ZÁPADOČESKÁ UNIVERZITA

University of West Bohemia in Pilsen
Department of Computer Science and Engineering
Univerzitni 8
30614 Pilsen
Czech Republic

# Selected Problems of Parallel Computer Graphics

**Josef Kohout**

Distribution: public

Technical Report No. DCSE/TR-2004-02
March, 2004

Technical Report No. DCSE/TR-2004-02

# Selected Problems of Parallel Computer Graphics

**Josef Kohout**

## Abstract

This thesis gives an overview of parallel techniques used in computer graphics. It investigates the problems of work distribution, load balancing, communication traffic reduction and merging of subsolutions. These problems are discussed in the relation to parallel rendering, the representative of traditional computer graphics, and the Delaunay triangulation, the representative of the computational geometry. The description is illustrated by numerous examples. This thesis also proposes several new parallel algorithms for the construction of Delaunay triangulation suitable for architectures with several processors and the shared memory. The proposed solution was tested and the results are summarized in this thesis. Preliminary proposal of some algorithms suitable for cluster architectures is given in the final part of the thesis.

This work was supported by the Ministry of Education of the Czech Republic – project MSM 235200005 and project AV2030801

# Table of Contents

# 1 Introduction

The computational power of sequential computers grows. We are currently able to process data sets in fraction of the time that was needed few years ago. We can also process data sets such that their processing was impossible on old computers because of the technical limitation of these computers. Although the grow of the computational power is significant, the demand for the processing of larger data sets or for the processing in such a manner to get the more quality result or to get the results in an interactive or even real-time seems to outpace this growth in many applications.

When an algorithm running on a sequential computer is not powerful enough to process a given data set according to the expected conditions or even it does not allow the processing at all, there are several ways how to deal with this data set. Simple and straightforward solution is to reduce the amount of data and/or to take some conditions back. Let us note that in many applications, however, this solution is not possible. We can also buy or lend more powerful sequential computer. If neither the first nor the second solution brings a substantial improvement, it is a time to look for a better sequential algorithm, i.e. faster one, demanding less resources, etc. Let us note that the new algorithm has to be compatible with the original one, i.e. it has to fulfill the same condition (e.g. an algorithm that does not ensure the requested quality of the results is useless). We succeed in our task often but not always.

If we are not successful on the field of sequential algorithms, the search for a parallel algorithm is the next logical step. The situation in the parallel processing is not, unfortunately, as easy as in the sequential processing because there exist plenty of parallel architectures and each of them is predetermined to be used for a different kind of problems. The algorithm designed for some architecture can be sometimes adopted for another but it requires very often a significant effort because different parallel architectures require different programming techniques. Moreover, the performance of such a modified algorithm is very often much lower than the performance of the original one.

If we are not able to find any existing appropriate parallel algorithm (it is common in a case that the solved problem is not well-known or if we put too strict conditions on the solution), the last possibility is to develop a parallel algorithm, i.e. to parallelize a sequential one. Let us note that the strategy 'to develop' brings often better results than the strategy 'to adopt'. According to the chosen parallel architecture we have to decide how the processors will naturally communicate. May a processor communicate directly with any other processor or it requires a courier for the communication with distant processors? Is there such a memory that is accessible for read-write purpose to more processors? How to reduce the communication traffic? What data structures should be used? Is it necessary for a processor to wait for a reply on its request or it may work simultaneously on a different job? Besides the problem of the communication, there is also a problem how to split the work among the processors because the effective algorithm has to ensure that no processor is idle for a significant time. Another question is how to compose the partial results in order to get the final result. These are only some of the questions that have to be answered.

## 1.1 Main Goals

The computer graphics (including computational geometry) is a field of human activity, which often demands large amounts of processing. Parallel architectures have been employed to meet the considerable demands of many problems. This thesis provides an overview of parallel techniques used in, according to our opinion, the most known applications from this field of research. Although we present these techniques in the relation to the applications, there are general enough. First, we investigate the parallel rendering, the representative of the traditional computer graphics, in detail. Next, the parallel construction of the Delaunay triangulation, an important problem of the computational geometry, is investigated. We have focused on data decomposition and load balancing, data structures used to reduce the communication traffic and the principles used to merge immediate results into the final result. We illustrate the discussion with numerous examples including the current state of the art.

The experience acquisitioned in the theoretical part of this thesis is used in the development of ourselves parallel algorithm for the Delaunay triangulation based on the method of the incremental insertion with local transformation. This method is simple and robust (it produces fine quality meshes), however, it is rather bit of a sequential character. Therefore, no wonder that there is, as far as the authors know, no existing parallel algorithm.

## 1.2 Acknowledgments

The great thanks belong to Dr. Ivana Kolingerová, from the University of West Bohemia, Pilsen, Czech Republic, who is the leader of this project. Thanks belong also to my parents and my friends and colleagues for their help and support.

I would like to thank 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. Thanks belong also to UniSys, Czech Republic for allowing us to experiment on their computer.

I would also like to thank Prof. V.Skala from the University of West Bohemia, Pilsen, Czech Republic for providing conditions in which this work has been possible and to thank anybody else who has participated on this research.

This work was supported by the Ministry of Education of The Czech Republic - project MSM 235 200 005 and the project AV2030801.

## 1.3 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 $c_1$, $c_2$ and $N_0 > 0$ and the asymptotical complexities $f_x(N)$ and $g_x(N)$, where $x$ is either *max*, *min* or *avg*. If $c_1 \cdot g_x(N) \le f_x(N) \le c_2 \cdot g_x(N)$ 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 their constant that would limit the function $f_x(N)$ from both sides. Therefore, let 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 average asymptotical complexity 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).

**Simplex** is a geometrical object in the dimension $d$ having $d+1$ vertices, each vertex is connected with all other vertices, i.e. it can be described by full graph. In $E^1$ it is a line segment, in $E^2$ a triangle and in $E^3$ a tetrahedron.

**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.

## 1.4 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 Elements – processor |
| PEs | | number of processing elements |

## 1.5 Organization of the Thesis

The rest of this thesis is structured into seven sections as follows. The next section gives a survey of existing parallel architectures and discusses general principles for the parallel algorithm design. This section should provide a good basic background to understand the problem of parallelization.

Section 3 investigates parallel computer graphics, mainly parallel rendering (including polygon rendering and ray-tracing), which is one of the most power demanding application and, therefore, many parallel techniques were proposed for this problem. Among these techniques there belong an even subdivision of the scene among processors, merging of the partial solutions (see D&C) and the use of special data structures in order to minimize the communication costs. Brief introduction to the problem of the rendering should provide a background for a reader who is not focused on the problems of computer graphics.

In Section 4, the problem of the Delaunay triangulation is described in detail. We discuss the properties of the DT and the methods for its construction. A survey of parallel algorithms of the DT is presented in Section 5. As the divide and conquer strategy is the most suitable one for many geometry problems and the problem of the construction of the DT is not an exception, we focused on the merging of the subsolutions.

The remaining sections are more practically oriented. The sixth section describes a sequential method for the construction of the Delaunay triangulation that was chosen as a base for our parallel algorithms. We have proposed several parallel algorithms for the parallel architectures with several processors and the shared memory, common equipment of the computer graphics laboratories. The detailed descriptions of these algorithms with the results of the performed experiments are given in Section 7.

Last section, Section 8, summarizes current results and concludes this thesis. In this section, the possibilities of the future work (e.g. the development of parallel Delaunay triangulation for clusters and the integration of the proposed algorithms into an application) 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 executions 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. E.g., 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 wheather 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 a quite a big set of physical parallel models to be adopted in a parallel computer. They can be generally classified into six practical models [Per99]: SIMD, parallel vector processor, symmetric multiprocessor, massive parallel processor, cluster (or network) of workstations and distributed shared memory.

Single Instruction Multiple Data (SIMD) was firstly introduced by Flynn in 1966. This model 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 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.

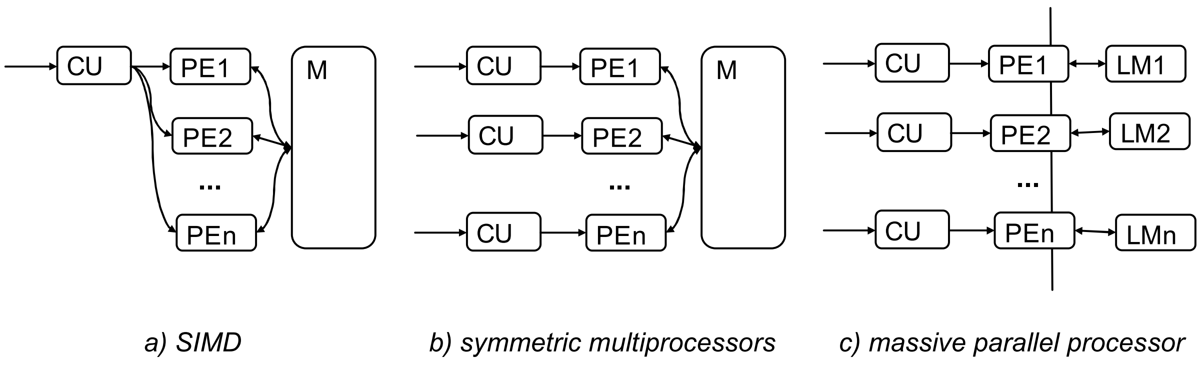


**Figure 2.1:** *Scheme of the model structures. Legend: PE - processing element, CU - control unit, M - memory, LM - local memory.*

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 $2^n$ 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).

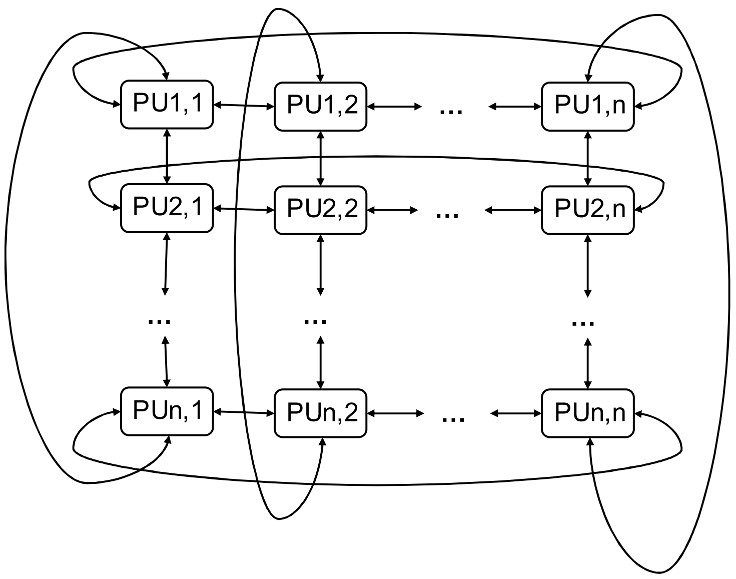


**Figure 2.2:** *An example of the topology of $E^2$ grid used in a massive parallel processor. Processing unit (PU) includes a processing element (PE) and its local memory (LM).*

Cluster of workstations (or also computer networks) 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.

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 machine 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 for the communication among the processes message passing (i.e. sending and receiving messages) 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 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).

Fixed partition of the data set, however, introduces a possibility of imbalanced workloads of PEs: the difference of the time consumed by the processing element that finished its work as the last one and the time consumed by the processing element that finished as the first one is not insignificant. Therefore, some algorithms use dynamic load-balancing that provides more flexibility in assigning workloads to processing elements. As the optimal dynamic load balancing strategy is *NP*-complete problem, some more or less efficient heuristics have to be 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 exists 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 the efficiency. A very important characteristic, however, is the scalability of a MPMD algorithm, which shows the capability of the algorithm to employ efficiently large number of processors. 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 common error that some researches do 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 contrary, 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 more 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 $s$:

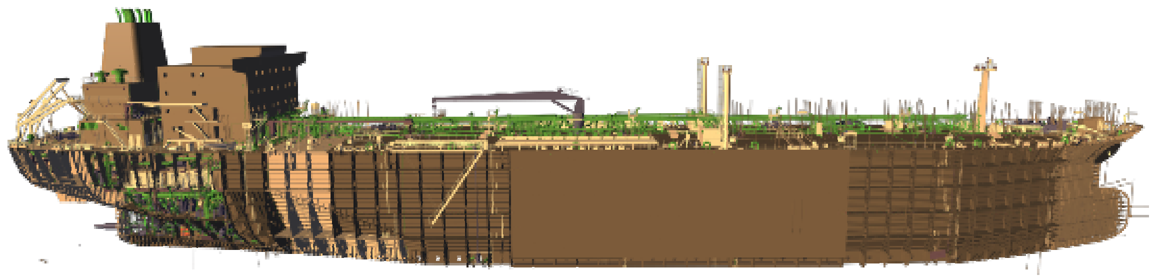
$$s = \frac{t_{seq}}{t_{par}} = \frac{N \cdot t_1 - x \cdot (t_1 - t_0)}{\frac{N}{2} \cdot t_1 - x \cdot (t_1 - t_0)} > 2 = \frac{N \cdot t_1 - x \cdot (t_1 - t_0)}{\frac{N}{2} \cdot t_1 - \frac{x}{2} \cdot (t_1 - t_0)} \tag{2.5}$$

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 least reason is typical for algorithms that perform a task only if some condition is matched. If we parallelize such an algorithm, we can avoid processing of more useless tasks than in the original serial algorithm. A good example is the backtracking [Rao92] where a node of the tree is expanded only in a 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.

