dey et al. [Dey01] presented an approach based on Divide & Conquer strategy. It subdivides the input dataset into several smaller groups whose min-max boxes overlap, reconstructs the surface of points in each group and then merges surfaces together. Using this solution, authors were able to handle a data set with 3.5 millions of points. There are two problems. First, the merging is not an easy task. Next, the reconstructed surface may suffer from artifacts, especially for non-uniformly sampled points or noisy data. *Figure 10.3* shows an example of incorrect reconstruction in E^2. Input points were split into four non-disjunctive groups (shaded areas). After the merging of partial surfaces, an obviously wrong edge appears in the result.

Last option is to exploit a couple of computers for the data storage and processing. We focused on this option because it allows us to process correctly data sets of theoretically unlimited size and to speed up the reconstruction by a distributed computing. The CRUST algorithm by Varnuška is written in Borland Delphi 7.0 and it contains several tens thousands lines of source code. In order to simplify the parallelization of the program, we exploit the VSM system proposed in Section 8 that is based on the data flow principle. As the Delphi programming language, however, does not offer the indexer operator, the parallelization of access to arrays distributed over processors needs some effort. Therefore, not every data structure consuming a lot of memory is distributed at present (we plan to do in a near future), which limits the size of largest data set to 6 millions of points. Currently, we managed to distribute major data structures, such as the Delaunay triangulation or output triangular surface. As the coordinates of points are accessed frequently from many routines in our

implementation, we decided to duplicate them on every processor. It is not too limiting and it increases significantly speed of the processing.

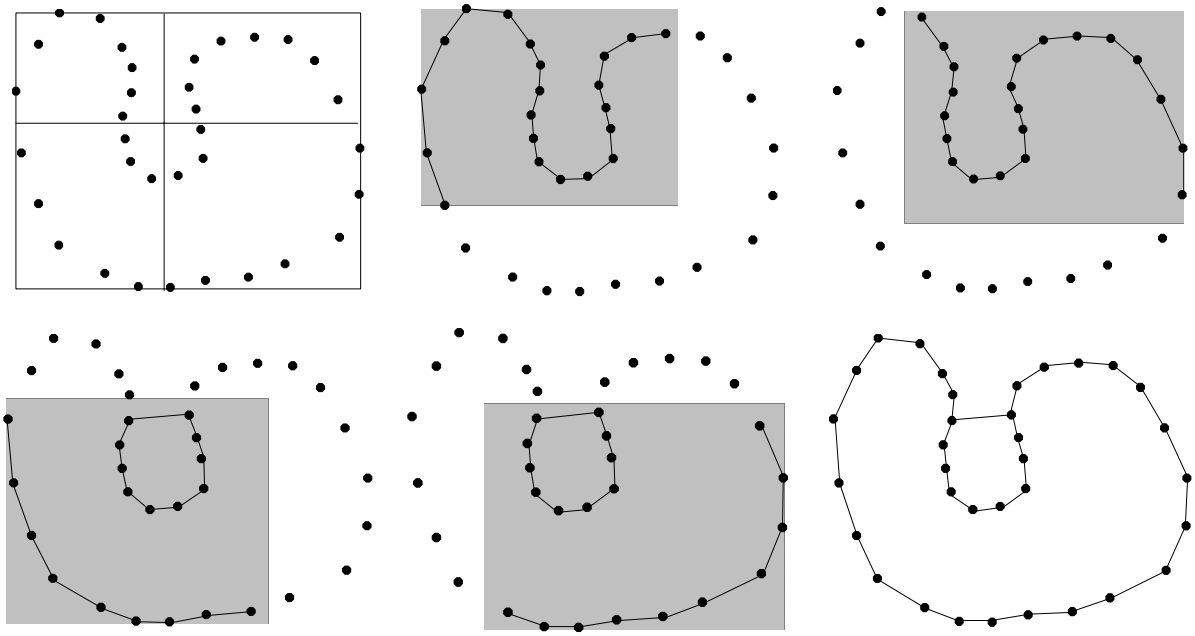


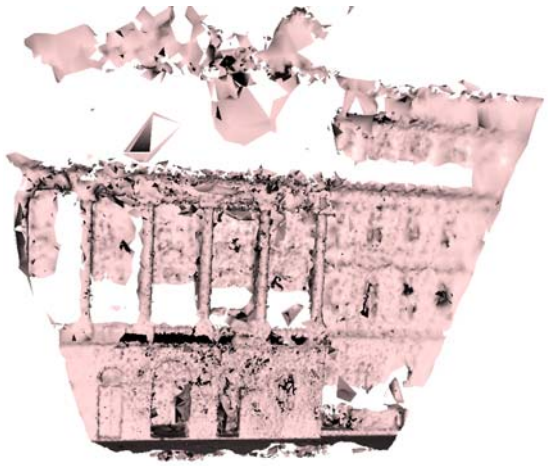
Figure 10.3: *An example of reconstruction per partes in E^2 producing an incorrect surface.*

In our current version, only the computation of DT is parallelized (results are presented in Section 9), all remaining parts of reconstruction run sequentially. Therefore, it is necessary to transmit huge amount of data, which is time consuming. As the computation of poles and normal vectors, extraction of triangles lying on the surface and the correction of wrong places including holes filling are local operations, proper parallelization would bring a significant improvement. We intend to do it in future work. The largest dataset that we have processed had 1.4M points and it was a part of the original Lucy model dataset. Its reconstruction ran on 4xP4 interconnected via 100Mb Ethernet (see Section 9). About 5GB of memory was consumed and the computation took about 6 hours. *Figure 10.4* shows results obtained by the decimation and by the distributed computing. As it can be seen, if the decimation technique is used, some details (e.g., brow, lips, hair and button on clothes) are lost.

a) point based rendering *b) decimation technique* *c) distributed computing*

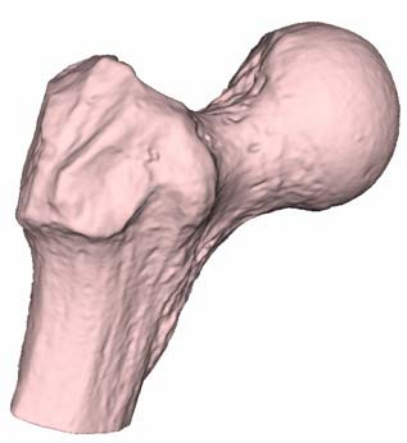
Figure 10.4: *Surface reconstruction of a part of Lucy model in comparison with the result obtained by point based rendering technique [Mich].*

Other examples of reconstructed surfaces of real data sets are presented in *Figure 10.5*. Let us note that although the time spent by the construction of the Delaunay triangulation grows almost linearly with the size of data set, the time consumed by the remaining parts of the surface reconstruction process depends also on the complexity of object (i.e., whether there is some noise, sharp edges, outliers, etc). Therefore, the reconstruction of a smaller data set may take more time than the reconstruction of a larger data set – compare the time needed for the reconstruction of bone and club models. Let us remind that except for the construction of the Delaunay triangulation, the reconstruction runs on one computer only and, therefore, it has to communicate very often, which explains why the total time needed for the reconstruction is so larger (i.e., several hours).



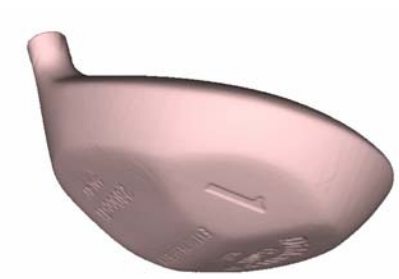
Points:	65 894
Tetrahedra:	415 312
Triangles:	117 742
DT time:	2 min 6 sec
Total time:	4 min 27 sec

a) facade of house



Points:	137 062
Tetrahedra:	935 304
Triangles:	274 118
DT time:	8 min 33 sec
Total time:	1 hr 38 min

b) bone



Points:	209 779
Tetrahedra:	1 322 057
Triangles:	419 500
DT time:	9 min 5 sec
Total time:	62 min 31 sec

c) club

Points:	437 645
Tetrahedra:	3 041 391
Triangles:	857 930
DT time:	22 min 40 sec
Total time:	2 hr 2 min

d) dragon

Figure 10.5: *Surface reconstruction of real data sets. Models adopted from [Var05].*

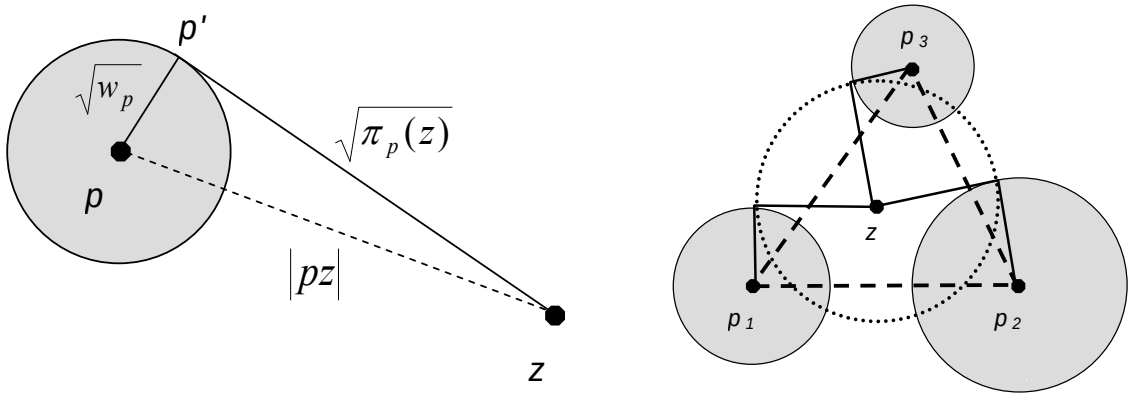
11 Possible Extensions

In Section 5, we described an incremental insertion algorithm for the construction of the Delaunay triangulation in E^2 and E^3 that was used as a base for our parallelization purposes. Although the Delaunay triangulation is directly applicable to many various problems, e.g., the surface reconstruction (see Section 10), there exist plenty of applications that require to incorporate constraints given in the form of prescribed faces into the triangulation, to use non-Euclidian metrics or weights of points during the computation, to delete, successively, some unimportant points from the triangulation. Some applications also call for the construction of the Delaunay triangulation in higher dimensions. In this section, we discuss possibilities of extension of proposed parallel algorithms to fulfill these requirements.

11.1 Weighted Triangulations and non-Euclidian Distances

Regular triangulations [Ede92, Fac95] are a generalization of Delaunay triangulations offering an extra degree of freedom by introducing weights for points. Given a point set S in E^d, a real valued weight w_p is assigned to every point p from the set. Let us note that the weighted point can be interpreted as a sphere with center p and radius $\sqrt{w_p}$. For each weighted point p, we define so-called *power distance* from a not weighted point $z \in E^d$ to the point p as $\pi_p(z) = |pz|^2 - w_p$, where $|pz|$ is Euclidian distance between points p and z. The geometrical meaning of the power distance is shown in *Figure 11.1a*.