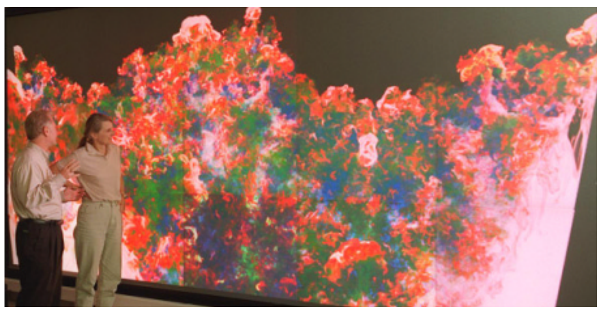
# 3 Parallel Computing in Computer Graphics

In many tasks in the computer graphics, there is a necessity to process big data sets [Hum02]. How to define a big data set? It is such a data set that its processing demands more resources than are available if we consider sequential processing only. Specific hardware limitations, time, spatial resolution and money belong among these resources. Big data sets are, typically, results of some measurement (e.g. range scanning, imagery, sensor networks) or results of some simulation (e.g. scientific computing, light transport, CAD). *Figure 3.1* shows several examples of really big data sets.



*a) double eagle tanker model containing 83 million triangles [UNC]*

*b) scans of Saint Matthew (386 MPolys) and the David (2 GPolys) - Stanford Digital Michelangelo Project [Mich]*



*c) simulation of compressible turbulence (2K x 2K x 2K mesh) - Sean Ahern and Randal Frank, LLNL [Sch00]*

**Figure 3.1:** *An example of big data sets.*

There are two numerous groups of algorithms that need to deal with big data sets. The first one consists of algorithms that visualize big data sets (see *Figure 3.1a* and *Figure 3.1b*). The problem of rendering is the best known. We can identify five rendering strategies [Wat00, Cro97]: polygon rendering, volume rendering, ray-tracing, radiosity rendering and terrain rendering. They differ in the type of scenes, which they are able to process, and also in the available quality of the results.

The second group includes mainly computational algorithms solving geometry problems. Typical problems are convexity problems, intersection problems, geometric searching problems, proximity problems (e.g. Voronoi diagrams), mesh generation problems (e.g. Delaunay triangulation) and optimization [Pre85]. Algorithms for the surface reconstruction (see *Figure 3.1b*), iso-surface extraction, etc. [Wat00] can be also included into this group.

In this section, we investigate parallel rendering, the representative of the first group. As the problem of parallel rendering is comprehensive, we focus on the best known rendering strategies: polygon rendering and ray-tracing only. Parallel algorithms for polygon rendering contain many approaches how to subdivide the work among the processors and how to merge the immediate results into the final solution. Ray-tracing introduces the problem of dynamic load-balancing and the problem of the use of proper data structures in order to reduce the communication among processors.

Parallel Delaunay triangulation, the representative of the second group, with its merging problem is discussed in Sections 4 and 5.

## 3.1 Polygon Rendering

A scene for polygon rendering is a collection of objects in three-dimensional space with associated properties (e.g. lightning and viewing parameters). The object may be a model of a real or fictitious thing. Each object is described by a set of polygons, typically, a set of triangles. Most descriptions of the geometry or shapes of objects are only approximate. In many applications, the approximation is chosen as accurate as images with acceptable quality can be produced. In CAD applications, however, the description has to be accurate because it is often used to drive a manufacturing process. As more polygons usually imply better accuracy of approximation, CAD models often contain several millions of polygons. *Figure 3.1a* shows a CAD model containing about 83 millions of triangles.

The polygon representation of objects is simple and, therefore, it is commonly used. As, moreover, objects represented by bi-cubic parametric patches, CSG, voxels and implicit functions are 'simply' transformable into polygon representation, no wonder that polygon rendering is one of major importance in computer graphics.

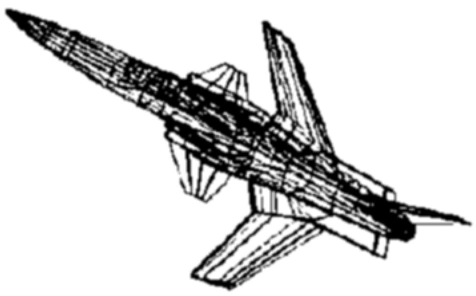
Rendering engines has to perform two main tasks [Wat00]. The first is to process the geometry of the object as it is determined by various transformations, e.g. modeling transformations, viewing transformations, etc. In this *transformation phase*, the geometry is transformed from 3D to 2D. It is followed by the *rasterization phase* that generates pixels from just computed 2D geometry. The rasterization includes a shading algorithm, which finds the appropriate shade for each pixel within the polygon's projection, and a hidden surface removal algorithm, which evaluates whether part of a polygon is obscured by another one that is closer to the viewer.

Shading algorithms evaluate intensity at the vertices of the polygon, then interpolate from these values to find an appropriate intensity for the polygon pixels. Coincident with this operation, a similar interpolation scheme evaluates the depth of each polygon pixel from the depths at the vertices (which are evaluated from the geometry of the scene and the viewer). These depths are stored in an array, known as Z-buffer. If the currently calculated depth of a polygon pixel is lower than the previously stored depth, the value in Z-buffer and intensity of an appropriate image pixel are updated.

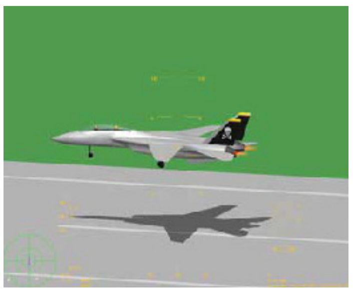
### 3.1.1 Functional Parallelism in Polygon Rendering

For complex scenes or high-quality images, the rendering is computationally intensive, requiring millions or billions of floating-point and integer operations for each image. No matter how fast is a polygon rendering implementation (no matter whether it is a software or a hardware implementation) and a system (may be sequential or parallel), on which it runs, the demand for power to render larger and more complex models in higher quality always seems to outpace it. Visualization of large models or producing of realistically looking images has applications in many areas including medicine, engineering and entertainment.

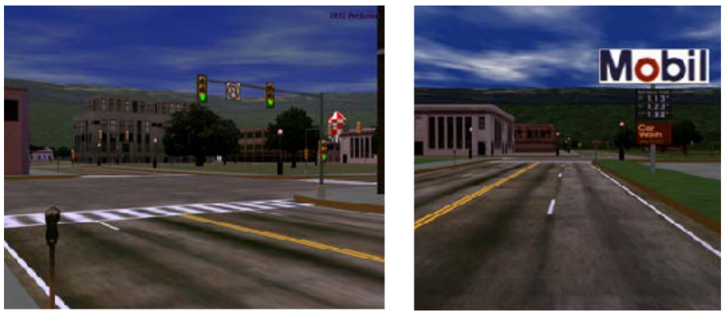
Let us demonstrate the hunger for the rendering performance. Prior to 1987 it was demanded only to produce wire models. Rendering of small scenes, usually with the flat shading algorithm, into images with low resolution (e.g. 320x200), which was successful in the years 1987 – 1992, was superseded by rendering in the years 1992 – 2000, which could process larger scenes and produce images in higher resolutions and also better looking images due to the use of textures and antialising algorithms. The era 2000 up to now brings possibility of programmability. Curved surfaces, programmable shading and image-based rendering (IBR) are supported. Image-based rendering refers loosely to techniques that generate new images from other images rather than from geometric primitives. IBR seems to hold the promise of shortcutting the traditional modeling/rendering pipeline or at a minimum hiding the latency between rendered frames. All these improvements result in the convergence of computer graphics and media processing that opens a possibility to produce fully rendered movies, i.e. movies without real actors. *Figure 3.2* shows several examples of applications from each generation. These examples were adopted from [Ake01].



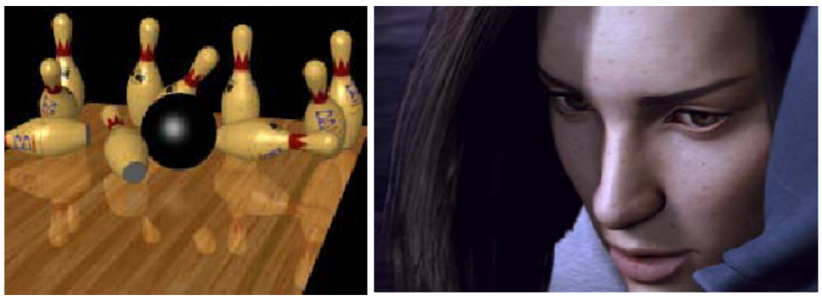
*a) wire model, prior to 1987*



*b) simple 3D model, 1987 - 1992*



*c) textures, large models, 1992 - 2000*



*d) programmability, 2000 - ?*

**Figure 3.2:** *Development of rendering techniques - mainly polygon rendering (b,c,d).*

Since 1987 hardware technology has advanced, processors are hundred times faster, memory sizes larger and their bandwidth is about Mbytes per second, a hardware implementation of polygon rendering was incorporated into graphical adapters. Let us give a brief description of this rendering hardware implementation. We can identify at least two phases in the polygon rendering, which are independent enough to be performed in a parallel way. These are the transformation and the rasterization phases. Usually it is necessary to render a sequence of images instead of a single one and, therefore, the functional parallelism comes into the consideration. The pipelining is a straightforward and simple solution used in all current graphical adapters. *Figure 3.3* shows a typical polygon rendering pipeline. The number of units and their order, however, varies in details of implementation. Nowadays (August 2003), powerful ATI and nVidia graphics adapters have 12 stages of pipelining.

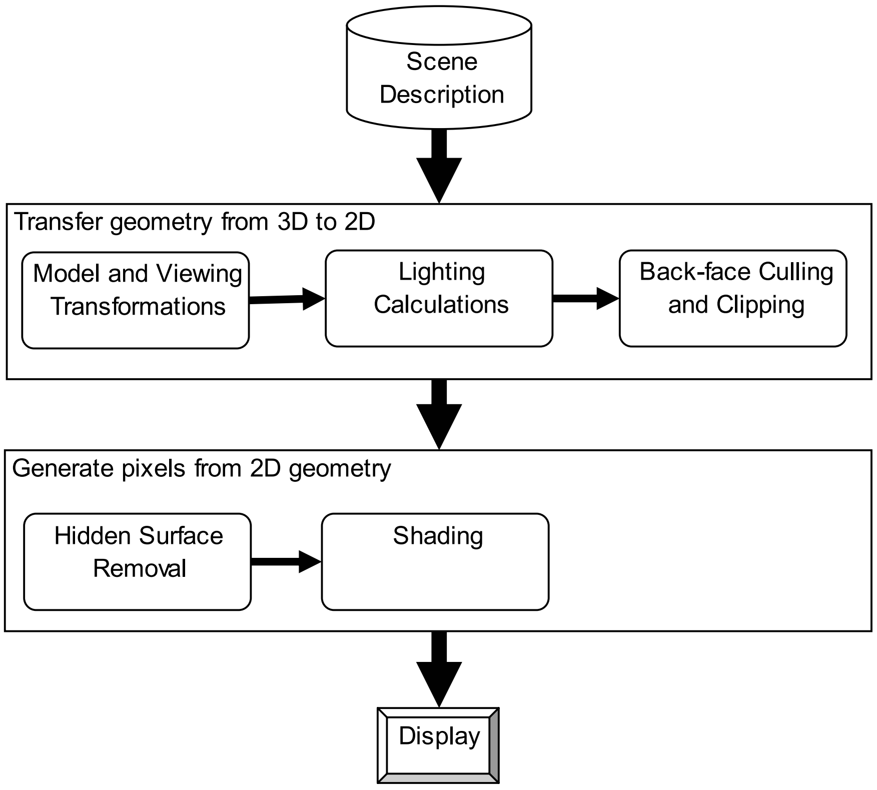

Scene Description
Transfer geometry from 3D to 2D
Model and Viewing Transformations
Lighting Calculations
Back-face Culling and Clipping
Generate pixels from 2D geometry
Hidden Surface Removal
Shading
Display

**Figure 3.3:** *A typical polygon rendering pipeline.*

### 3.1.2 Data Parallelism in Polygon Rendering

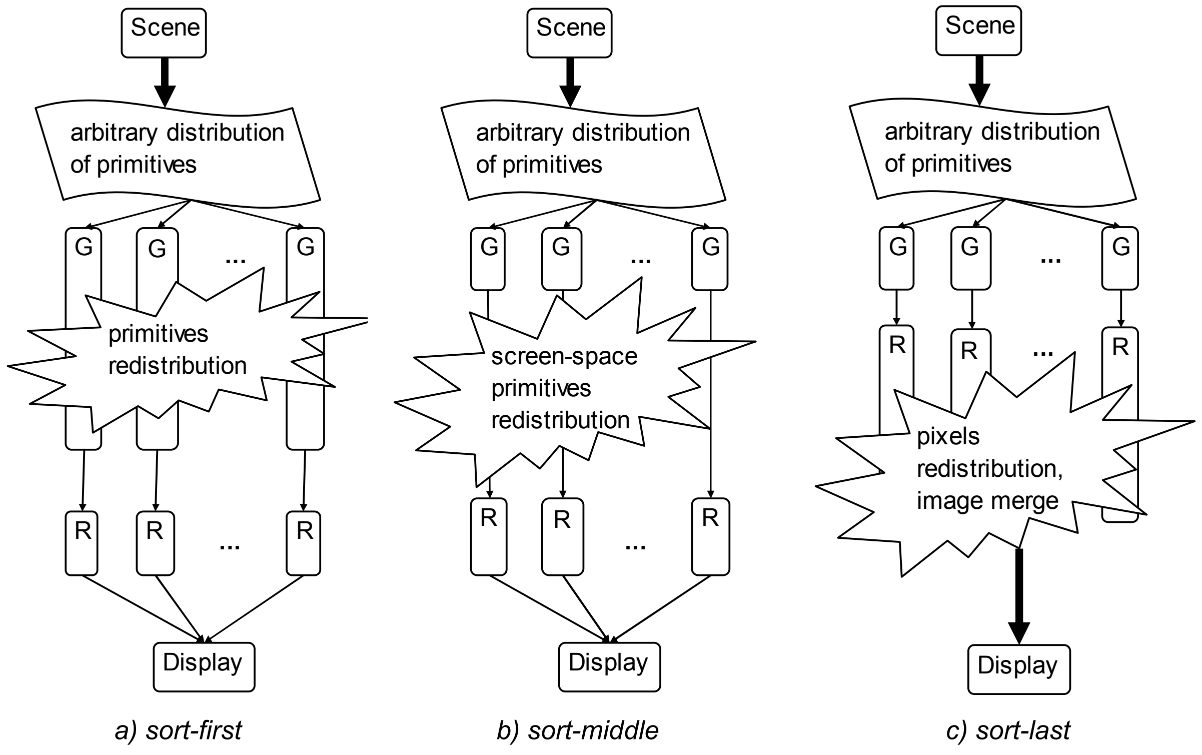
No matter how fast graphics hardware technology advances, demanding applications, such as scientific visualization, CAD, vehicle simulation and virtual reality can require hundreds of MFLOPS of floating-point performance and gigabytes per second of memory bandwidth, far beyond the capabilities of a sequential computer. For example, GeForce 4 Ti 4800 nVidia's top graphical adapter (August 2003) cannot transform more than 136 millions of vertices per second. If an application needs to render several images per second, this limitation is serious, even when an image-based rendering is considered, because it limits the available size of scene. To keep the number of images per second but increase the size of scene, many applications use some sequential techniques to reduce number of vertices needed to represent an object. These include use of efficient data structures, i.e. vertex arays [OpenGL], strips or fans, use of data reduction, i.e. refinement [Pup94] or decimation [Sch92], and use of view dependent level of detail [Hop96]. However, there still exist applications such that even use of all these acceleration techniques is not sufficient. An example of such an application is the walk-through visualization of CAD models.

Besides the performance limitations, there are also other limitations, e.g. the size of textures or the resolution of final images. In cases that a complex scene has to be rendered in a reasonable time (often in real-time) or an image of higher resolution than is supported is required, then a parallel solution using data-parallelism approach is necessary.

There are two possible strategies: *object-parallel* and *image-parallel*. In the first one, the tasks for the processors are formed by partitioning the geometric description of the scene. Rendering operations are then applied in parallel to subsets of geometric data, producing pixel values which must be combined together to form a final image. Since time needed for the rasterization of geometric primitives depends on their size, one processor could finish its task much earlier than another one. Another problem is that the merging step can cause heavy bandwidth demands on memory busses or communications network.

In the *image-parallel* approach, tasks are formed by partitioning the image pixels, and each processor renders only those geometric primitives which contribute to the assigned pixels. No merge phase is required, however, if portions of a single geometric primitive map to several different regions in the image space, it is necessary to exchange them among the processors. It may also result in additional or redundant computations.

To avoid bottlenecks, most parallel algorithms exploit both approaches. Object and image data are partitioned among the available processors. At some point in the rendering pipeline the processors have to exchange their data. Molnar et al. [Mol94] introduced a classification of parallel algorithms as *sort-first*, *sort-middle* or *sort-last*, depending on whether the communication step occurs somewhere in the transformation phase, between the transformation and the rasterization phase or somewhere in the rasterization phase. *Figure 3.4* brings the scheme of these approaches.



Scene
arbitrary distribution of primitives
G
G
...
G
primitives redistribution
R
R
...
R
Display
a) sort-first
Scene
arbitrary distribution of primitives
G
G
...
G
screen-space primitives redistribution
R
R
...
R
Display
b) sort-middle
Scene
arbitrary distribution of primitives
G
G
...
G
R
R
...
R
pixels redistribution, image merge
Display
c) sort-last

**Figure 3.4:** *The principles of three general approaches in polygon rendering data parallelism.*

#### 3.1.2.1 Sort-first

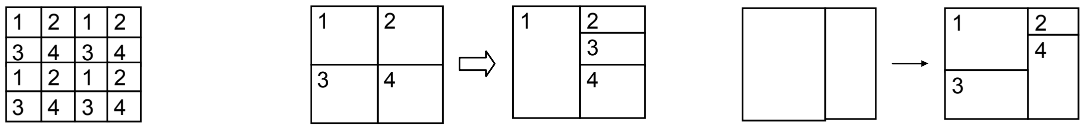
Sort-first algorithms distribute primitives early in the rendering pipeline to processors that can perform the remaining rendering calculations – see *Figure 3.4a*. At the beginning of the computation, object primitives (i.e. polygons) are distributed arbitrarily over available processors. The image-space is partitioned into regions (the number of regions is equal to the number of processors) and the regions are also distributed. Each processor performs enough transformations (generally, bounding boxes of disjunctive groups of primitives are used) to determine into which regions each of its primitive falls. Some primitives will fall into the regions of processors other than the one on which they reside. These primitives has to be redistributed over an interconnect network to the appropriate processors. If we are rendering a single image, almost all the primitives may need to be sent. However, if we render multiple images in an animation or real-time application and the scene does not change too much between two images, only the small number of primitives has to be redistributed. In such a case, sort-first requires much less communication bandwidth than the other approaches.

After the redistribution of primitives, each processor has to perform the entire rendering of its part of scene into its part of final image. Therefore, sort-first algorithms are capable to produce rendering in any resolution. Moreover, the speed of the rendering on each processor can be improved by the use of hardware acceleration of the graphical adapter – indeed, if this adapter is available to the processor. On the contrary, sort-first algorithms suffer from load-imbalance, which reduces the performance. Such an imbalance arises from the distribution of primitives over screen. It is highly probable that some regions of the screen will receive much more primitives than other. The general strategy how to reduce the imbalance is to divide the screen into more regions than there are processors and assign the regions to the processors in an interlaced fashion – see *Figure 3.5a*. Although there is still the non-zero probability that one region will contain several times more primitives than others (no matter how small the regions are), it is unlikely to happen in practice. One problem with this strategy is that it increases the number of the primitives to be redistributed and the number of the shared primitives. A shared primitive is such a primitive that covers several regions and, therefore, it requires duplicated storage as well as computation effort.

An efficient sort-first rendering algorithm needs a fast adaptive partition of the image-space ensuring even distribution of primitives over screen and minimizing the number of shared primitives. It is impossible to find an optimal partition that meets all these three goals and, therefore, many heuristic methods have been developed.

Roble [Rob88] starts with a standard rectangular decomposition. According to the number of primitives in each region, lightly loaded regions are combined together and highly loaded regions are split in half – see *Figure 3.5b*. As the algorithm does not consider the sizes of the primitives, one region may contain much larger primitives than another one. It reduces the efficiency of the rendering because the time consumed in the rasterization is dependent on the size of the primitives. Moreover, it is possible that the most of primitives falls on one side of the split, no matter how many iterations of the algorithm are performed. Therefore, even distribution of primitives over the screen is not generally ensured.

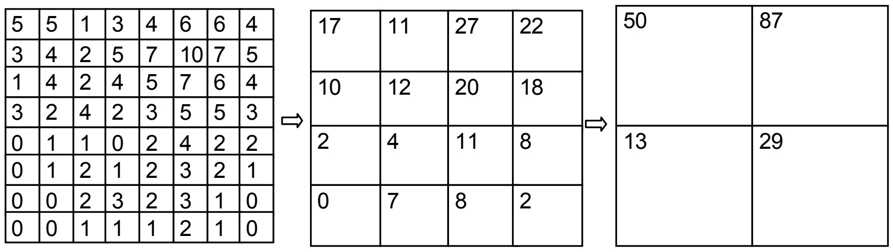
Whelan [Whe85] divides the longer dimension of the screen into two parts in such a manner that both regions contain the same number of primitives. To ensure even distribution of primitives, a median cut based on the centroids of the primitives is used. The regions are recursively partitioned until the number of regions equals to the number $n=2^k$ of available processors. One problem with this approach is that if there is large number of primitives, the computation of the median (even if an approximate method, e.g. [Bat99], or parallel approach [Har97] is used) consumes quite a lot of time. As the sizes of the primitives are again not considered, load-imbalances in the rendering may occur. *Figure 3.5c* shows a screen partition by Whelan's algorithm.



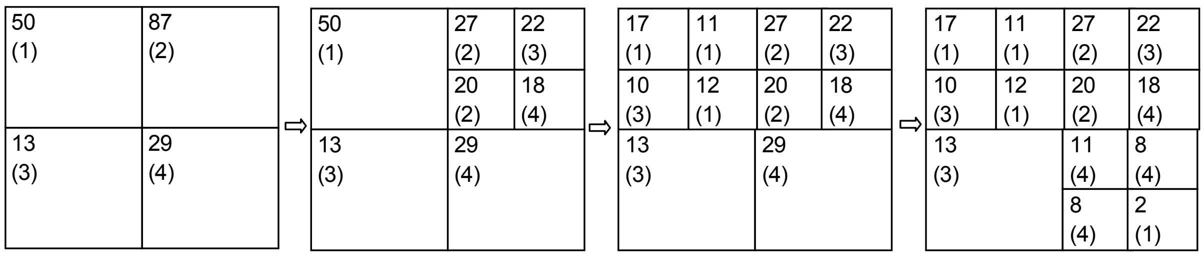
*a) static - 4 PEs* *b) Roble - 4 PEs* *c) Whelan - 4 PEs*

**Figure 3.5:** *Classic image-space partition.*

Another strategy is in Whitman [Whi92]. Firstly, the image-space is subdivided into fine mesh similarly as in the static method. Each cell of the mesh is capable to store an integer counter *C* and a list *L* of primitives. At the beginning the counter is set to zero and the list is empty. Then bounding boxes of primitives are found [Wat00, Mue97] and they are tested against the cells. If a bounding box overlaps (i.e. fully or partly covers) a cell, the bounded primitives are inserted into the list *L* and the counter *C* is increased by one. When all bounding boxes are processed, the algorithm starts to create a hierarchical tree structure over the mesh. Each node in the tree contains an integer counter $C_{t+1}$ storing a sum of the counters $C_t$ (or *C* at the level one above the mesh level) of all node children. *Figure 3.6a* shows an example of several levels in the quadtree, the mostly used hierarchical structure.



*a) Whitman - several last steps of the creation of quadtree, the value in a cell counts the number of primitives overlapping this cell.*

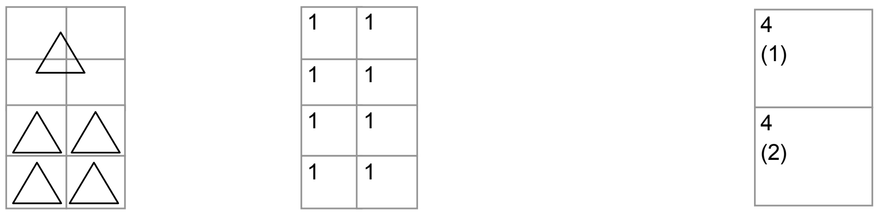


*b) Whitman - several first steps of the subdivision into regions to be used by 4 PEs.*

**Figure 3.6:** *Whitman's image-space partition*

The generated tree is then traversed top-down. At the beginning all nodes are collapsed and the region described in the root (i.e. the whole image-space) is assigned to the first processor. In each step, the collapsed node with the largest value of $C_t$ is unassigned, then it is expanded and the regions described in its children are successively assigned to processors. We assign always the region with the largest number of primitives to the processor with the minimal work load. The algorithm stops, when the number of leaves in the expanded sub-tree, i.e. tree consisting of expanded nodes only, is ten times the number of processors. Each leaf in the expanded sub-tree describes one region to be assigned to one processor. *Figure 3.6b* show an example of the tree expanding. Numbers in round brackets identify the processor that will receive primitives stored in that region.

Although Whitman's algorithm considers the sizes of the primitives, work load-imbalances may occur even if the sum of counters $C_t$ of processor's regions is the same for all processors. The reason is that work may be overestimated due to double-counting of primitives. *Figure 3.7* shows an example of this problem.