For any simplex, it is possible to find a point z such that the power distances from this point to every point of the simplex are the same – see *Figure 11.1b*. A weight equal to the square of the computed value of power distance is assigned to the point z. The weighted point z is called the *orthogonal center* of the simplex and the sphere with radius $\sqrt{w_z}$ centered at z is the *orthosphere* of the simplex. Let us note that if the weights of points of this simplex are zero, then the orthosphere and the circum-sphere of the simplex are identical.



a) the power distance from z to p *b) the orthogonal center of simplex p_1,p_2,p_3*

Figure 11.1: *The geometrical meaning of power distance and orthogonal center in E^2.*

A triangulation is regular only if all simplices are locally regular. A simplex p_1, p_2, p_3 or, in the case of E^3, p_1, p_2, p_3, p_4 is locally regular if the power distance from any point $q \in S - \{p_1, p_2, p_3, p_4\}$ to the orthogonal center of the simplex is larger than the weight w_q assigned to this point q, i.e., $\pi_z(q) > w_q$. It is clear that any method described in Section 3 for the construction of the Delaunay triangulation can be used also for the construction of regular triangulation.

All that is needed is it to supersede the Delaunay empty circum-sphere condition by the condition of regularity. Let us focus on the method of incremental insertion with local transformation – see Section 5. Points are successively inserted into the existing regular triangulation and, as in the Delaunay triangulation, if a set of adjacent simplices violates the condition of regularity, local transformations have to be applied. *Figure 11.2* shows an example of local transformation in E^2. The edge shared by two adjacent triangles is invalid, i.e., the triangles are not regular, and, therefore, it is swapped.

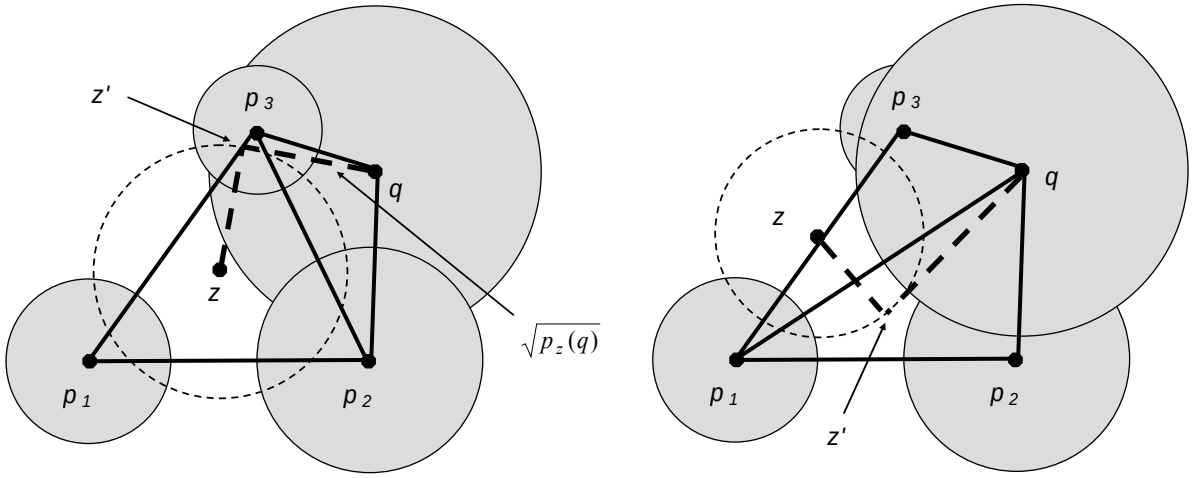


Figure 11.2: *Local transformations in E^2. The edge is swapped.*

If the geometrical meaning of power distance and orthogonal center is taken into account, we can reformulate the condition of regularity as follows. A simplex is regular, if for any point q from S (except for points in the vertices of the simplex) the point z' of contact of tangent to the orthosphere of the given simplex going through the point q does not lie inside the sphere with radius $\sqrt{w_q}$ centered at the point q – see *Figure 11.2*. This means that to decide whether an edge (or a face in E^3) is invalid, i.e., whether a local transformation must be applied or not, we need to test the mutual position of some sphere and point. It is exactly the same test as the one used in the Delaunay triangulation, only spheres and points to be checked are different.

As the test of empty circum-sphere used in the Delaunay triangulation and the test of regularity are, actually, the same, any parallel algorithm proposed in Sections 7 and 8 can be used for the construction of regular triangulation without a necessity to modify it (indeed, the Delaunay test has to be superseded by the test of regularity).

Vigo et al. [Vig00] discussed a generalization of the Delaunay triangulation exploiting a non-Euclidian anisotropically deformed space in order to create triangulation of points lying on a terrain surface where the shape of triangles depends on the curvature of the surface in each direction. In the ideal case, the algorithm produces a triangulation having edges aligned according to curvature directions and their length proportional to curvature (i.e., long edges in plane regions, short edges in curved regions). Authors show that either the Delaunay empty circum-circle criterion has to be replaced by a more general test of empty circum-ellipse or the input points have to be transformed according to the deformation of space and then the usual construction using Euclidian distances may be performed. No matter which solution is used, in both cases, any parallel algorithm proposed in Sections 7 and 8 works correctly and does not need any modification. Let us note that the power distance used in the regular triangulation is also a kind of non-Euclidian metrics.

11.2 Deletion of Points

In some applications (e.g., for the decimation purpose), it is necessary to delete some points from the given Delaunay triangulation in such a manner that the new triangulation is also the Delaunay one. Algorithms based on the method of incremental insertion may easily be turned into fully dynamic algorithms supporting insertion or deletion of a point on demand. The deletion of a point p can be considered as an operation reverse to the insertion of this point. A simple algorithm performs successive application of local transformations until the degree of the point p is three or, in E^3, four. Afterwards, the point can be easily removed from the triangulation and another sequence of local transformations is applied to restore the Delaunay property – see *Figure 11.3*.

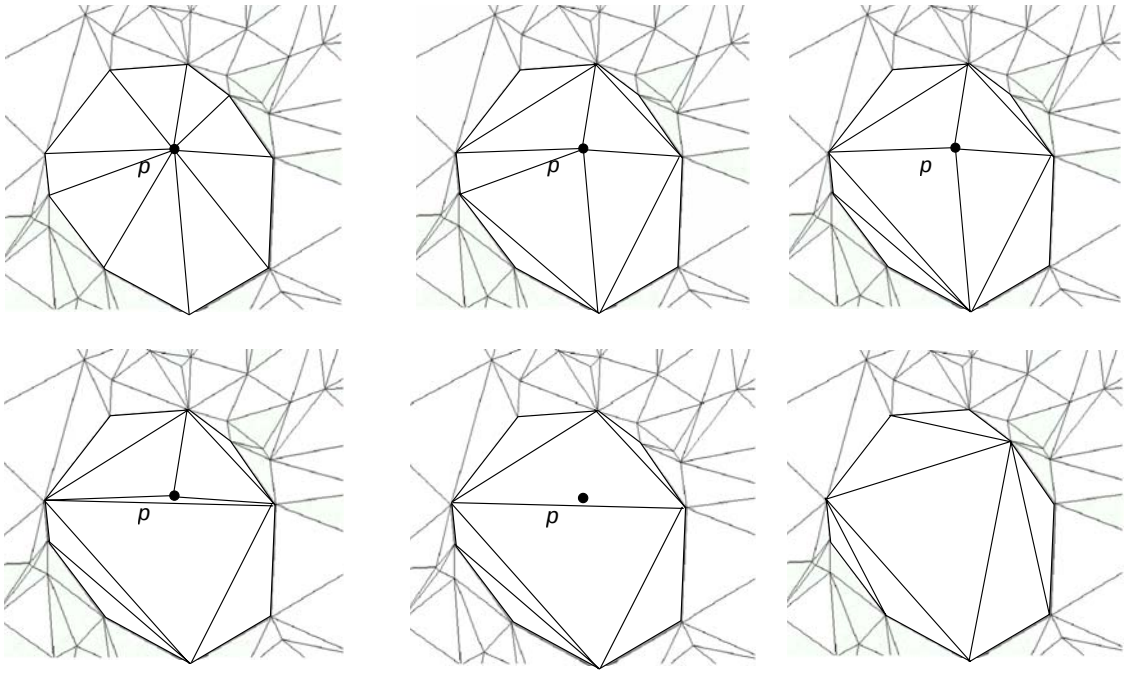
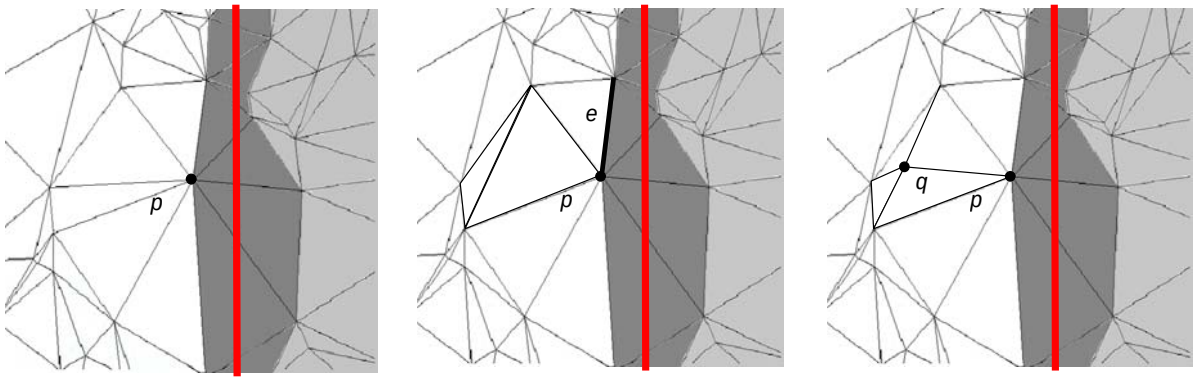


Figure 11.3: *The deletion of the point p from the Delaunay triangulation in E^2.*

The advantage of the algorithm is its simplicity and robustness. The DAG structure or related hierarchical structures (e.g., [Dev98]) used to speed up the location is supported without any significant modification: the deletion, successively, introduces new nodes that are added to the structure. On the other hand, the implementation of deletion in E^3 is quite complex. According to our opinion, the first stage of the algorithm, moreover, might not converge.

In batch and pessimistic approaches, any modification of the triangulation is limited to one processor and, therefore, the generalization of algorithms to incorporate the deletion operation seems to be straightforward. In the operation flow, if the deletion requires to access local simplices only, there is also no problem. If a processor needs to operate with a shared simplex, the operation is performed in a synchronized way via the interface. According to our experiments (see Section 9), it is useful to start several working threads on each processor and whenever a thread is going to wait for the completion of synchronized operation, another thread is released and it proceeds with its work. Although it is not very probable, the thread might undo the transformations already done by the waiting – see *Figure 11.4*. The solution to this problem is simple: not to release any thread in the case of deletion. It is clear that the efficiency of the flow approach decreases with the growing number of synchronized deletions.