*a) five identical triangles* *b) cells evaluation* *c) two regions to be assigned, the second processor will consume four times more time*

**Figure 3.7:** *The problem with double-counting of primitives.*

Mueller [Mue95] combines Whitman's and Whelan's approaches. His MAHD algorithm starts with the evaluation of the cells of a fine mesh. Unlike Whitman's method, the counter $C$ is of a float data type and it is increased by a value inversely proportional to the number of cells that a bounding box of primitive overlaps. It solves the problem with double-counting of primitives. In the next step, a summed-area table of the mesh is computed. Summed-area table $T_s$ of a table $T$ is such a table that the value of its cell at the position [i, j] is equal to the sum of the values of the cells in $T$ at positions [0 up to i, 0 up to j]. *Figure 3.8* shows an example of the mesh and its summed-area.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 3 | 2 | 1 | 0 | 1 | 0 | 0 |
| 3 | 6 | 5 | 4 | 3 | 2 | 2 | 2 | 2 | 0 |
| 4 | 7 | 6 | 5 | 5 | 4 | 3 | 2 | 3 | 2 |
| 6 | 8 | 9 | 7 | 7 | 6 | 4 | 3 | 2 | 1 |
| 6 | 5 | 6 | 4 | 3 | 2 | 2 | 1 | 2 | 2 |
| 6 | 5 | 4 | 3 | 2 | 1 | 1 | 1 | 2 | 1 |
| 7 | 6 | 5 | 2 | 1 | 1 | 0 | 0 | 1 | 1 |
| 8 | 9 | 7 | 2 | 1 | 0 | 1 | 0 | 0 | 0 |
| 6 | 8 | 9 | 3 | 2 | 2 | 1 | 0 | 0 | 0 |
| 6 | 5 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 0 |
| 6 | 5 | 4 | 4 | 6 | 4 | 3 | 2 | 1 | 0 |
| 7 | 6 | 5 | 4 | 2 | 2 | 1 | 1 | 1 | 1 |
| 6 | 6 | 6 | 5 | 4 | 3 | 2 | 0 | 1 | 1 |

*a) mesh*

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 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 |

*b) summed-area table*

**Figure 3.8:** *First stage of Mueller's MAHD algorithm*

The summed-area table is then recursively subdivided into required number of regions. The subdivision is similar to the one in the Whelan's method, however, in the case of the Mueller algorithm, it is very efficient thanks to the summed-area table and it allows using of any number of processors. Let us explain the subdivision by way of an example. We will divide the mesh from *Figure 3.8a* into five regions. In the first step, we divide the summed-area table into two parts in ration 3:2. Therefore one horizontal cut such that the first part of the summed-area consists of 3·409/5 = 245.4 primitives, has to be found. First, the position of the value 245.4 is found in the one-dimensional array 13, 42, up to 409 by the binary-search algorithm. Second, 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 3.9a* shows. Next three steps are similar to the just described one – see *Figure 3.9b, c, d.* After the subdivision the work-loads are 89, 75, 83, 91 and 71, i.e. relatively evenly distributed.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 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 |

*a) after the first step*

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 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 |

*b) after the second step*

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 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 | 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 third step*

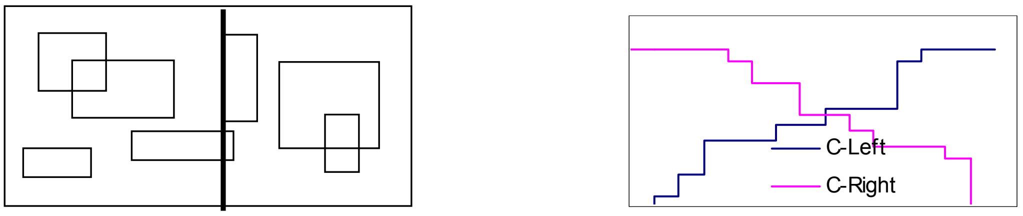
| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 3 | 6 | 2 | 3 | 3 | 4 | 4 | 4 |
| 4 | 12 | 8 | 15 | 5 | 8 | 10 | 13 | 15 | 15 |
| 8 | 23 | 14 | 26 | 10 | 17 | 22 | 27 | 32 | 34 |
| 14 | 37 | 23 | 42 | 17 | 30 | 39 | 47 | 54 | 57 |
| 20 | 48 | 29 | 52 | 20 | 35 | 46 | 55 | 64 | 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) final subdivision*

**Figure 3.9:** *Second stage of Mueller's MAHD algorithm*

One of many variations of the Mueller's algorithm called Graph Partitioning Based Subdivision (GS) is described in Kurc et al. [Kur97]. In the GS, each edge between the cells is evaluated proportionally to the number of primitives crossing this edge and the algorithm tries to form the region in such a manner to minimize the number of shared primitives. Unlike the Mueller's algorithm, generated regions may not be rectangular regions and, moreover, several isolated regions may be assigned to one processor. According to the experiments published in [Kur97], the GS approach achieves better results than the Mueller's MAHD.

[Sam99] introduces the KD-split algorithm similar to Whelan's approach. Firstly, axis-aligned bounding boxes of primitives are found. The region is split by a sweep-line into two parts. The sweep line starts at left side of the region and, therefore, the cost of the rendering of the left part is zero and the cost of the rendering of the right part is equal to the cost of rendering all primitives. The vertical edges of the bounding boxes are sorted and the sweep-line moves iteratively to the right side of the region stopping in each of these edges. Whenever the left edge of a box is reached, the cost of the left part is increased by the cost of the rendering of the primitives bounded by the reached box. Whenever the right edge of a box is reached, the cost of the right part is decreased by the same way. The cut is chosen at the location where the functions of estimated costs intersect themselves. The algorithm continues recursively until the required number of regions is generated. Let us note that the regions processed at even levels of the recursion are usually splits by a horizontal sweep-line. *Figure 3.10* shows an example of the KD-split of a region.



*a) bounding boxes in the region and the optimal position of the cut*

*b) estimated costs of the rendering of the left and the right part*

**Figure 3.10:** *KD-split algorithm - one step.*

The survey of image-space partitioning algorithms just given in the text above is not exhaustive. There exist other algorithms [Sam00, Sam99, Kur97 and El194], mostly some variations of the presented, but they are not so commonly used. Moreover, it is not a goal of this thesis to investigate polygon rendering in detail. Therefore, let us omit their description.

Some parallel polygon rendering algorithms are not fully satisfied with any adaptive partitioning used at the beginning of the rendering and, therefore, they add some advanced load-balancing features also during the rendering process [e.g. El194]. Typically, if a processor finishes its work and there is still some overloaded processor, it will 'steal' half of work from the overloaded processor. Let us note that it is necessary to implement some mechanism avoiding that several finished processors simultaneously will steal all work.

If a sequence of images (frames) has to be rendered a new issue arises. The region boundaries can be recalculated by the algorithm in each frame or only when it is needed. In the second case, the boundaries are retained until one region is the bottleneck, i.e. it has to process more than *1/n* of the entire workload, where *n* is the number of the processors. It is more difficult to be implemented but it usually leads to higher performance of the rendering.

### 3.1.2.2 Sort-middle

Sort-middle strategy distributes the object primitives arbitrarily among the available processors at the beginning of the computation and subdivides image-space into the regions as it was in the sort-first approach. Each processor performs the whole transformation phase, i.e. geometry processing is performed and the primitives are converted into the screen-coordinates, then it classifies the resulting screen-space primitives and sends them to the appropriate processors to be rasterized. Let us recall the sort-middle rendering pipeline in *Figure 3.4b*.

Sort-middle algorithms are simple and straightforward. One problem with sort-middle is the high communication cost, proportional to the number of generated screen-space primitives. There are at least two examples when many screen-space primitives have to be generated. First, some rendering systems (e.g. RenderMan [Ups89]) generate several screen-space primitives to be distributed for each object primitive in order to generate images in higher quality. Second, in the image-based rendering by warping (IBRW), the number of distributed primitives is at least twice the number of output pixels.

In a sort-middle algorithm, all processors may perform both the geometry processing and rasterization or the algorithm could divide the processors into two groups: one group performing the geometry processing and the second group performing the rasterization. Keeping the processors all in one group reduces the possibility of load-imbalance by the uneven distribution of primitives over screen. Although the imbalance is not as significant as in the sort-first, it is still problem to be faced. On the other hand, splitting the processors into two groups could reduce overhead in the algorithm and it reduces the redistribution.

Now, let us discuss the possibilities of the redistribution. The straightforward solution is to redistribute the primitives after all processors have finished their transformations. In such a case, the sort-middle algorithm can easily adopt some sort-first adaptive algorithm for image-space partition to reduce load-imbalance that could occur during the rasterization phase. On other hand, this solution introduces extremely high traffic and, therefore, it is often useless for rendering of a complex scene.

Another possibility is to let a processor redistribute its generated primitives immediately. If a processor redistributed all its generated primitives at one time, it would not bring any substantial improvement to the first solution. Therefore, the processors send their primitives to the appropriate rasterizers in groups during the entire transformation phase. In such a case, the image-space is, typically, divided into more regions than there are processors and the regions are assigned to the processors in an interlaced fashion – see *Figure 3.5a*.

#### 3.1.2.3 Sort-last

Let us recall *Figure 3.4c* showing the sort-last rendering pipeline. At the beginning of the computation, the sort-last algorithms distribute the object primitives arbitrarily among the available processors. Each processor performs the entire rendering of its assigned primitives, i.e. it processes the geometry of the object primitives, convert them into the screen-coordinates and rasterizes the screen-space primitives to produce pixels of final image. No object primitive is processed redundantly by more processors as it appears in sort-first or sort-middle approaches. Similarly to the sort-first, sort-last strategy can use hardware acceleration of graphical adapter.

To obtain the final image, it is necessary to merge the generated pixels together and, therefore, their colors and depths in the z-buffer have to be distributed among the processors. Let us note that the transparency handling is not an easy task in sort-last algorithms and, therefore, many of them do not support it. Moreover, rendering systems usually uses anti-aliasing and if it is performed by super-sampling, i.e. a several times larger immediate image is generated and then it is filtered down to get the required image resolution, the pixel traffic may be extremely high.

Let us discuss two possibilities how to perform the merge phase. In the first one, the processors send only the generated pixels. It minimizes communication cost in the merge phase, however, network traffic and merging can be imbalanced if more pixels are sent to one compositor than to another. Another possibility is to send full image containing the generated pixels. In such a case, load imbalances are usually insignificant.

The merge phase influences significantly the efficiency of the entire sort-last algorithm. [Tay02] compares six commonly used approaches for the image composition: centralized, pipeline, tree, hypercube, mesh and bus. In the centralized approach, there are *n* renderers and just one composer – see *Figure 3.11a*. In each of *n* steps, one renderer sends its immediate image to the composer, which merges the received image with the previously stored image. If only generated pixels are sent, the total amount *B* of transferred pixels and the total time $L_B$ required by a renderer to send its pixels are independent of the number of renderers. On other hand, while the composer is heavily loaded for *n* steps, the activity of a renderer is limited to only one step. Therefore, the total time $L_C$ required by a composer to perform all its work grows linearly with the increasing number of renderers, which results in poor scalability.

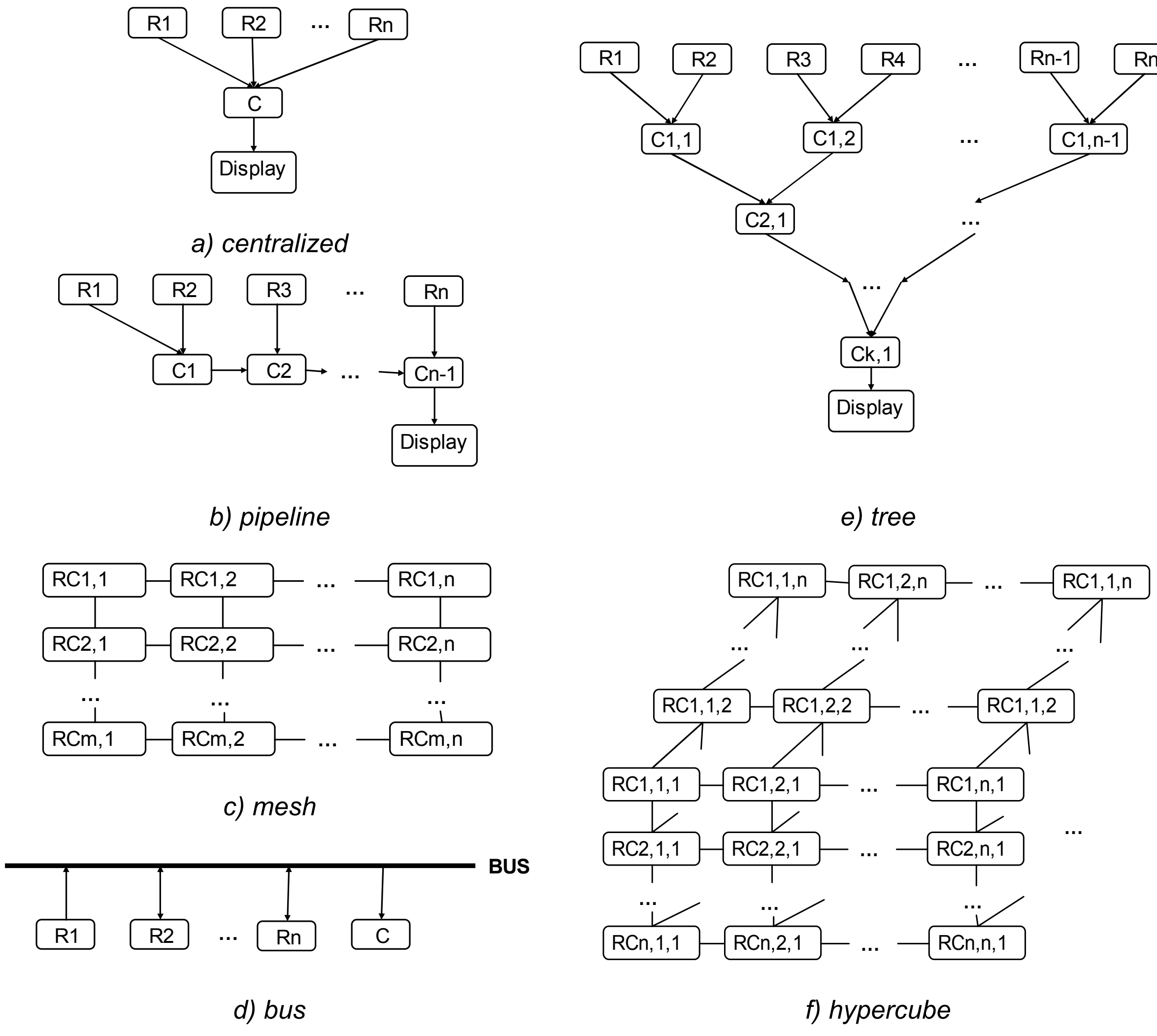
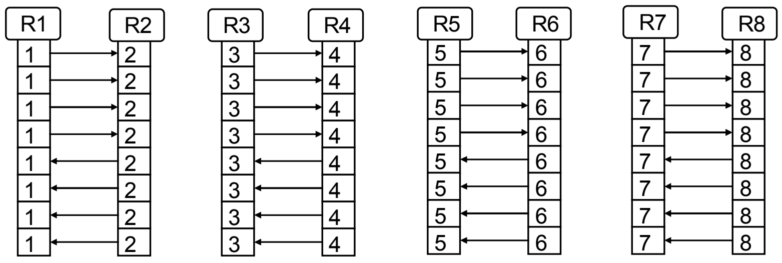


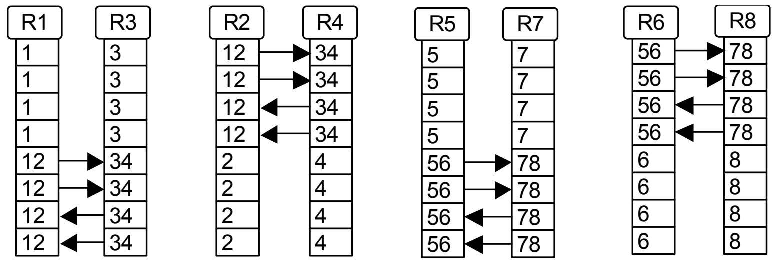
**Figure 3.11:** *Basic approaches for the image composition. Legend: R - renderer, C - composer, RC - renderer + composer in one PE.*

In the pipeline approach, there are *n* renderers and *n-1* composers. In each of *n* steps, one renderer and one composer (another renderer in the first step) send their generated pixels to the appropriate composer that merges the pixels together. *Figure 3.11b* shows a scheme of this approach. All characteristics, i.e. *B*, $L_B$ and $L_C$ depend linearly on the number of renderers. However, pipeline approach achieves almost linear scalability when a sequence of images has to be rendered. In such a case, full-screen immediate images are typically used. As the efficiency for a sequence of images is unreachable by any other approach, many parallel systems adopted the pipeline technique, e.g. [Mol92, Sch88, Wei81 and Rom79].

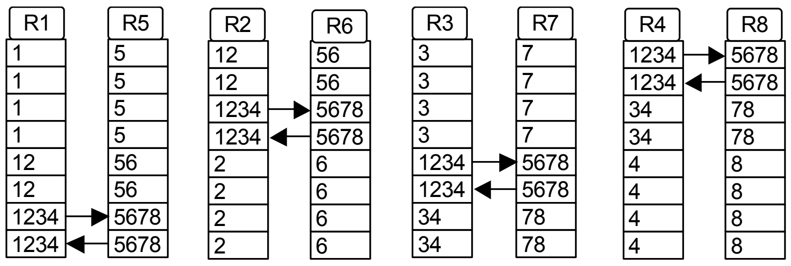
Tree is a generalized pipeline approach. It consists of $n=2^k$ renderers and $n-1$ composers. In each of *k* steps, two composers (two renderes in the first step) send their pixels to the appropriate composer that merges the pixels together – see *Figure 3.11e.* All characteristics, i.e. *B*, $L_B$ and $L_C$ grow with a logarithmic trend according to the increasing number of renderers, however, tree approach achieves lower efficiency than the centralized or pipeline approaches if only several renderers are considered. Despite it, this strategy is used in many parallel systems, e.g. [Ran97, Sco93, Mol91, Li91, The89, Sha88 and Fus82]. Let us note that the best performance is achieved when a sequence of images has to be rendered on a parallel architecture with processing elements inter-connected to a binary tree.



*a) step 1*



*b) step 2*



*c) step 3*

**Figure 3.12:** *Three steps of hypercube composition approach. The numbers in each strip of image identify the already merged renderers' parts; the arrows show strip to be send. Legend: R – renderer.*

The hypercube contains $n=2^k$ processors. Each processor in this architecture (see *Figure 3.11f*) performs rendering as well as composition. After the rendering, the processor subdivides its immediate image into *n* strips and in each of *k* steps of the composition phase, the processor sends at least one strip to another processor and merges the received strips with its appropriate strips. After *k* steps, each processor retains one strip of the final image. If hypercube approach is used within the n-Cube parallel architecture, the inter-processor communication is done only between neighbors and, therefore, the best performance is achieved. When the number of renderers increases, the characteristics $L_B$ and $L_C$ decrease with a hyperbolic trend while the total amount *B* of sent data grows logarithmically (the growing is slower than in the tree approach sending full-screen images). *Figure 3.12* shows an example of the composition on the system with 8 processors. Hypercube approach or its variations are used, for example, in [Ude99, Kur98, San97, Han96, Kar94, Ma94 and Mac92].

Mesh approach, which is similar to the hypercube approach, suits well for parallel architectures with the processing elements structured into two-dimensional grid of *n* columns and *m* rows – see *Figure 3.11c*. The immediate images are partitioned evenly into *n·m* regions. The composition phase consists of two stages. In the first stage, which takes *m-1* steps, horizontal strips (i.e. blocks of *n* adjacent regions) are exchanged among the processors placed in one column. *Figure 3.13* shows an example of the first stage for 3×4 processors.

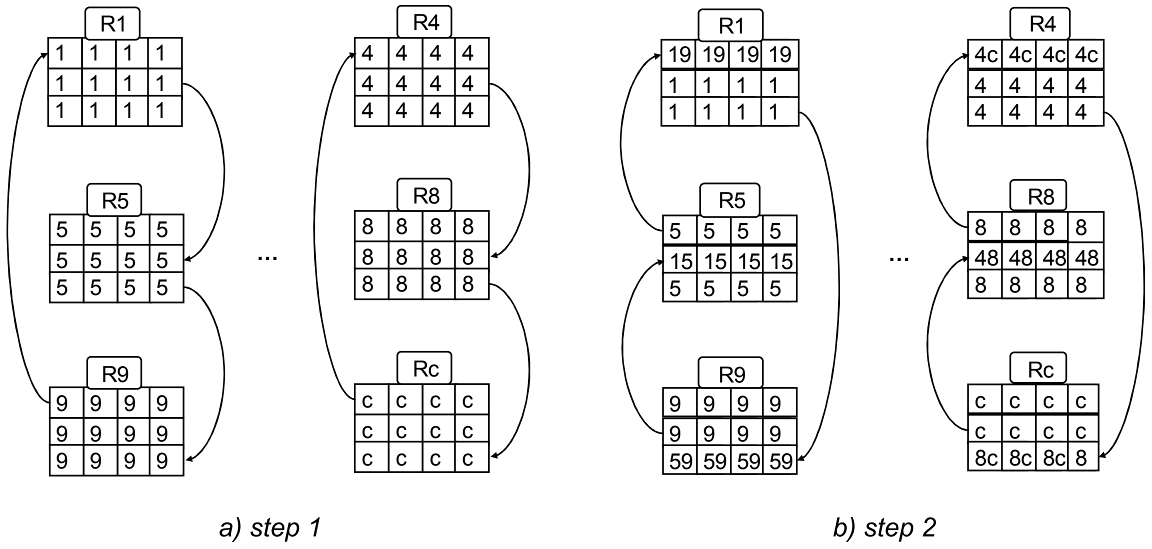


**Figure 3.13:** *The first stage of mesh composition approach. The numbers or letters in each region of image identify the already merged renderers' parts; the arrows show horizontal strip to be send. Legend: R – renderer.*

In the second stage consuming *n-1* steps, regions are communicated among the processors placed in one row. *Figure 3.14* shows an example of the second stage for 3×4 processors. After *n+m-2* steps, each processor retains one region of the final image.

All characteristics, i.e. *B*, $L_B$ and $L_C$ decrease with a hyperbolic trend (the decrease for $L_B$ and $L_C$ is faster than for the hypercube) when the number of renderers increases. Despite it, as far as the author knows, the mesh approach is used rarely, e.g. in [Kur98 and Lee96].

The bus approach assumes that *n* renderers and one composer are connected together via a shared bus – see *Figure 3.11d*. The architecture has to support broadcasting, i.e. data sent by one node (renderer or composer) can be received simultaneously by all nodes. The well-known instance of such an architecture is any cluster of workstations. After the rendering, each renderer retains a prepared set of generated pixels to be merged together. There are *n* steps of the image composition. In the first step, the first renderer broadcasts its generated pixels. The broadcasted pixels are received by other renderers and the composer. While the composer simply stores them, the renderes have to check their sets of pixels against them: if the depth of a pixel in the renderer's set is larger than the depth of the appropriate pixel in the received set, the pixel is removed. In the second step, the second renderer broadcasts its reduced set of pixels. The composer updates the final image while the third up to *n*-th renderer once again check their sets. After *n* steps, the final image is available at the composer. As the characteristics *B* and $L_B$ decrease with a hyperbolic trend when the number of renderers increases and, moreover, the total amount of transferred pixels (*B*) is the lowest of all here presented approaches, it can be used even in the case that the processors are inter-connected by low-cost slow links (e.g. Ethernet). On the other hand, the total time $L_C$ required by a composer to perform all its work grows with the increasing number of renderers. As the amount of pixels to be processed is smaller and smaller in each step, the growth shows a logarithmical trend. More details about this approach can be found in [Cox94].

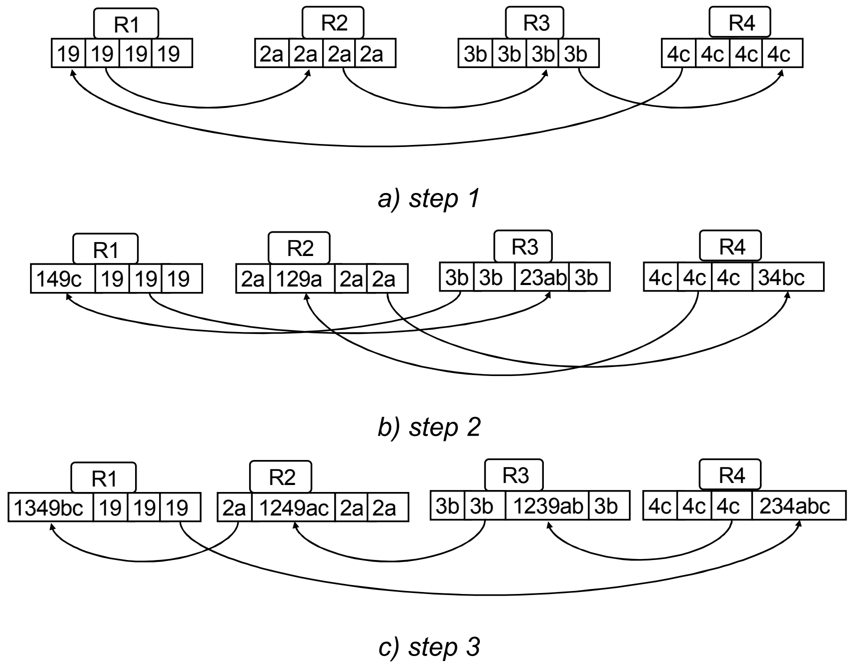


**Figure 3.14:** *The second stage of mesh composition approach. The numbers or letters in each region of image identify already merged the renderers' parts; the arrows show region to be sent. Legend: R – renderer.*

### 3.1.3 Hardware and Software Solutions with Data Parallelism

Many hardware or software solutions were already mentioned in the previous text. In this subsection, which was written mainly according to the information published in [Cha02, Eld01, Ell96], we describe several famous hardware or software solutions in more detail.

The Stanford WireGL [Hum01] is a group of several software components based on the sort-first approach. WireGL is determined to be run on a cluster of workstations. The most important component of the system is a renderer. The renderer runs on the workstation with a graphical adapter connected to a display device. Each renderer processes OpenGL [OpenGL] compatible commands to get its part of the final image. Full hardware acceleration is used. The display devices are placed in a matrix forming a tiled display.

Another component in the system is an OpenGL compatible software layer. This component is used by the client application. It is responsible for the sort-first stuff, i.e. it contacts the renderers and sends them the appropriate group of OpenGL commands. The WireGL architecture allows the simultaneous rendering of the data to more client applications. Therefore, you can develop a distributed algorithm to improve the frame rate; each computing node in the cluster will render just its data.