The mixed flow is just generalization of the operation flow and, therefore, it requires similar modifications to allow dynamic removal of vertices from the Delaunay triangulation.



a) the point p to be deleted, the degree of p is 9

b) before the synchronized flip of edge e, the degree of p is 8

c) another thread inserts the point q, the degree of p is 9

Figure 11.4: *The problem of deletion of the point p lying near the boundary of areas of two processors in the operation flow approach.*

A challenge is, however, the generalization of optimistic methods. If simplices are locked using a 'lock' parameter (see Section 7.5) and a transactional mechanism is exploited, as in the optimistic method and the data flow, there is no doubt that the work of processors is correctly synchronized and the parallelization is straightforward. In the burglary method and the circum-circle method, the deadlock is handled, however, by violated reuse of simplices locked for another thread. Therefore, after the waiting thread is released, it may found out that the triangulation has changed since it started to wait. It is exactly the same problem as it was in the operation flow. There is only one possible solution: repeat the process until it succeeds. If we expect a large number of deletion requests, the optimistic method is probably more suitable as it should achieve better speed-up.

The circum-circle method introduces also another small problem. A processor may operate with a triangle only if no point currently being inserted by other processors lies inside the circum-circle of this triangle. In the case of the deletion of the point p, the point does not lie inside a circum-circle because it belongs to the Delaunay triangulation. It, however, lies on circum-circles of triangles sharing the point p. Therefore, the test has to be changed: the processor is allowed to modify only such a triangle that any point currently being inserted or deleted by other processors lies outside the circum-circle of this triangle.

Let us discuss another approach for the deletion of a vertex from the Delaunay triangulation. It is based on the idea used in the Bowyer-Watson algorithm (see Section 3). Unlike the previous approach with local transformations, it is provably correct in any dimension. Usually, it runs also faster. On the other hand, the DAG is quite difficult to be updated.

In two dimensions, the deletion of the point p means that m triangles must be removed from the triangulation and $m-2$ new triangles must be created to fill the hole – see *Figure 11.5a, b, e*. Although m may be equal to the number of points in the triangulation, it is well known that the expected value of m is 6 without any assumption on the point distribution. In three dimensions, the situation is worse. First, the number of simplices required to fill the hole does not depend on the number of simplices that were removed. Next, the expected value of m is usually larger and it depends on the point distribution. Devillers

[Dev99] mentioned that for Poisson distribution it is about 27. Therefore, in the expected case the deletion in both dimensions influences only a local part of the triangulation. As the insertion of a point is also localized (see Section 7), we can expect that modified parallel algorithms will behave similarly to their unmodified versions, i.e., they should reach similar efficiency and scalability.

A very popular algorithm for the retriangulation of hole is based on successive cutting of ears of this hole. Let us describe this algorithm in E^2, its extension for E^3 is straightforward. Three topologically consecutive vertices q_i, q_j, q_k along the boundary of the hole form an ear if the line segment q_i, q_k its inside the hole and does not cross its boundary. If the triple of vertices q_i, q_j, q_k is an ear, then the triangle q_i, q_j, q_k is constructed and the vertex q_j is removed from the polygon describing the boundary of the hole. The algorithm continues cutting one ear in each step until the hole is filled. It is clear that the hole may be retriangulated, usually, in several ways. After the retriangulation, we have an arbitrary triangulation and, therefore, the newly created simplices must be tested against the Delaunay criterion and if the outcome of this test is negative, local transformations are performed to restore the Delaunay triangulation. Devillers [Dev99] proposed, however, an efficient algorithm (it requires $O(m \cdot \log m)$, where m is the number of points forming the cavity) based on lifting of points to higher dimension. It picks such ears that the retriangulation is the Delaunay one and, therefore, no local transformation is needed. An example of retriangulation of hole is shown in *Figure 11.5*.

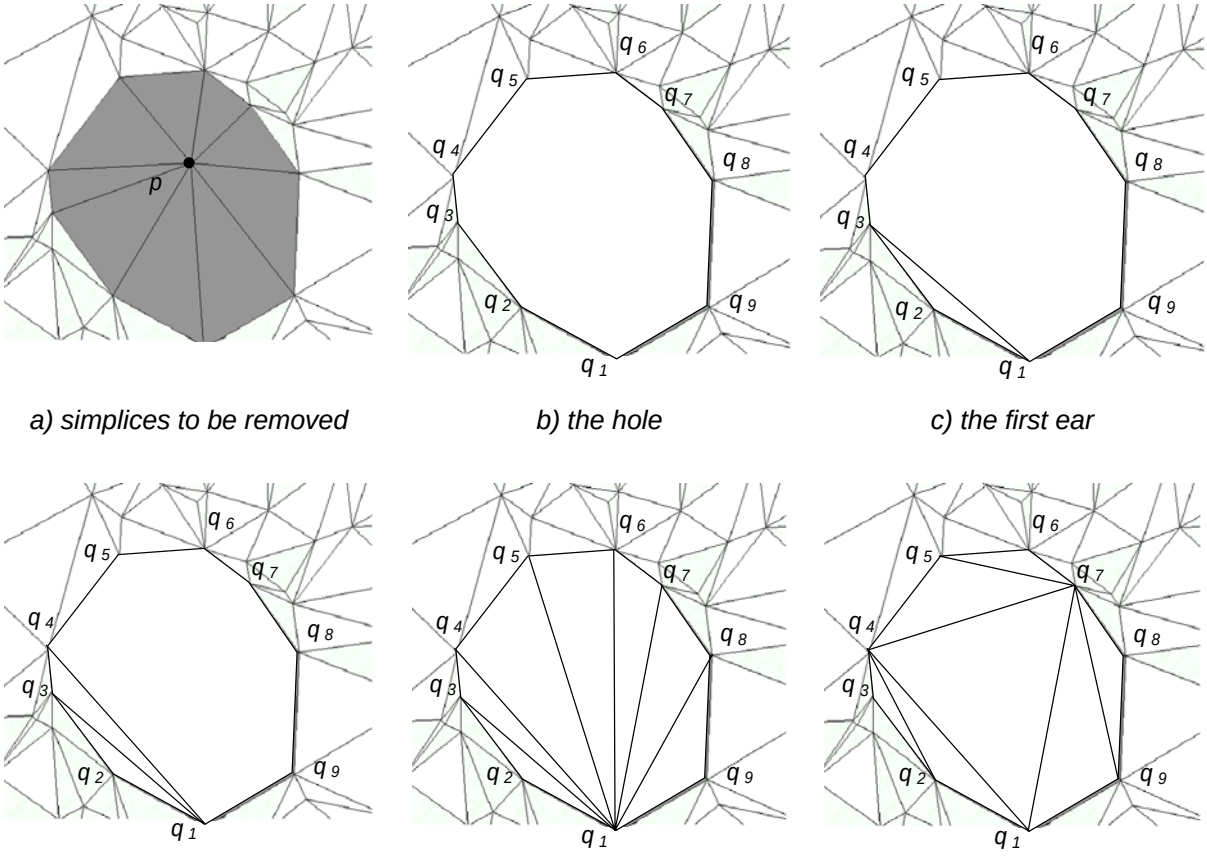


Figure 11.5: *The deletion of the point p from the Delaunay triangulation in E^2 using the hole retriangulation approach.*

Batch and pessimistic methods can be again generalized without any difficulty. For all other methods, however, the basic problem is the temporary existence of hole in the triangulation because while one processor works on the retriangulation of the hole, another processor might want to insert a point lying somewhere in this hole and, indeed, the operation of insertion of a point into a hole is not defined. The solution is as follows. First, simplices to be removed are detected. Next, new simplices filling the hole formed by detected simplices are constructed and the connectivity between them is found. In the third step, the patch is connected to the Delaunay triangulation. Finally, the hierarchical structure used for the location is updated or simplices are removed. The generalization of the algorithm is, now, straightforward.

Using the retriangulation strategy by Devillers, the deletion needs to operate only with simplices incident with the point p to be removed and, in order to update the connectivity, with their neighbors, too. It is possible to detect these simplices in advance. In the case of the operation flow or the mixed flow, they are simply sent as a part of operation to be performed. Again, it is necessary to disallow run of another thread until the deletion completes. The generalization of optimistic methods or the data flow is even simpler: these algorithms lock simplices and perform the operation. Once simplices have been locked, the deletion always succeeds and, therefore, neither the burglary nor the circum-circle method needs to repeat the request. Therefore, we can expect that a parallel algorithm using the ear cutting approach should be quite efficient.

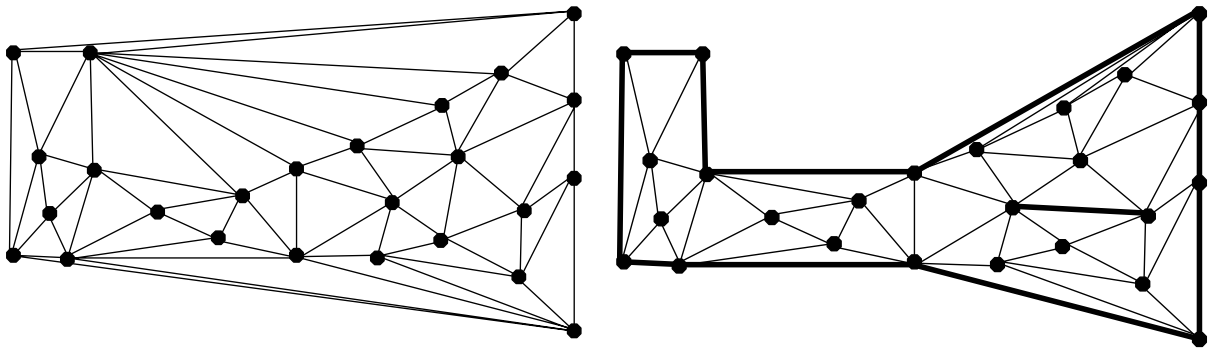
11.3 Higher Dimensions

Several, mainly theoretical, papers, e.g., [Wat81], deal with the problem of construction of the Delaunay triangulation of a set of points in an arbitrary dimension E^d. Algorithms based on the incremental construction or higher dimensional embedding are easily extensible but the implementation might be quite complex. Let us remind to the reader that, in the case of higher dimensional embedding, to construct the Delaunay triangulation in E^4, we need to compute the convex hull of the given points in E^5, which is quite difficult for understanding. Algorithms based on the incremental insertion are also extensible and they offer simplicity. As far as we know, there is, unfortunately, no simple generalization of local transformations to higher dimensions. Therefore, it is necessary to use cavity retriangulation strategy described in Section 3. All simplices containing the point to be inserted in their circum-spheres are located and removed from the triangulation. Afterwards, the cavity (hole) is retriangulated in such a manner to fulfill the Delaunay criterion.