The WireGL contains also one very useful component. This component takes as an input an already existing binary application that uses the OpenGL API and allows this application to render on a tiled display depending on the configuration of WireGL.

It allows users to build a graphical system capable of handling demanding real-time or high resolution tasks on the computers interconnected with Myrinet or TCP/IP compatible network technology. The disadvantages of the current version are that it does not support dynamic load-balancing and that the resolution of the final image is determined by the number and the hardware resolution of the used display devices.

Correa et al. [Cor02] present a sort-first parallel system for a cluster of workstations with a tiled display. Their system has been designed for the rendering of large models. Given a model, a pre-processing algorithm is used to build an on-disk octree hierarchical representation of the model. At run-time, each processor renders the image for its display device using multiple threads to perform the fetching of primitives from the disk, the visibility computation and the rendering. The Message Passing Interface – MPI [MPI] is used for the communication. Unlike theWireGL, this system allows also the inspection of a model (i.e. a walk-through). The proposed solution is able to render more than 10 frames per second of the UNC Power plant model containing 13 millions of triangles in the resolution 4036x3072 pixels on a cluster of 16 computers with Athlon 900MHz processor and 512 MB of memory.

Samanta et al. [Sam99] describe other sort-first architecture for clusters. Each renderer is, as usually, equipped by a graphical adapter connected to a display device. Unlike the previously described systems, the division of the image-space is not predetermined by the number and the resolution of these display devices. The subdivision is much finer and the renderer has to process several distant regions. Therefore, the renderers have to communicate the generated pixels that should be displayed elsewhere. As the entire database of the primitives is replicated on each node, there is no other necessity of the communication.

In their more recent work [Sam00], the authors try to minimize work load imbalances by integrating some sort-last features. According to the published results this hybrid approach outperforms sort-first and sort-last algorithms in the parallel scalability allowing larger numbers of processors and higher screen resolutions.

Probably one of the most-known examples of a specialized multiprocessor for polygon rendering is the series of Pixel-Planes machines developed at UNC-Chapel Hill. Pixel-Planes 5 [Fuc89] is sort-middle architecture. The available processors are split into two groups: graphics processors and renderers. All processors are connected to a ring network capable of handling eight messages simultaneously. The image-space is subdivided into 128x128 pixels regions. Pixel-Plane 5 implements simple load-balancing: after the transformation phase, the regions are successively assigned to rasterizers in such a manner that the region with the highest estimated cost of rendering all its primitives is assigned to the processor with the lowest work-load. Let us note that the proposed inter-processor communication limits significantly the efficiency of the architecture.

The SGI InfiniteReality [Mon97] is a sort-middle architecture that uses bus-based broadcast communication to distribute primitives. The image-space is subdivided into fine regions that are assigned to processors in an interlaced fashion.

Sort-last systems have existed in various forms for more than thirty years. UNC Pixel-Flow architecture designed by Molnar et al. [Mol92] is one of the famous sort-last systems. The processing elements are inter-connected in a pipeline fashion. The shading and texturing are not applied until the visibility of all primitives is calculated. Therefore, it is necessary to exchange z-buffers among all renderers before the shading may start. On the contrary, the rendering is faster because the complexity of shading is independent of the complexity of the scene. Moreover, it allows using custom shading algorithms. Pixel-Flow hardware implementation is modular, two or more chips could be easily connected together to increase the rendering speed. According to the results presented in [Mol92], the Pixel-Flow architecture shows almost linear speed-up.

## 3.2 Ray-tracing

A scene for ray-tracing is a collection of objects that are represented by sets of polygons, parametric patches (e.g. NURBS), CSG or implicit functions. Starting at the camera (or eye) position, one or more rays are casted through each pixel of the final image backward into the scene. The color value of the pixel is determined by the color values of the rays that were casted through this pixel.

The kernel of the ray-tracing algorithm is a recursive function returning a color value of a given ray. The given ray is tested against all objects in the scene to determine if it intersects any object. If the outcome is negative, the appropriate color of the background is returned. Otherwise, reflected and transmitted rays are computed and the function is called recursively to get their color values. Let us note that the existence of these rays depends on the material properties assigned to the object; most surfaces of the objects are reflective only. Then, for each source of the light in the scene, a shadow ray directing to that point light is computed and the color value of this ray is evaluated. The evaluation depends whether the shadow ray intersects any object, i.e. whether the origin of the ray is illuminated by the light, it depends also on the color of the light and on the material and the texture of the object. The function returns a sum of the values of reflected, transmitted and shadow rays. An example of the ray-tracing process is given in *Figure 3.15*. The letters *P*, *R*, *T* and *S* denote the primary, reflected, transmitted and shadow rays in the scene.

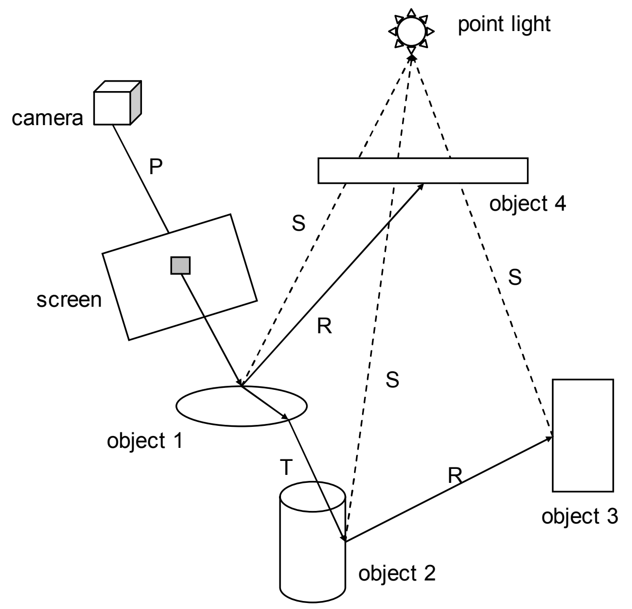


**Figure 3.15:** *The principle of the ray-tracing algorithm.*

Wald et al. [Wal01] in their report mention many advantages of the ray-tracing approach. Probably the well-known feature of ray tracing is that it computes physically correct reflections, refractions and shading. Therefore, highly realistic images are produced. While the traditional polygon rendering processes the geometry of all primitives and then discard the non visible, the ray-tracing processes the geometry of visible primitives only. Moreover, if a simple search data structure is used, the logarithmic complexity in the number of primitives is achieved which enables an efficient rendering of complex scenes. Similarly, ray-tracing shades the screen-space primitives after the visibility has been determined which is in the contrast with the polygon rendering (if we ignore the deferred shading approaches). In the ray-tracing, programming shaders (e.g. for the RenderMan [Ups89]), which create special lighting and appearance effects, is straightforward.

One problem with the ray-tracing is that it is computationally very intensive and, therefore, it consumes a lot of time. There are plenty of possibilities how to accelerate the rendering. Much research has focused on using acceleration structures [Watt00], e.g. BSP trees, octrees, hierarchical grids or bounding volume hierarchies.

Another possibility is to combine polygon rendering and ray-tracing [Sta00]. One of the simplest solutions assigns a unique color to each object in the scene and then uses standard graphics hardware means to render the scene from the positions of the camera and light sources. Using these images accelerates the tracing of primary and shadow rays because the object to be intersected by a ray is simply identified by the color. There are two problems with this approach. First, it suffers from the potential occurrence of artifacts caused by the discretization. More serious problem is that it is useful only if rasterization hardware is considerably faster than the ray tracer. However, Wald et al. [Wal01] show that their implementation of ray-tracing runs faster than high-performance graphical adapters for complex models.

The image based rendering mixed with the ray-tracing [Lar98, Par99] is not the least possibility how to speedup the rendering. In this approach, a new image is approximated with information from previous images and some new rays are traced in order to update this new image as time allows. The disadvantage of this approach is that it leads to noticeable artifacts in dynamic situations.

### 3.2.1 Data Parallelism in Ray-tracing

Ray-tracing is relatively easy to be parallelized because of its inherently parallel nature. This is definitely true especially in the case when the entire description of the scene can be duplicated on every processor. In such a case each processor computes independently a part of the final image. As different surfaces scatter different amounts of rays, i.e. different amount of computation needed, the cost to compute individual pixels can vary dramatically. Therefore, the efficiency of the algorithm is dependent on efficient load balancing. There are two main possibilities of load-balancing: static and dynamic (see Section 2).

Heirich et al. [Hei97] compare various strategies for the load balancing. Their results show that any static subdivision of the image-space suffers from serious load imbalances. The subdivision into more regions than there are processors and assigning these regions to the processors randomly or in an interlaced fashion (see *Figure 3.5a*) does not help. The situation is rarely much better when the regions are as small as one pixel.

Badouel et al. [Bad90] describes a dynamic load balancing scheme based on the master-slave model. There is no image-space subdivision at the beginning, i.e. no task is assigned. When a slave becomes idle, it contacts the master to get a task of 3x3 adjacent pixels. Another dynamic balancing of master-slave kind was proposed by Pandzic et al. [Pan95]. In their approach, the size of each task is proportional to the workload left to be computed and inversely proportional to the number of processors. In this way it is ensured that the tasks get smaller as the work proceeds, with the effect of fine-tuning the load balance towards the end of the processing. The smallest possible task contains just one line of pixels.

A dynamic load balancing strategy based on master-slave scheme is simple and efficient but not scalable. The bottleneck of the system is the central processor (master) because its work load grows with the increasing number of the slaves. Therefore, Green et al. [Gre90] uses a hierarchy of masters to improve the scalability.

More efficient dynamic load balancing is based on the work stealing strategy. In this strategy, the image-space is initially statically subdivided and assigned to the processors. Whenever a processor becomes idle, it sends a query to all processors in order to determine their work load. The work load is quantified as a number of pending rays. After receiving all status reports, the first processor sends a request to every processor that reported more than is the average work load. Each of the contacted processors removes a designated amount of pending rays and sends it to the calling processor which places the received rays into its own queue of pending rays.

The problem with the general work stealing strategy is that a processor has to wait until the rays are not received and as the time required for the sending of rays depends on the physical distance of the communicating processors, it might take a relatively long time. An improved load balancing strategy was proposed by Heirich et al. [Hei97]. A processor sends a query in advance, i.e. when it detects that the tracing of just few rays remains. Moreover, only the neighboring processors are contacted which leads to the high efficiency if the ray-tracing runs on a n-Cube parallel architecture. Similar solutions are described in [Cor97, Wil93].

If data have to be distributed over local memories of the parallel computer processors, communications between processors cannot be avoided. For such inter-processor communication two principal strategies have been proposed: object dataflow and ray dataflow. In the object dataflow, when a computation cannot be performed because of a missing object, this piece of data is fetched and then copied in the local memory of the processor in order to complete its calculations. In the ray dataflow, when a computation cannot be performed because of a missing object, the traced ray is sent to the processor storing the missing object and the computation is performed by the contacted processor. Various dynamic load balancing strategies are described in [Žar95, Bad94]. Now, let us to describe few interesting approaches on the field of the distributed ray-tracing.

One way of fetching a missing object in object dataflow is the use of shared virtual memory [Kea94, Bad90]. In such a case, the pixels redistribution strategies just described above are still useful. In the approach by Badouel et al. [Bad90], the description of the scene, i.e. the objects, are distributed at the beginning over the processors without any particular strategies. Each processor may access any object through the memory management. This management detects whether the demanded object is in the local cache. If the outcome is positive, the pointer on the object is returned, otherwise, the processor sends a request to its appropriate counterpart to get this object. When the processor receives the object, it stores the object in the cache. If the cache is full (i.e. it is unable to store a new object), the least recently used object is removed. As the search for this object is performed during the communication, it does not take any additional time.

The algorithm by Caspary et al. [Cas89] represents a group of ray dataflow algorithms. It starts by organizing the objects in a hierarchical tree of bounding volumes, a well-known technique to accelerate the tracing. The basic idea is to cut the tree to produce a two-level object-space partitioning. The upper part of the tree including the root node is replicated on every processor, while the sub-trees below the cuts are distributed among the processors. *Figure 3.16a* shows a scheme of this system. The key issue is where to cut the tree. If the cuts are too high in the tree, the number of sub-trees will be small, and load balance will be poor. If cuts are too low in the tree, much of the object data will have to be replicated, limiting the size of scenes which can be accommodated.

A similar approach is presented by Jevans [Jev89]. The object-space is subdivided into the voxels and the structure of this subdivision is broadcasted to all processors. Then the voxels within their objects are distributed over the processors in random fashion. *Figure 3.16b* shows a scheme of this approach.