The problem of insertion of a point into the Delaunay triangulation in E^d is similar to the problem of deletion of a vertex from the triangulation by the ear cutting technique that was described in the previous section. Therefore, let us claim that if the extension of any of proposed parallel algorithms into higher dimensions is required, it will require the same modifications as in the case of the generalization of this algorithm to incorporate the deletion. In addition, the issues about the efficiency and scalability are the same.

11.4 Constrained Delaunay Triangulation

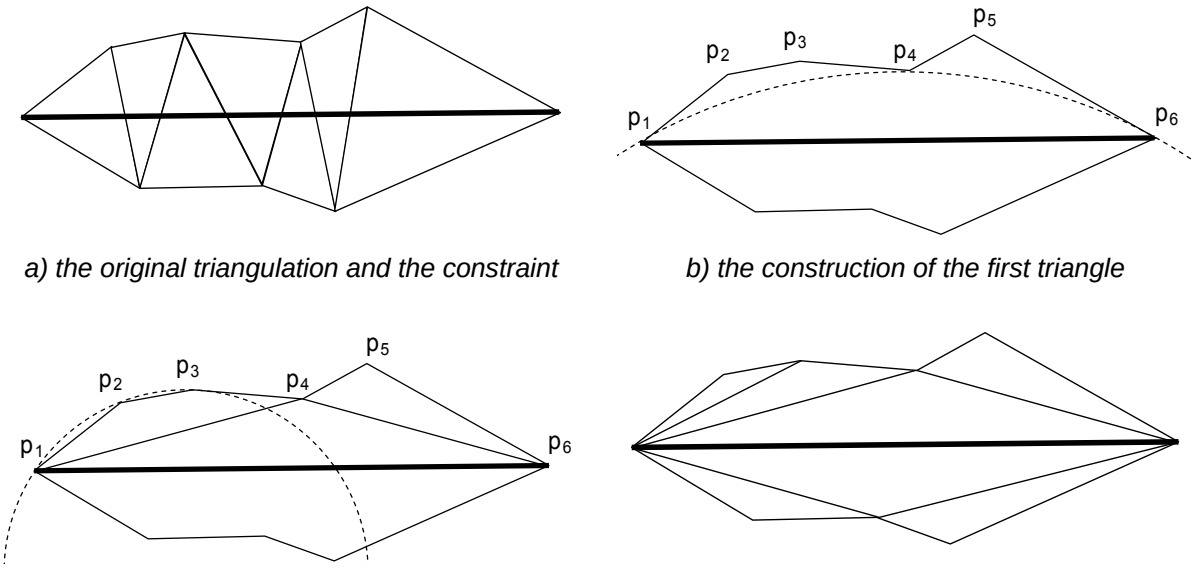
Constrained Delaunay triangulation is a generalization of Delaunay triangulation offering a possibility to incorporate some prescribed edges or faces (i.e., constraints) into the triangulation. Typically, these constraints are used either to express the shape of object whose sampled points are to be triangulated or to introduce some physical limitations. *Figure 11.6* compares the Delaunay triangulation and the CDT of the same input set in E^2. The prescribed edges are thick. Constrained Delaunay triangulation is used in many applications, e.g., numerical analysis and finite element methods (FEM), pattern recognition [Pra00, Xia02], etc.



a) DT *b) CDT*

Figure 11.6: *An example of the Delaunay triangulation (DT) and the Constrained Delaunay triangulation (CDT) in E^2. Prescribed edges are thick.*

From the point of view of the algorithm based on the incremental insertion with local transformations, a constraint is an edge or a face from the triangulation that cannot be flipped, i.e., this edge (or face) is always considered valid in the meaning of the Delaunay criterion. This means that the legalization stops on constraints. The best-known approach for the insertion of a constraint into the triangulation works as follows. First, all simplices crossed by this constraint are detected – see *Figure 11.7a*. These simplices are removed from the triangulation, which results in two adjacent holes separated just by the constraint. Then, both holes have to be retriangulated. For this purpose, the ear cutting algorithm (in its either original version or version by Devillers) that was presented in Section 11.2 can be used.



a) the original triangulation and the constraint *b) the construction of the first triangle*

c) the construction of the second triangle *d) the result*

Figure 11.7: *An example of the insertion of a constraint (thick edge) into Delaunay triangulation in E^2.*

Unlike the deletion of a vertex, all vertices of hole lie in the same half-plane (or half-space) defined by the constraint, which allows us to consider another, much easier, algorithm. It is based on the D&C strategy. Starting with the constraint edge (or face), in each step of the

recursion, the algorithm constructs a simplex such that no vertex from the hole (naturally, except for the vertices forming the simplex) lies inside the circum-sphere of this simplex. The constructed simplex issues new two edges (faces) and may split the hole into two smaller holes that are retriangulated in next step – see *Figure 11.7b, c, d*. At the end of the insertion of the constraint, the connectivity between simplices is updated.

Sloan [Slo92] suggested another algorithm for the insertion of a constraint into the triangulation in E^2. Its generalization to higher dimensions seems to be possible, however, very complicated. Starting from any triangle containing the first vertex of the given constraint, the algorithm searches the triangulation until it reaches the triangle containing the second vertex of the constraint. For each triangle visited during the search, the algorithm checks whether there is an edge intersected by the constraint. If the outcome of this test is positive, the edge is flipped. It can be shown that the successive performing of flips ensures that when the second vertex of the constraint is reached, the constraint is included in the triangulation. Afterwards, the legalization has to be performed in order to restore the Delaunay property of the triangulation. An example of such insertion of a constraint can be seen in *Figure 11.8*.

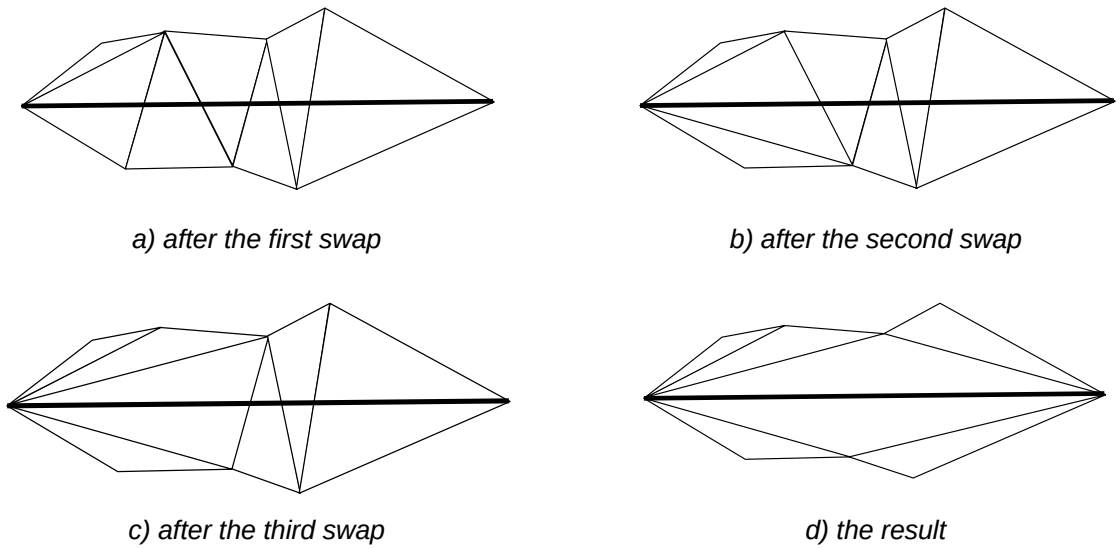


Figure 11.8: *An example of the insertion of a constraint (thick edge) into Delaunay triangulation in E^2 using the successive application of local transformations.*

No matter which of algorithms we have just described is used, the generalization of proposed parallel algorithms for the CDT is, without any doubt, similar to the already discussed problem of the deletion of a vertex from the triangulation and, therefore, impacts of the generalization on the parallelization are the same.

There is also another possibility how to insert constraints into the Delaunay triangulation. Maur et al. [Mau04] show that for a given set of points, it is possible to assign a weight to every point in such a manner that the regular triangulation of these points is identical to the required CDT. Although authors are focused on planar case only, the extension to E^3 seems to be possible. Further research is needed. If we were able to convert the problem of the construction of the CDT into the problem of construction of the regular triangulation, the generalization of proposed parallel algorithms would be simple – see Section 11.1.

11.4.1 CDT Based Parallel Algorithm for Clusters of Workstations

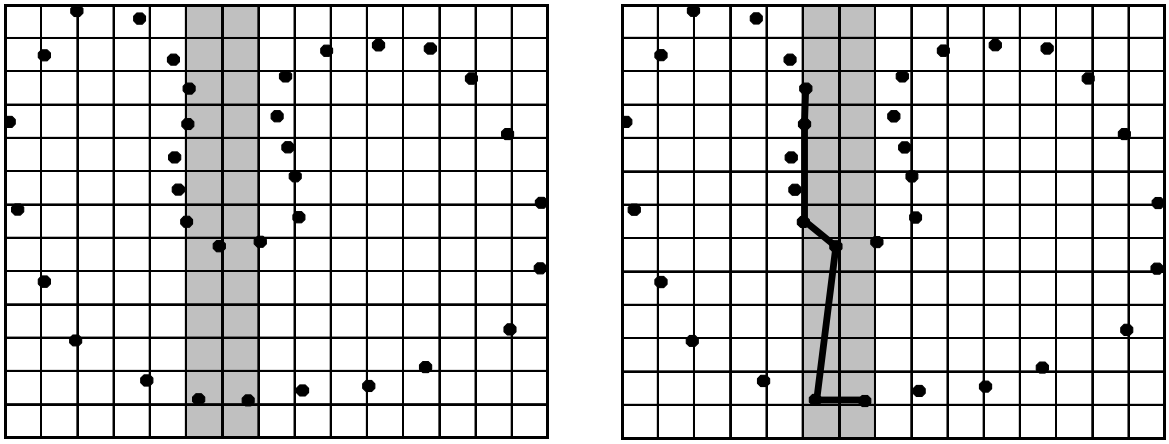
It is a sad fact that our parallel solution for clusters of workstations is not scalable due to a huge amount of communication (see Section 9). Inspired by the Hardwick's approach [Har97] described in Section 4 and exploiting the knowledge that the legalization process stops on a constraint, we decided to develop an algorithm that does not need to communicate. In the first stage, points are subdivided into k disjunctive areas where k is the number of available processors. Then, edges (or, in E^3, faces) such that they will be in the resulting Delaunay triangulation are constructed on boundaries between areas. These edges (or faces) are marked as constraints. Afterwards, a primary triangulation of points of these constraints is created and it is together with the constraints submitted to every processor. Each processor then inserts its points. As the constraints are Delaunay edges (or faces), they are never flipped and, therefore, the communication can be avoided. In the last stage, local triangulations are merged together.

Let us describe the algorithm in detail. For easier understanding, we explain it in E^2. The extension to E^3 is simple. The first stage is very close to the Hardwick's one. At the beginning, Mueller's algorithm (see Section 6) is used to subdivide points. It requires $O(N + k \cdot R)$ time where N is the number of points to be triangulated and R is the number of cells in the grid (in the worst-case R is N). Let us note that Hardwick needs $O(N \cdot \log N)$.

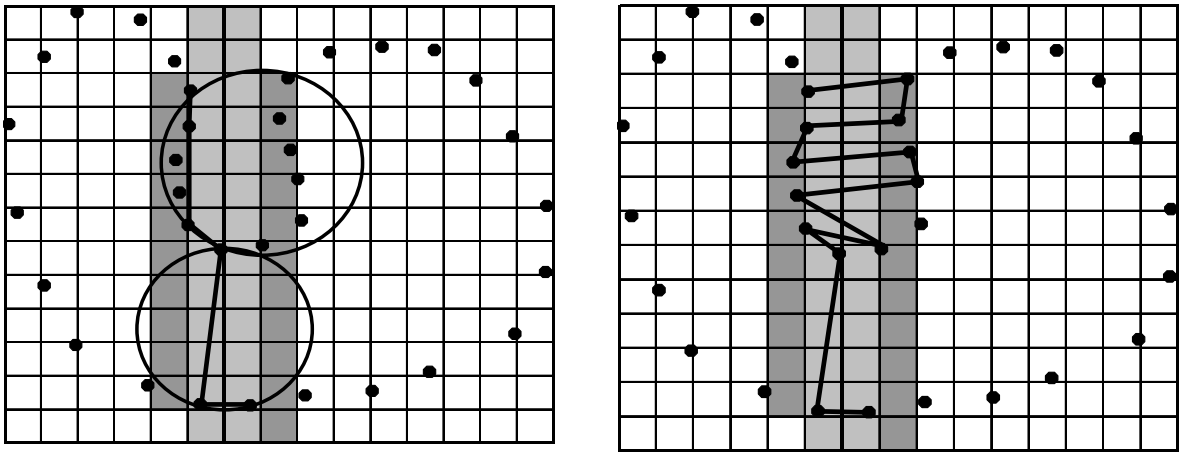
For each boundary between two areas, the projection plane perpendicular to the plane of input points and going through the boundary is constructed. Points lying in cells along the boundary are projected onto this projection plane using the same approach as Hardwick, the lower convex hull of the projected points is found and back projection gives a set of line segments. Unlike the Hardwick's algorithm, only a subset $M_1$of all points S is used in the computation of the lower convex hull $CH(M)$ and, therefore, constructed line segments may not be a part of the resulting Delaunay triangulation. Therefore, for each triple of points connected by two line segments, a circum-circle of these points is created and it is tested whether it covers only cells whose points were chosen for the computation of $CH(M)$. If the outcome of this test is negative, points lying in adjacent cells covered at least partly by this circum-circle are added into the set M, the lower convex hull of M is recomputed and a new set of line segments is extracted – see *Figure 11.9*. The algorithm continues in this way until all circum-circles lie inside already processed cells. When it stops, line segments are edges that will be in the Delaunay triangulation of the input set S.

The advantage of the described algorithm is that it allows processing of extremely large data sets that cannot be stored in memory of one computer, e.g., the Barbuto data set [Mich] having about 350 millions of points requires approximately 4GB of memory. As it, unlike the Hardwick's algorithm, processes only a subset of points, it runs, in the expected case, faster.

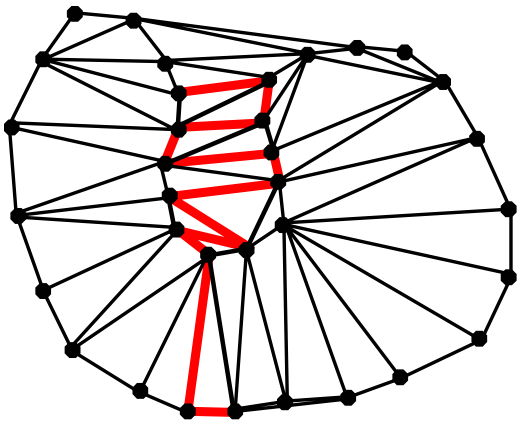
For the computation of convex hull, an algorithm by the incremental insertion can be used. The algorithm works as follows. Starting with a triangle as the initial convex hull, all points are successively tested whether they lie outside the current convex hull. If the result of the test is positive, the convex hull has to be updated in such a manner that the point lying outside belongs now to the new convex hull. *Figure 11.10* shows an example of the whole process. Although the algorithm runs in the worst-case in $O(m^2)$, it requires only $O(\mathrm{m})$ in the expected-case where m is the number of points in the set M. The advantage of this algorithm is that when new points are added into M, the whole recalculation of convex hull $CH(M)$ is not necessary; these points are simply inserted and the convex hull is updated. Therefore, to construct the Delaunay edges on boundaries, we need $O(k \cdot m)$ time in the expected-case, while the Hardwick's algorithm consumes $O(N \cdot \log N)$.



a) the first step



b) the second step



c) the resulting Delaunay triangulation

Figure 11.9: *The successive construction of Delaunay edges on boundary in E^2.*

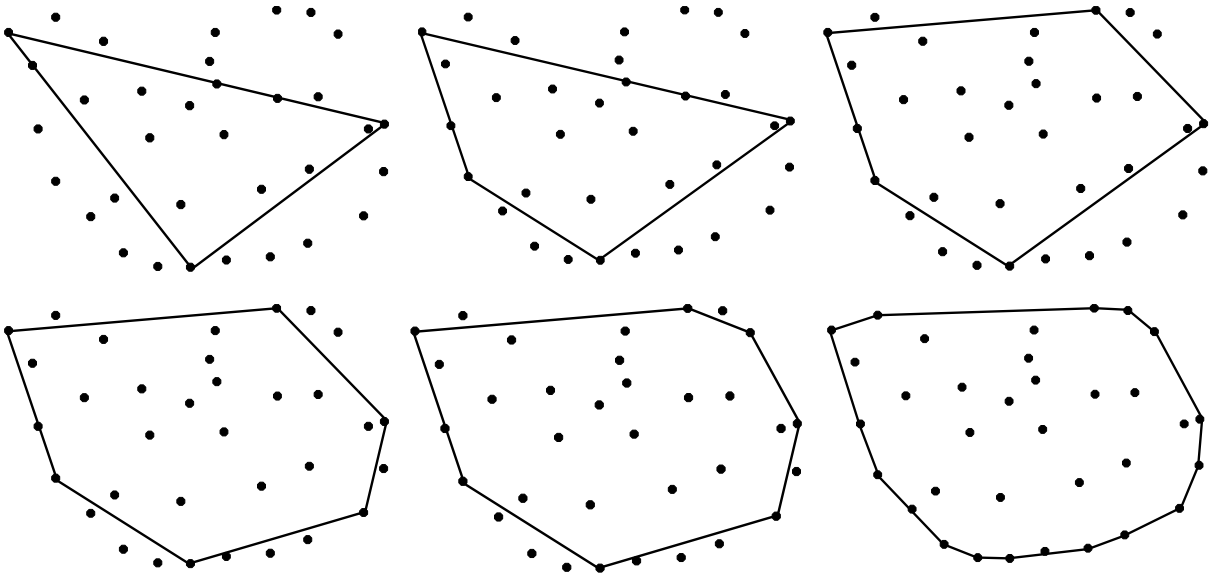


Figure 11.10: *The first five steps and the result (shown in the last image) of the construction of convex hull in E^2 by incremental insertion.*

An auxiliary big simplex is constructed, all points of edges detected in previous stage are inserted into the triangulation and constraints are found – see *Figure 11.11*. The entire initial Delaunay triangulation together with the constraints is sent to every processor. Each processor inserts the points lying in its area bounded by constraints into the triangulation. When the insertion completes, the processor sends identifications of simplices containing any constraint to its adjacent counterparts and these processors exploit the sent information to update the connectivity between local triangulations (the use of a look-up table is expected).

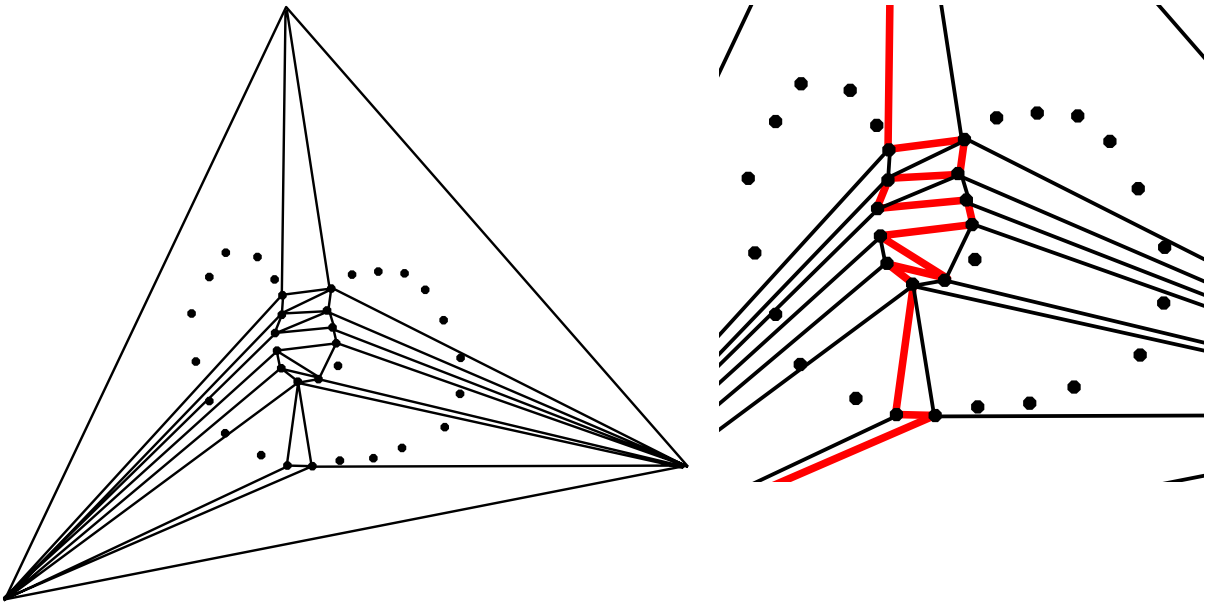


Figure 11.11: *The primary Delaunay triangulation in E^2; detail is shown in the right image. Constraints are denoted by thick edges.*

As the algorithm does not need communicate during the insertion, it should be efficient and scalable. On the other hand, points cannot arrive online (except for points lying in cells whose points were not used for the computation of convex hull), unlike the operation, data or mixed flows described in Section 8.