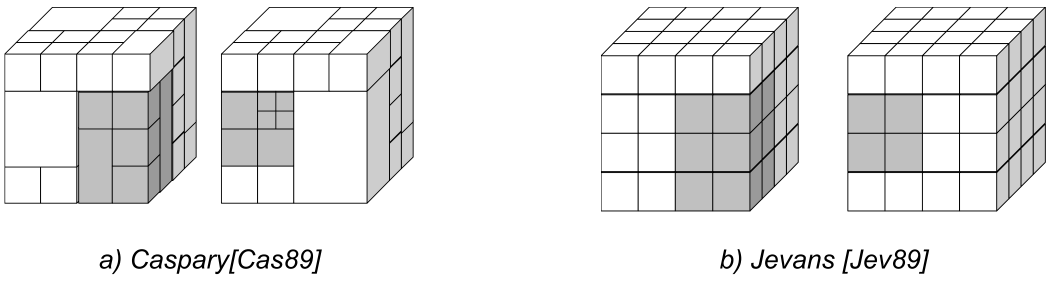


**Figure 3.16:** *The comparison of hierarchical structures by Caspary and Jevans on two processors. The gray cells of space contain the object stored on the processor.*

An alternative solution is resented in Kim et al. [Kim96]. The ring based network topology is assumed. In such a case, the communication is pipelined. The objects are distributed randomly over the processors. A ray will then have to be passed from processor to processor until it has visited all the processors. Load balancing can be achieved by simply moving some objects along the pipeline from a heavily loaded processor to a less busy processor.

Nebel [Neb98] describes a hybrid dataflow approach. Each processor can exchange both objects and rays at any moment. The key issue is to determine which kind of dataflow to use to get optimal efficiency. In order to achieve this goal, each processor needs to know the work load of every processor, i.e. the number of still not processed rays, the list of rays that cannot be longer traced without a communication and the list of all the required remote objects. The information about the work load is sent within the rays or the objects. If a processor finds out that it is loaded more than is the minimal load, then it may send the rays from the first list to the appropriate processors. The group of rays supposed to belong to some processor is submitted only in a case that the number of rays in this group is sufficient enough and the receiving processor is loaded less than the sending processor. Afterwards, the first processor determines the remote objects that are required by a sufficient number of rays and asks for them. Nebel's approach shows higher efficiency than traditional data flow algorithms. Moreover, this approach substantially reduces communication flows and, therefore, it allows using massive parallelism.

### 3.2.2 Hardware and Software Solutions with Data Parallelism

Many software solutions were already mentioned in the previous text. In this subsection, we describe several interesting hardware or software solutions in more detail. The first famous architecture for the interactive ray-tracing was proposed by Muuss et al. [Muu95]. The authors required to simulate the airplane and missile sensors and their automatic target recognition system. The model contained realistic outdoor scenes modeled with several thousands of CSG primitives, which correspond to several millions of polygons when transformed to the polygonal representation. In order to achieve about 1-2 images in video quality per second, the authors used SGI PowerChallenge architecture with 96 processors and the shared memory.

Another interactive ray-tracing system, called Utah, appears in Parker et al. [Par99]. Again a shared-memory computer is assumed. The implementation of their ray tracer is carefully optimized for SGI Origin 2000 supercomputer, e.g. it takes advantage of fast synchronization available on this supercomputer. The system supports also direct rendering of volumetric data, realistic shadows and image based rendering. An optimized static load-balancing scheme is used. The sizes of the tasks are multiples of the hardware cache-granularity in order to achieve higher speed-up. The proposed system is efficient, e.g. the rendering of the scene containing 35 million spheres ran at approximately 15 images in the resolution of 512x512 per seconds using 60 processors. Moreover, the system showed also near-linear scalability for up to 128 processors.

The Muus and Utah systems achieve interactive rates with high image quality even for highly complex scenes. However, their approaches are based on expensive supercomputers. Wald et al. [Wal01, Wal01a] designed a ray tracer using object dataflow strategy that runs on a cluster of workstations. In a preprocessing step, a top-level axis-aligned BSP tree is built similarly to the approach presented in [Cas89] and this structure is broadcasted to all processors. Simultaneously, the object space is subdivided into the small voxels as in [Jev89] and the voxels are distributed over the processors. Each processor then creates a low-level BSP tree for its every voxel. Unlike the classical approaches [e.g. Neb98, Kea94, Bad90], where an object is considered as the smallest element to be sent and cached, the proposed algorithm sends an entire voxel and its objects. This solution introduces some overhead, however, in practice it leads to higher efficiency because of the ray-coherence. The term ray-coherence denotes the tendency that two adjacent rays will intersect the same objects which can be exploited to accelerate significantly the tracing. To reduce the transfered data volume, the voxels are packed before they are sent. Rather than waiting for the required voxel, the processor starts to process next ray. For the load-balancing, Wald et al. use a similar approach to the one published in [Par99]. As the proposed ray-tracer was carefully optimized, it is very efficient and also scalable. It even outperforms the best hardware graphical adapters (2001) when the scenes with one million or more triangles are rendered. Using seven Pentium III 800 MHz processors connected with 100 Mbit Ethernet, the authors were able to achieve 3-5 images in the resolution 640x480 per second of four models of the UNC Power-plant [UNC]. The total complexity of the scene was roughly about 50 million triangles. If an additional hardware support is added, the frame rate increases up to 8.

The solution just described is limited to static environments and walkthrough applications because a dynamic change in the scene (e.g. the movement of the object) would require changes in the structures for the acceleration of ray tracing and these changes cannot be performed in real-time. In their recent work [Wal03], Wald et al. faced this problem and improved their algorithms to handle few dynamic changes in the scene.

SaarCOR [Sch03] is a hardware architecture for the ray-tracing being developed at present at the Saarland University. It is a single chip that reaches the performance comparable to the current graphical adapters but uses less hardware resources and requires less memory bandwidth. The hardware design is highly flexible and scalable.

Another important event on the field of real-time ray-tracing is the development of graphics API that provides an abstraction layer, i.e. enabling programmers to write 3D graphics programs that can run with any underlying rendering engine. OpenRT [ORT] is such an API. As it is similar to the OpenGL API, it allows easy porting of many existing applications.

# 4 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] papers 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:

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

*Figure 4.1* 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 *d*-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 configuration 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.

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 4.1*. 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.

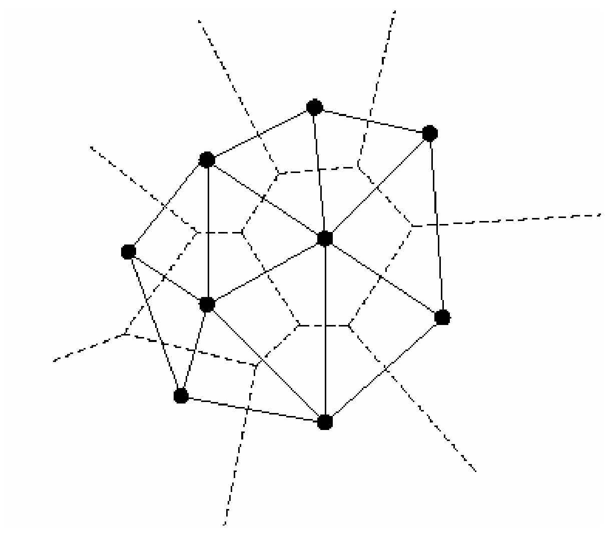
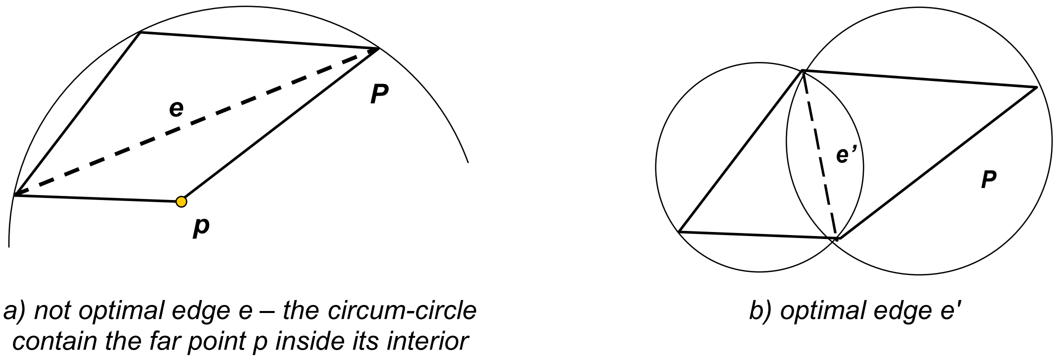


**Figure 4.1:** *The Delaunay triangulation (solid lines) and the Voronoi diagram (dashed lines) of the same set of points (big black dots)*

## 4.1 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 4.2*. Let us note that there exists another strategy of local transformations: 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.



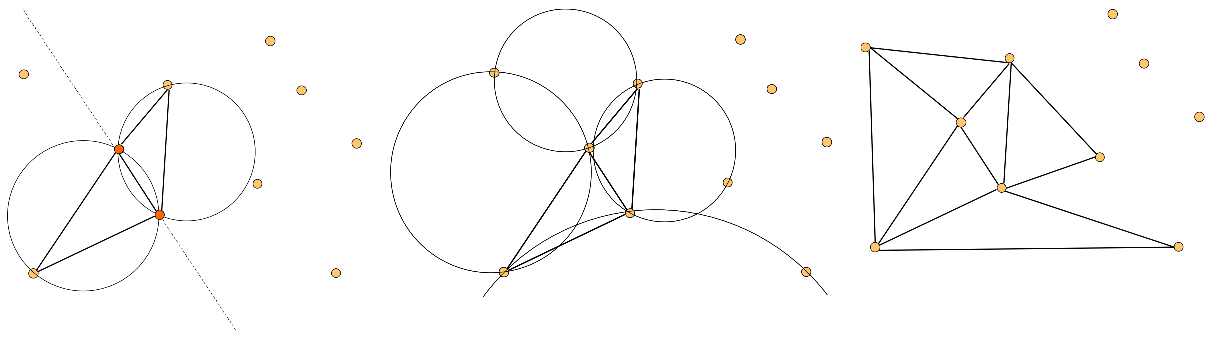
*a) not optimal edge e – the circum-circle contain the far point p inside its interior*

*b) optimal edge e'*

**Figure 4.2:** *Flipping the diagonals to convert a general triangulation into DT.*

## 4.2 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 4.3* shows several steps of the construction.



*a) initial two triangles* *b) construction of next triangles* *c) partial triangulation*

**Figure 4.3:** *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 potential triangle that was generated during the process. If it does not reach any point before the top of the circum-circle is reached, the triangle with this circum-circle is added into the triangulation. *Figure 4.4* 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 4.4a*. 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 4.4b*. 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 4.4c*.

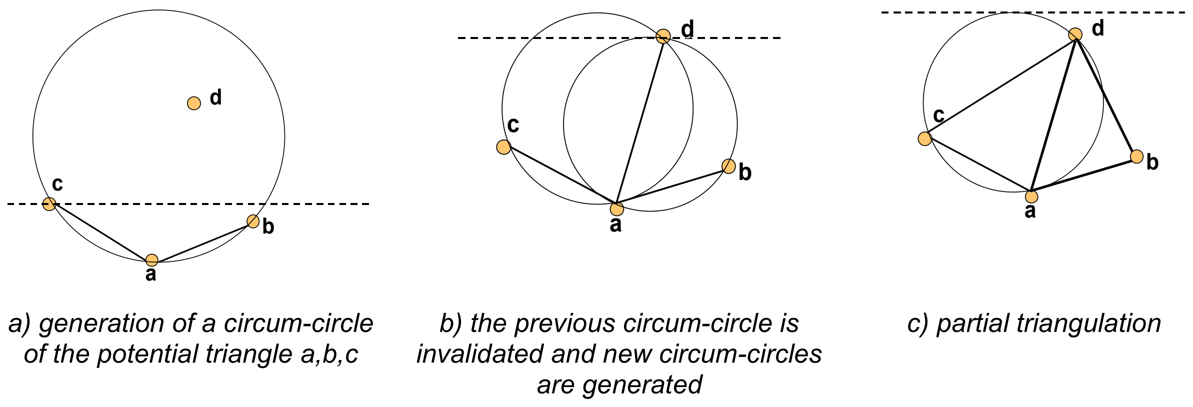

d
c
b
a
a) generation of a circum-circle of the potential triangle a,b,c
d
c
b
a
b) the previous circum-circle is invalidated and new circum-circles are generated
d
c
b
a
c) partial triangulation

**Figure 4.4:** *The sweeping in $E^2$ by Fortune.*

## 4.3 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 found and this simplex and appropriate simplices in the neighborhood are modified in such a manner to incorporate current point and 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 Directed Acyclic Graph (DAG) is used [Ber97]. 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], use of quadtrees or bucketing techniques. The random walk techniques are especially popular. They consume less memory, however, expected time $O(N^{1/4})$ 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 4.5* shows an example of the insertion. In this example no local improvement technique is required, after the retriangulation we have already the Delaunay triangulation.

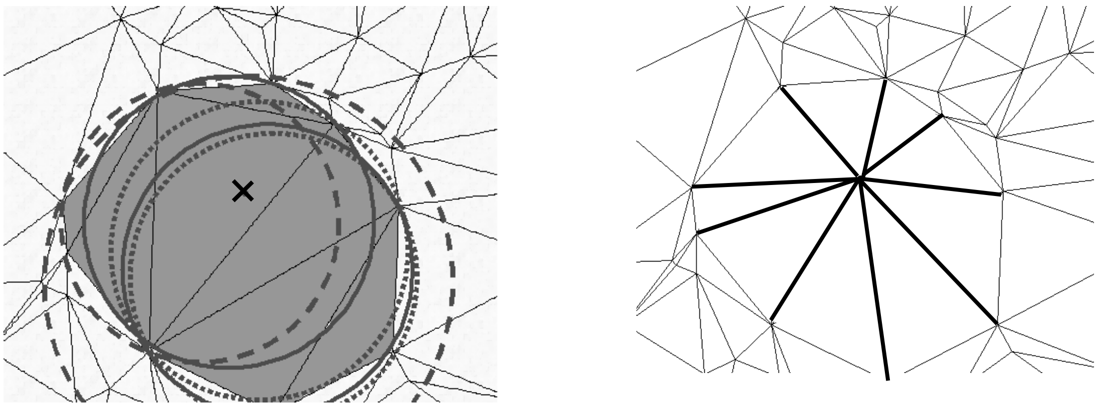

*a) triangles forming the cavity* *b) retriangulation of the cavity*

**Figure 4.5:** *Insertion of a point (marked by the cross) inside the DT by Bowyer-Watson approach.*

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 star-shape 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 4.5* shows an example of the insertion. In this example, no local improvement technique is required, after the retriangulation we have already the Delaunay triangulation.