In this section, we have described several possible extensions to the Delaunay triangulation and discussed the impacts of these extensions on the parallelization. Except for a few cases, any parallel algorithm proposed in Sections 7 and 8 can be easily generalized to incorporate constraints given in the form of prescribed faces into the triangulation, to use non-Euclidian metrics or weights of points during the computation.

12 Conclusion and Future Work

In this thesis, we discussed sequential principles of the construction of the Delaunay triangulation in E^2 and E^3 and described the best-known existing parallel algorithms. We proposed several parallel algorithms based on the method of incremental insertion with local transformations. This method was chosen as a base for our parallelization purposes because of its simplicity, robustness (i.e., the quality of the resulting mesh is ensured) and its possibility to be easily extended for weights of points, constraints, etc. Some of these algorithms are suitable for symmetric multiprocessors (i.e., architectures with several processors and shared memory). Such a hardware configuration (especially the case with two-processors) became widely spread in the last few years in the computer graphics area. At present, there is a lack of algorithms proper for these architectures. The stress was put on the simplicity of parallelization so they could be implemented by a wide computer community, even by a person not focused on the parallel processing, and the efficiency so they could be an attractive choice in competition with long-existing serial and parallel algorithms.

While some of the proposed algorithms are easy to be implemented but not very efficient (e.g., the pessimistic method), others prove opposite behavior (e.g., the optimistic method). Some of them are usable in E^2 only (e.g., the batch method), other work in E^3 as well. We implemented all algorithms in C++ and tested them carefully on workstations with up to eight processors. According to our experiments, we reached similar efficiency as already existing, more complex, parallel algorithms. This acknowledges that a development of a sophisticated solution consumes huge time and the resulting solution often reaches comparable efficiency as the simplest one – i.e., our hint is: make your solution as simple as possible.

We also developed several algorithms for clusters of workstations (i.e., a collection of independent computers interconnected via network). Unlike the case of symmetric multiprocessors, the stress was put primarily on the ability to process large data sets that cannot be processed on one computer. We focus on the parallel construction of the Delaunay triangulation in E^3 where the existence of parallel solution is more important as there exist many applications requiring the processing of data sets having tens millions of points, e.g., the surface reconstruction. We were able to handle data sets up to 4 millions in E^2 and up to 1.4 millions in E^3 on a cluster of 8 workstations. Due to an intensive required communication between processors, unfortunately, proposed algorithms are not scalable. At the end of this thesis, we suggested, therefore, another algorithm suitable for the clusters of workstations. It is based on the CDT approach and it does not need to communicate during the insertion of points, thus it should be efficient. The implementation and verification of this algorithm is the first objective of our future research.

We proposed and implemented an application independent software layer that provides universal routines for the manipulation with data, no matter whether the data is stored locally or remotely, and means for synchronization between processors. Actually, this layer simulates the shared memory and, therefore, we call it Virtual Shared Memory (VSM) manager. The VSM was used for the parallelization of the application of surface reconstruction (its sequential version was developed by Varnuška [Var05]). Currently, we managed to distribute major data structures, such as the Delaunay triangulation or output triangular surface. The computation of DT is parallelized; all remaining parts of reconstruction run sequentially. The second objective of our future research is to complete the parallelization of this application so we are able to not only process large data sets but also process them in a reasonable time – at present the reconstruction of a data set with 1.4 millions consumes about 6 hours.

In this thesis, we also discussed several possible extensions to the Delaunay triangulation and the impacts of these extensions on the proposed parallelization. Except for a few cases, any parallel algorithm can be easily generalized to incorporate constraints given in the form of prescribed faces into the triangulation, to use non-Euclidian metrics or weights of points during the computation and to allow deletion of points from the Delaunay triangulation. The various location strategies, e.g., the approach with the DAG or walking techniques, and their effect on the efficiency of proposed parallel solution were investigated

At the end of the thesis, let us give some hints that might be useful for anybody who would like to develop his/her parallel algorithm that needs simultaneous browsing and modification of a tree or a graph, e.g., the DAG. If architecture with shared memory is considered, the most important rule is to minimize the number and the length of the critical sections as much as possible. When a shared variable has to be tested and/or modified in an atomic way, a use of appropriate atomic instruction takes only 20% of the time which is consumed when such a test or modification are implemented via critical section supported by the standard system tools. The heap management spends quite a lot of time in a critical section when a node has to be allocated (or freed). If a thread allocates an array of nodes in an advance and then successively simply picks the nodes from this array, better efficiency is achieved, especially in such a case when the shared structure is changed very often. Indeed, this simple solution trades time for memory. The efficiency can be also increased, if a context of routine calls is considered: when a thread runs some routine, some group of nodes has been already locked and there is no necessity to call the locking routine. For example, when inside the circum-circle of the first triangle a point of the second triangle lies and, therefore, the edge between them should be swapped, the nodes of the first triangle and all its neighbors are already locked. Last hint: a development of a sophisticated solution consumes a huge time and the resulting solution often reaches comparable efficiency with the simplest one, i.e., make your solution as simple as possible.

For architecture with distributed memory, e.g., clusters of workstation, the most important, as our experiments show, is to minimize the amount of required communication between processors. This is necessary, especially, in the case that a local operation (i.e., an operation that does not need to communicate in order to be completed) consumes a very short time because many of these operations are needed to counterbalance one operation that needs some communication. For example, the routine for the subdivision of a local simplex contains several instructions only and, therefore, it takes several hundreds times less time than the routine for the subdivision of a remote simplex. It is a reason why the data flow approach is not too scalable. As the communication is very expensive, some, on the first view, complex and unnecessary code may be efficient. The development of a parallel algorithm for architecture with distributed memory is quite difficult. Typically, it is being developed on a sequential computer. It can be, however, very confusing because even if the parallel algorithm seems to be efficient while it runs on this computer, its efficiency may significantly drop down when the algorithm is used on a cluster of workstations. Synchronization between processors is also not simple to be implemented. We need to avoid deadlocks caused by mutual waiting of processors. However, the detection of deadlock requires an intensive communication, which harms the efficiency.

In this last paragraph, let us summarize the benefits of our research. We proposed several novel parallel algorithms for the construction of the Delaunay triangulation in E^2 and E^3 that are simple to be understood and implemented and that are robust (in the meaning of numerical stability). The solution is suitable for symmetric multiprocessors and clusters of workstations; architectures commonly used at present. Proposed algorithms can be easily generalized for the construction of the Delaunay triangulation in E^d, to incorporate constraints given by a set of

prescribed edges (or faces), to use weights of points, etc. We also proposed a novel system simulating shared memory for the distributed computing. This system was used for the parallelization of the application of surface reconstruction from scattered point clouds based on the Delaunay triangulation.

References

[Ada03] Adamian L, Jackups R, Binkowski AT, Liang J. Higher-order interhelical spatial interactions in membrane proteins. Journal of Molecular Biology 2003; 327(1):251-272.

[Agg88] Aggarwal A, Chazelle B, Guibas L, O'Dunlaig C, Yap C. Parallel computational geometry. Algorithmica 1988; 3(3):293-327.

[Amd67] Amdahl GM. Validity of single-processor approach to achieving large-scale computing capability. In: Proceedings of AFIPS Conference, Reston, VA, 1967. p. 483-485.

[Ame02] Amenta N, Bern M, Kamvysselis M. A new Voronoi-based surface reconstruction algorithm. In: Proceeedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques, ACM Press, 1998, p. 415-421

[Att01] Attali D, Lachaud OJ. Delaunay conforming iso-surface, skeleton extraction and noise removal. Computational Geometry 2001; 19(2-3):175-189.

[Bat99] Battiato S, Cantone D, Catalano D, Cincotti G, Hofri M. An efficient algorithm for the approximate median selection problem. In: Proceedings of 5th Seminar on the Analysis of Algorithms, June 1999, Barcelona, Spain. http://www.cs.wpi.edu/~hofri/medsel.pdf

[Béc02] Béchet E, Cuilliere JC, Trochu F. Generation of a finite element MESH from stereolithography (STL) files. Computer-Aided Design 2002; 34(1):1-17.

[Ber97] de Berg M, van Kreveld M, Overmars M, Schwarzkopf O. Computational Geometry. Algorithms and applications, Springer-Verlag Berlin Heidelberg, 1997.

[Bil98] Bilas A. Improving the performance of shared virtual memory on system area networks. PhD thesis, Department of Computer Science, Princeton University, November 1998.

[Boi84] Boissonat JD. Geometric structures for three-dimensional shape representation. ACM Trans. Graphics 3, 1984, p. 266 - 286

[Bro79] Brown KQ. Voronoi diagrams from convex hulls. Information Processing Letters 1979; 9(5):223-228.

[Car03] Carr JC, Beatson RK., McCallum BC, Fright WR, McLennant TJ, Mitchell TJ. Smooth surface reconstruction from noisy range data. In: Proceedings of GRAPHITE 2003, ACM Press, 2003, pp. 119–126.

[Che01] 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, Virginia, USA, March 2001. 11 pages. SIAM Press.

[Che89] Chew L. Guaranteed-quality triangular meshes. Tech. Rep. TR-89-983, Cornell University, Ithaca, 1989.

[Chr96] Chrisochoides N, Sukup F. Task parallel implementation of the Bowyer-Watson algorithm. In: Proceedings of the 5th International Conference on Numerical Grid Generation in Computational Fluid Dynamic and Related Fields, 1996.

[Chr99] Chrisochoides N, Nave D. Simultaneous mesh generation and partitioning for Delaunay meshes. In: Proceedings of the 8 th International Meshing Roundtable, South Lake Tahoe, CA, USA, 1999. p. 55-66.

[Cig93] Cignoni P, Montani C, Perego R, Scopigno R. Parallel 3D Delaunay triangulation. Computer Graphics Forum (Eurographics'93) 1993; 12(3):C129 – C142.

[Chu03] 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):301-324.

[Cro97] Crockett TW. An introduction to parallel rendering. Parallel Computing 1997; 23(1997):819-843.

[Del34a] B. Delaunay. Sur la sphere vide. Izv. Akad. Nauk SSSR, Otdelenie Matematicheskii i Estestvennyka Nauk 7 (1934), 793-800.

[Del34b] Б. Н. Делоне, А.Д. Александров, Н. Падуровым. Математические основы структурного анализа кристаллов, Москва, Матем. литература, 1934.

[Dev01] Devillers O, Pion S, Teillaud M. Walking in triangulation. In: Proceedings of 17th Annual Symposium on Computational Geometry, ACM, Medford, Massachusetts, USA, June 3-5, 2001. p. 106-114

[Dev98] Devillers O. Improved incremental randomized Delaunay triangulation. In: Proceedings of 14th Annual Symposium on Computational Geometry, ACM, 1998. p. 106-115.

[Dev99] Devillers O. On deletion in Delaunay triangulations. In: Proceedings of SCG'99, Miami Beach Florida, ACM, 1999. p. 181-188

[Dey01] Dey TK, Giesen J, Hudson J. Delaunay based shape reconstruction from large data. In: Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics, IEEE Press, 2001. p. 19–27.

[Dwy86] Dwyer RA. A Simple Divide-and-conquer algorithm for constructing Delaunay triangulation in O(n log log n) expected time. In: Proceedings of the 2nd Annual Symposium on Computational Geometry, ACM, 1986. p. 276-284.

[Ede92] Edelsbrunner H, Shah NR. Incremental topological flipping works for regular triangulations. In: Proceedings of 8th Annual Computational Geometry, 6/92, Berlin, Germany, ACM, 1992. p. 43-52

[Fac95] Facello MA. Implementation of randomized algorithm for Delaunay and regular triangulations in three dimensions. Computer Aided Geometric Design, Elsevier, 1995; 12:349–370.

[For87] Fortune S. A sweepline algorithm for Voronoi diagrams. Algorithmica 1987; 2:153–174.

[Gar03] Garcia F, Solano J, Stojmenovic I, Stojmenovic M. Higher dimensional hexagonal networks. Parallel and Distributed Computing 2003; 63(2003):1164-1172

[God97] Godman JE, O'Rourke J. Handbook of discrete and computational geometry. CRC Press, 1997

[Gol97] Golias NA, Dutton RW. Delaunay triangulation and 3D adaptive mesh generation. Finite Elements in Analysis and Design 1997; 25(1997):331-341

[Gon02] 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):210-220.

[Gra03] Grabner M. Compressed adaptive multiresolution encoding. PhD thesis, TU Graz, 2003.

[Gui85] 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.

[Gui92] Guibas LJ, Knuth D.E, Sharir M. Randomized incremental construction of Delaunay and Voronoi diagrams. Algorithmica 1992; 7:381-413.

[Har97] Hardwick JC. Implementation and evaluation of an efficient parallel Delaunay triangulation algorithm. In: Proceedings of 9th Annual Symposium on Parallel Algorithms and Architectures, 1997. p. 22-25.

[Hop96] Hoppe H. Efficient implementation of progressive meshes. Computers & Graphics 1998; 22(1):27-36

[Ivr03] Ivrissimtzis I, Jeong WK, Seidel HP. Using growing cell structures for surface reconstruction. In: Proceedings of Shape Modeling International 2003, Seoul, Korea, May 2003, Kim M.-S., (Ed.), IEEE, p. 78–86.

[Jež99] Ježek K, Matějovic P, Racek S. Paralelní architektury a programy, University of West Bohemia, Pilsen, Czech Republic, 1999

[Joe91] Joe B. Construction of three-dimensional Delaunay triangulations using local transformations. Computer Aided Geometric Design 1991; 8(1991):123-142

[Kai86] Kai Li. Shared virtual memory on loosely coupled multiprocessors. PhD thesis, Dep. of Computer Science, Yale University, New Haven, CT, October 1986.

[Krä02] Krämer A. Topological coding and visualization grammar of the development of C. elegans. In: Proceedings of SCI 2002 / ISAS 2002. http://busselab.uni-kiel.de/Site1/downloads/selectedpublication/akraemerPaperID324EC_SCI2002.pdf

[Koh03b] Kohout J, Kolingerová I. Parallel Delaunay triangulation in E^3: Make it simple. The Visual Computer 2003, Springer-Verlag, Heidelberg; 19(7&8): 532-548

[Koh04a] Kohout J. Selected problems of parallel computer graphics. Technical report DCSE/TR-2004-02, University of West Bohemia, Czech Republic, 2004

[Koh04c] Kohout J, Kolingerová I, Žára J. Practically oriented parallel Delaunay triangulation in E^2 for computers with shared memory. Computers & Graphics 2004, Elsevier, Pergamon Press; 28(5):703-718.

[Kol02] Kolingerová I, Kohout J. Optimistic parallel Delaunay triangulation. The Visual Computer 2002, Springer-Verlag, Heidelberg; 18(8):511-529.

[Lee97] Lee F. Constructing the constrained Delaunay triangulation on the Intel Paragon. In: Proceedings of the 13th Annual Symposium on Computational Geometry, ACM, 1997. p. 464–467.

[Lee01] 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):341-352.

[Liu04] Liu ML.: Distributed computing - principles and applications. Pearson Education, Inc. 2004. ISBN 0-201-79644-9

[Mag98] Magillo P., Puppo E.: Algorithms for parallel terrain modelling and visualisation, Parallel Processing Algorithms for GIS, 1998, p. 351 – 386

[Mau04] Maur P, Kolingerová I. The employment of regular triangulation for constrained Delaunay triangulation. In: Proceedings of Computational Science and Its Applications – ICCSA 2004, Assisi, Italy, May 14-17, 2004, Springer-Verlag, p. 198 – 206

[Mich] The digital Michelangelo project, http://www-graphics.stanford.edu/projects/mich

[MPI] Message Passing Interface, http://www-unix.mcs.anl.gov/mpi/

[Mul03] Mulchrone KF. Application of Delaunay triangulation to the nearest neighbour method of strain analysis. Journal of Structural Geology 2003; 25(5):689-702.

[MVE] Modular Visualization Environment, http://herakles.zcu.cz/research.php

[Nis01] 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):5273-5301.

[Oka92] Okabe A, Boots B, Sugihara K. Spatial tesselations: Concepts and applications of Voronoi diagrams. John Wiley & Sons Ltd, 1992.

[Oku96] Okusanya T, Peraire J. Parallel unstructured mesh generation, Presented at 5th Int. Conf. on Numerical Grid Generation in Computational Fluid Dynamics and Related Fields, Mississippi, 1996

[Oku97] Okusanya T, Peraire J. 3D Parallel unstructured mesh generation, http://citeseer.nj.nec.com/article/okusanya97parallel.html

[Ost99] Ostromoukhov V, Hersch RD. Stochastic clustered-dot dithering. Color Imaging: Device-independent Color, Color Hardcopy, and Graphic Arts IV, SPIE 1999; 3648:496 – 505.

[Pan95] Pandzic I, Magnetat N, Roethlisberger M. Parallel ray tracing on the IBM SP2 and CRAY T3D. EPFL – Supercomputing review, 7, November 1995

[Par03] Park JH, Park HW. Fast view interpolation of stereo images using image gradient and disparity triangulation. Signal Processing: Image Communication 2003; 18(5):401-416.

[Per99] Perkowski MA. Digital systems design using VHDL and Verilog. Lecture Notes 1999, Portland State University, Portland, Oregon 97201, USA. http://www.ee.pdx.edu/~mperkows/CLASS_VHDL_99/050.Introduction-to-Parallel-Computing.pdf

[Pra00] Prasad L, Rao L.R. A geometric transform for shape feature extraction. In: Proceedings of the 45th SPIE Annual Meeting, San Diego,CA, 2000.

[Pup94] Puppo E, Davis LS, DeMenthon D, Teng A. Parallel terrain triangulation. International Journal of Geographical Information Systems 1994; 8(2):105-128.

[Pre85] Preparata FP, Shamos MI. Computational Geometry. Springer-Verlag, New York, 1985.

[Rad99] Radke J, Flodmark A. The use of spatial decomposition for constructing street centerlines. Geographic Information Services 1999; 5(1):15-23.

[Rao92] Rao VN, Kumar V. On the efficiency of parallel backtracking. IEEE Transactions on Parallel and Distributed Systems 1993; 4(4):427-437

[Sch99] Schewchuk JR. Lectures notes on Delaunay mesh generation. Department of Electrical Engineering and Computer Science, University of California at Berkley, CA 94720, 1999.

[Sie94] Sienicki J, Agrawal P, Agrawal VD, Bushnell ML. Superlinear speedup in multiprocessing environment. In: Proceedings of the First International Workshop on Parallel Processing, 1994. p. 261-265.

[Slo92] Sloan SW. A fast algorithm for generating constrained Delaunay triangulations. Computers & Structures 1992; 47(3):441-450

[Su94] Su P. Efficient parallel algorithms for closest point problems. Dissertation, Dartmouth College, Hanover, NH, 1994

[Sun95] Sun XH, Zhu J. Performance considerations of shared virtual memory machines. IEEE TRansactions on Parallel and Distributed Systems 1995; 6(11):1185-1194

[Spi02] Spielman DA, Shang H.T, Alper U. Parallel Delaunay refinement:algorithms and analyses. In: Proceedings of 11th International Meshing Roundtable, Sandia National Laboratories, September 15-18 2002. p.205-218.

[Sta99] Stanford 3D databases, http://www.graphics.stanford.edu/data/3Dscanrep/

[Tam01] Tammemäe K. Computer architectures. Lecture notes 2001, TU Tallinn, Estonia,

[Tek00] Tekalp AM, Ostermann J. Face and 2D mesh animation in MPEG4. Signal Processing: Image Communication 2000; 15(4-5):387-421.

[Ten93] Teng YA, Sullivan F, Beichl F, Puppo E. A data parallel algorithm for three-dimensional Delaunay triangulation and its implementation. In: Proceedings of Super-Computing '93, 1993. p.112-121.

[Var03] Varnuška M, Kolingerová I. Improvements to surface reconstruction by the CRUST algorithm. In: Proceedings of SCCG 2003, April 24-26 2003, Budmerice, Slovakia, p. 101-109

[Var05] Varnuška M. Surface reconstruction from scattered point data. PhD thesis, Univeristy of West Bohemia, 2005.

[Vig97] Vigo M. An improved incremental algorithm for construting restricted Delaunay triangulations. Computers & Graphics 1997; 22:215-223.

[Vig00] Vigo M, Pla N. Computing directional constrained Delaunay triangulations. Computer & Graphics 2000; 24:181-190.

[Wal00] 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, Texas, United States. p.26

[Wat81] Watson DF. Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes, Computer Journal 1981, 24(2):167-172.

[Whe85] Whelan D. Animac: A multiprocessor architecture for real-time computer animation. Ph.D. dissertation, California Institute of Technology, 1985.

[Xia02] Xiao Y, Yan H. Text region extraction in a document image based on the Delaunay tessellation. Pattern Recognition 2003, 36(3):799-809

[Žal99] Žalik B. Database of Terrain Data, University of Maribor, 1999

[Žal03] 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):119-138.

Appendix: Activities

Reviewed publications

- already published

[1] Kohout J, Hlavatý T, Kolingerová I, Skala V. Feature extraction of 2-manifold using Delaunay triangulation. In: Proceedings of 17th Conference on Scientific Computing Algorithmy 2005, Slovak University of Technology, Podbanské, Slovakia, March 13-18, 2005. p. 290-299

[2] Kohout J, Kolingerová I, Žára J. Practically oriented parallel Delaunay triangulation in E^2 for computers with shared memory. Computers & Graphics 2004, Elsevier, Pergamon Press; 28(5):703-718.

[3] Kohout J. Delunay triangulation in parallel and distributed environment. State of the Art and Concepts of Doctoral Thesis, University of West Bohemia, Czech Republic, 2004

[4] Kohout J, Kolingerová I. Parallel Delaunay triangulation in E^3: Make it simple. The Visual Computer 2003, Springer-Verlag, Heidelberg; 19(7&8): 532-548

[5] Kohout J, Kolingerová I. Parallel Delaunay Triangulation based on Circum-Circle Criterion. In: Proceedings of SCCG 2003, Comenius University, April 24-26, 2003, Budmerice, Slovakia. p. 85-93 – *published also in ACM Publishing House, NY, ISBN 1-58113-861-X*

[6] Kohout J, Kolingerová I. Parallel Delaunay Triangulation in 2D and 3D. In: Proceedings of East West Vision 2002, Österreichische Computer Gessellschaft, September 12-13, 2002, Graz, Austria. p. 143-148

[7] Kohout J. Paralelní Delaunatova triangulace. Master Thesis, University of West Bohemia, Czech Republic, 2002

[8] Kolingerová I, Kohout J. Optimistic parallel Delaunay triangulation. The Visual Computer 2002, Springer-Verlag, Heidelberg; 18(8):511-529 – *cited by*:

  [1] Harris FC. Theory of Parallel and Distributed Processing, Course CS 732, University of Nevada, USA, http://www.cse.unr.edu/~fredh/class/732/S2003/class/class31.html

[9] Kolingerová I, Kohout J. Pessimistic threaded Delaunay triangulation by randomized incremental insertion. In: Proceedings of Graphicon 2000, August 28-30, 2000, Moscow, Russia. p. 76-83 – *cited by*:

  [1] Gavrilova ML. Empirical studies of optimization techniques in the event-driven simulation of mechanically alloyed materials: iterative solving environments and optimization techniques for scientific applications. The Journal of Supercomputing, May 2004, vol. 28, no. 2, pp. 165-176(12)

  [2] Akl SG. Inherently parallel geometric problems. Technical Report 2004-480, Queen's University, Kingston, Canada, April 2004. http://www.cs.queensu.ca/home/akl/techreports/tr.ps

[3] Gavrilova ML. On a nearest-neighbor problem under Minkowski and power metrics for large data sets. The Journal of Supercomputing, May 2002, vol. 22, no. 1, pp. 87-98(12)

- accepted for the publication

[10] Kohout J, Kolingerová I, Žára J. Parallel Delaunay triangulation in E^2 and E^3 for computers with shared memory. Parallel Computing 2005, Elsevier, North-Holland – *currently in press*

Non-reviewed publications

- related to this work

[11] Kohout J. Selected problems of parallel computer graphics. Technical Report DCSE/TR-2004-02, University of West Bohemia, Czech Republic, 2004

[12] Kohout J. Parallel Incremental Delaunay Triangulation. In: Proceedings of 5th Central Europian Seminar on Computer Graphics*,* Comenius University, Budmerice, Slovakia, 2001. p. 85-94 – *awarded by the third prize in the best paper competition*

[13] Doubek J, Kohout J. Spojenou silou, Systém pro distribuované zpracování - GSD. Chip 10/01 + CD ROM, 2001. p. 154 (*in Czech*)

- other

[14] Kohout J, Mautner P, Zuzák F. Automatická detekce částic v digitálních mikrogramech. In: Proceedings of *15. Konference s mezinárodní účastí - Výpočtová mechanika '99,* Nečtiny, Czech Republic, 1999. (*in Czech*)

[15] Kohout J, Mautner P, Zuzák F. Metody zpracování digitálního ferogramu pro tribodiagnostiku. In: Proceedings of *15. Konference s mezinárodní účastí - Výpočtová mechanika '99,* Nečtiny, Czech Republic, 1999. (*in Czech*)

Presentations and talks abroad

- December 2004 – Velká data v počítačové grafice, invited talk (in Czech), VŠB–TU Ostrava, Czech Republic
- September 2004 – Surface reconstruction as an application of Delaunay tetrahedronization, presentation (together with Michal Varnuška), TU Graz, Austria
- June 2004 – Delaunay triangulation in parallel and distributed environment, presentation, TU Maribor, Slovenia
- January 2004 – Paralelní rendering, invited talk (in Czech), VŠB–TU Ostrava, Czech Republic
- October 2003 – Delaunay triangulation in parallel and distributed environment, presentation, TU Graz, Austria
- May 2003 – Delaunay triangulation in 2D and 3D in parallel and distributed environment, presentation, TU Chemnitz, Germany

Stays and other activities

- 2004 – main researcher of project FRVŠ 1342/2004/G1
- September 2004 – two weeks stay at TU Graz, Austria, project Aktion 36p9
- September 2004 – passive participation on the EG 2004 conference
- June 2004 – one week stay at TU Maribor, Slovenia, project Kontakt 16-2003-04
- September 2003 – passive participation on the EG 2003 conference
- October 2003 – two weeks stay at TU Graz, Austria, project Aktion 36p9
- May 2003 – one week stay at TU Chemnitz, Germany, Socrates-Erasmus teaching mobility
- Spring 2003 – one semester study abroad at Queen's University of Bath, UK, Socrates-Erasmus mobility
- 2002 - 2004 – researcher of project MSMT 23500005
- 2001 – researcher of project AV030801

Known requests for proposed parallel code

- Clément MENIER <Clement.Menier@inrialpes.fr>, PhD student at INRIA MOVI, France – Parallel 3D Delaunay triangulator (optimistic method prefered)
- "joe alter, inc" <joealterinc@hotmail.com>, staff of a small company – Parallel 3D Delaunay triangulator, *as the code (according to our institutional regulations) cannot be used in commercial applications, the request could not be fulfilled.*

Appendix: Color Plates

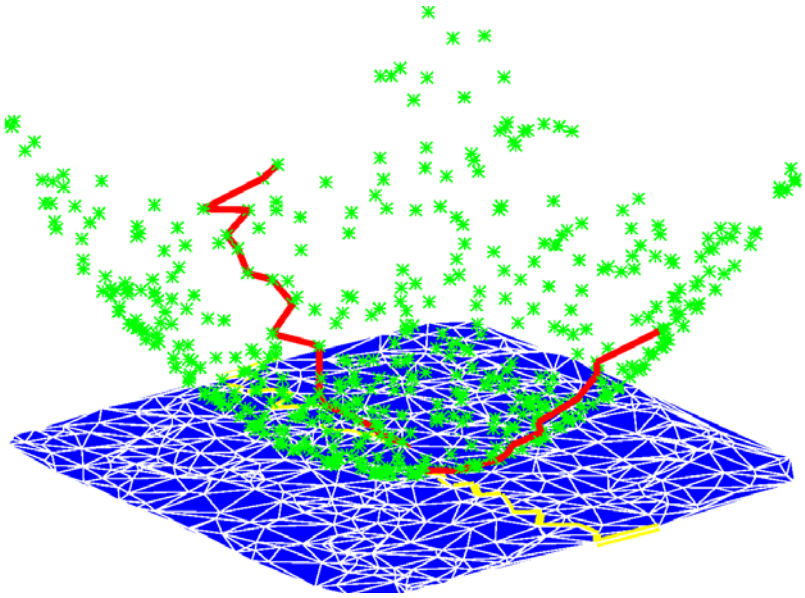
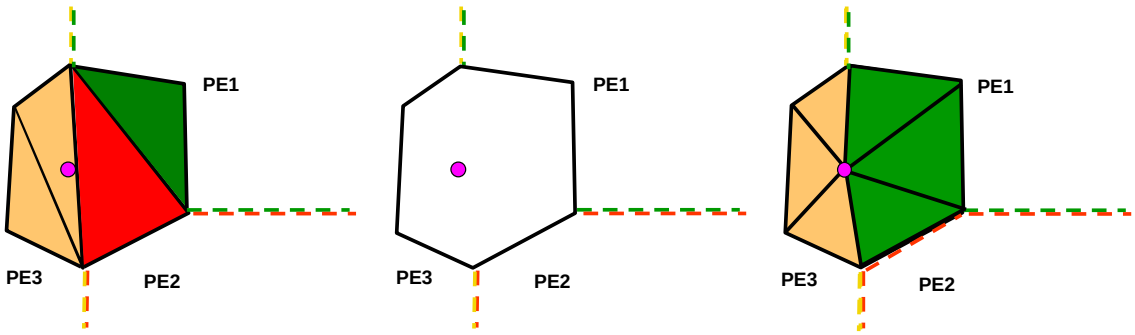
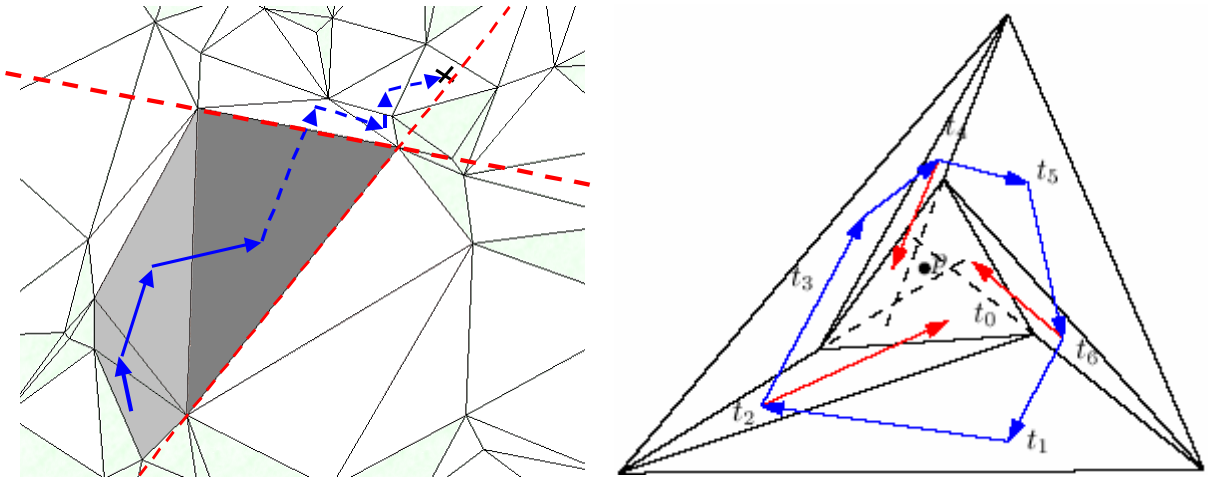


Figure 3.7: *The projected points and the corresponding Delaunay triangulation [Har97]. Only a tiny part of the 3D convex hull is shown (red bold line segments).*



a) the original partition *b) the shared cavity* *c) the cavity retriangulation*

Figure 4.2: *The insertion of a point (big dot) into the triangulation in E^2. This insertion causes retriangulation of a cavity shared by three processors (PE1, PE2 and PE3) including the update of their regions' boundaries. The simplices belonging to the same region are shown in the same color.*



a) the path of visibility walk, the dark gray triangle is currently being tested, light gray triangles were visited in previous steps

b) an infinite cycle for the visibility walk [Dev01]

Figure 5.3: *The visibility walk algorithm.*