In its simplest form, the Bowyer-Watson algorithm is not robust against floating-point roundoff error, which may cause appearance of overlapping simplices [Sch99]. It is also more difficult for the implementation than the approach with successive performing of local transformations.

## 4.4 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 4.6* illustrates this approach. Let us note that the complexity of construction 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.

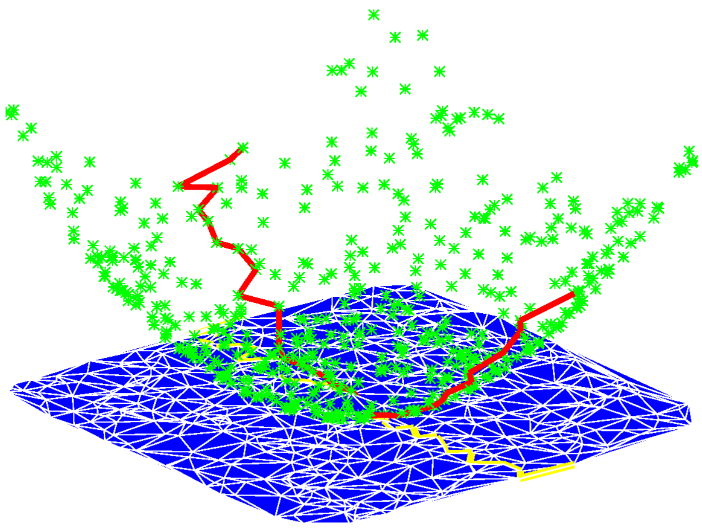


**Figure 4.6:** *The projected points and the corresponding Delaunay triangulation [Har97]. Only a tiny part of the 3D convex hull is shown (black bold line segments).*

## 4.5 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 few points are 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 4.7* 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).

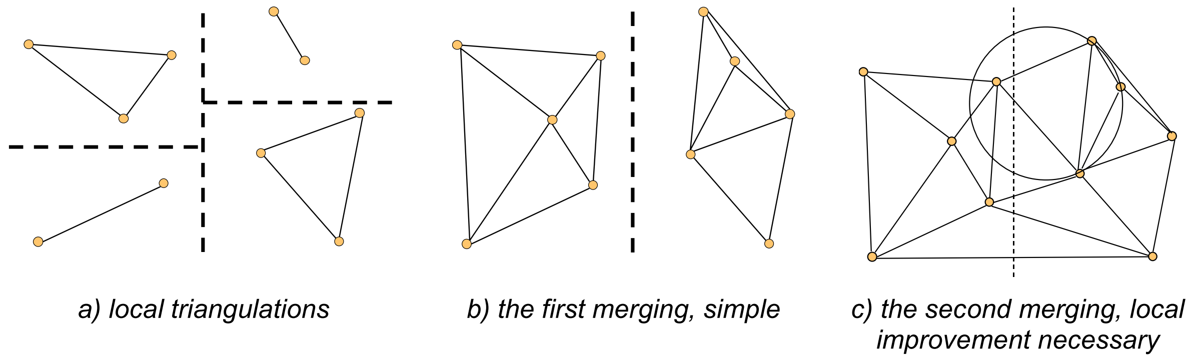

a) local triangulations
b) the first merging, simple
c) the second merging, local improvement necessary

**Figure 4.7:** *Several steps of the construction of the DT in $E^2$ by D&C.*

# 5 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 (*Figure 3.1*).

## 5.1 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.

### 5.1.1 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 5.1* shows an example of the triangulation.

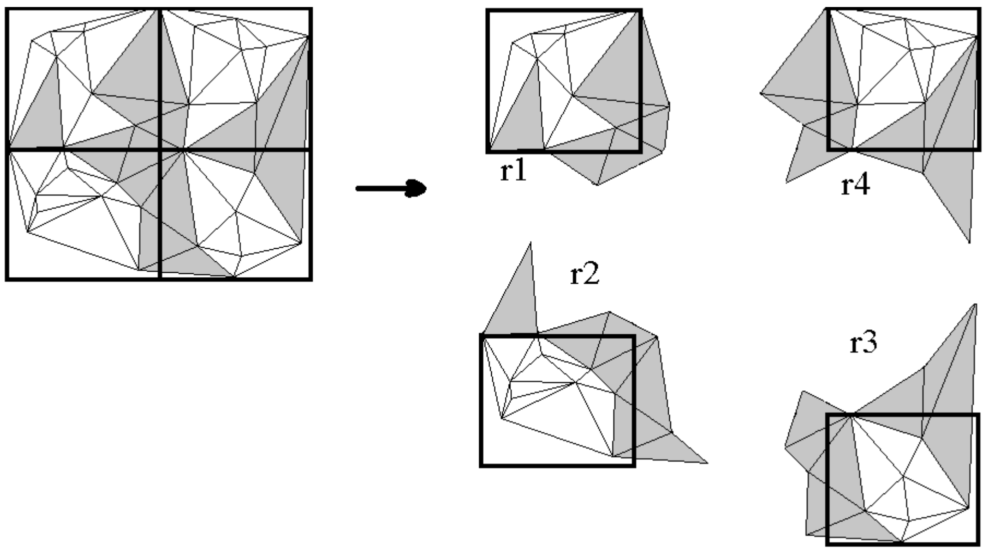


**Figure 5.1:** *The Delaunay triangulation in $E^2$ of using 4 PEs. The dark triangles are constructed redundantly by more processors.*

For the problem of the *DT(S)* in $E^3$ the authors present, for example, the 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., 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.

### 5.1.2 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 constructs 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.

## 5.2 Incremental Insertion

The nature of the incremental insertion is rather sequential and, therefore, sequential algorithms belonging to this category are rarely parallelized. There is, as far as we know, 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].

### 5.2.1 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 5.2* 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.

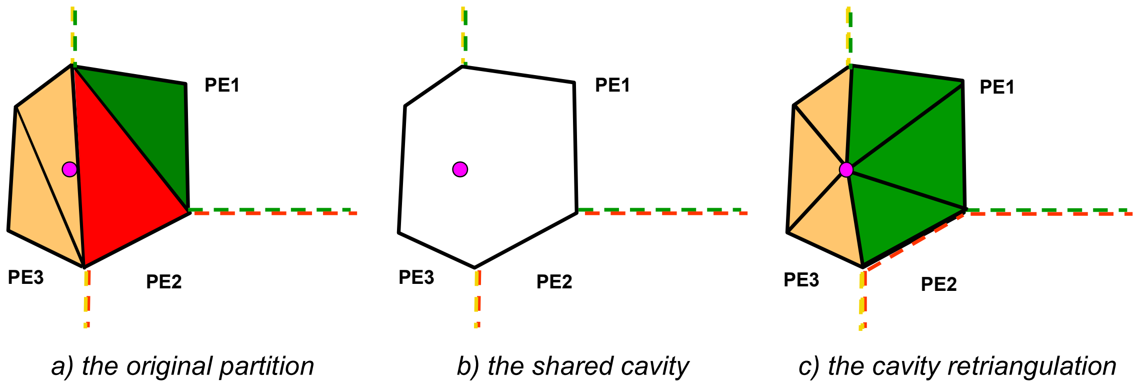


**Figure 5.2:** *The insertion of a point (big dot) into the triangulation in $E^2$. This insertion causes retriangulation of a cavity shared by three processors (PE1, PE2 and PE3) including the update of their regions' boundaries. The simplices belonging to the same region are shown in the same color.*

### 5.2.2 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 5.3*. 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 which 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 5.3*) 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 work-load balance is improved.

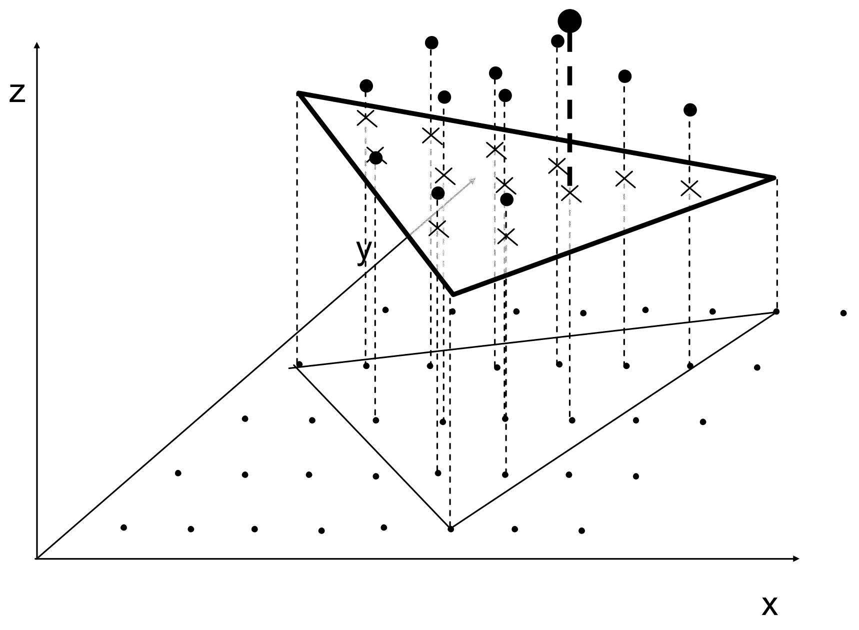


**Figure 5.3:** *A triangular patch (bold triangle) and its appropriate set of input points (big dots). The point with the worst approximation (the approximations are marked by crosses) is shown bigger.*

### 5.2.3 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 presented 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 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 among 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 144600 points. In our opinion, the reached speed-up is quite low.

Probably better results could be achieved by the approach of 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.

## 5.3 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.

### 5.3.1 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 $\alpha$ 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 5.4*.

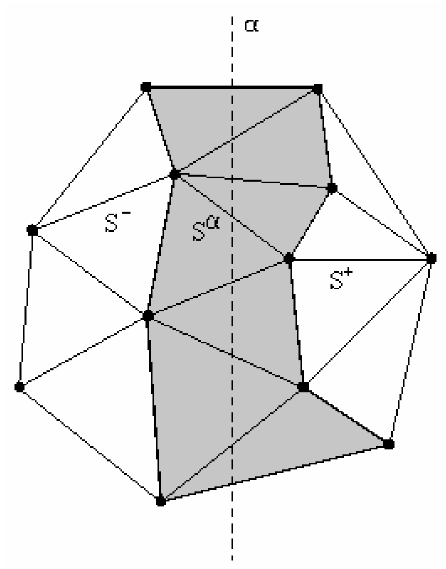


**Figure 5.4**: *The triangulation by DeWall in $E^2$ [Mag98]*

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 work-load imbalance. Hardwick [Har97] claims that the situation is much worse (up to ten times) if we consider non-uniform data sets.

### 5.3.2 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 5.5* shows the merging of two local triangulations and their wall of the points lying inside a circle.

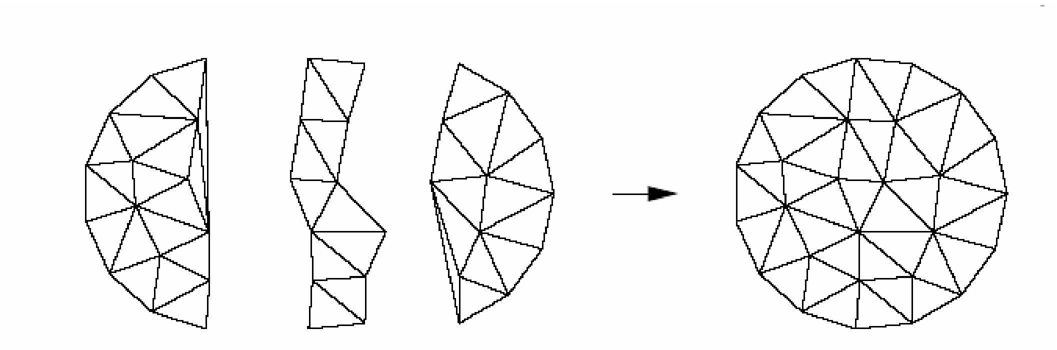
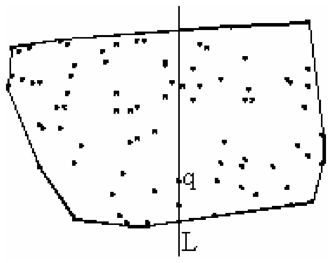


**Figure 5.5:** *The Delaunay triangulation given by a merge of two local triangulations and its common interface (created from two local interfaces) [Che01]*

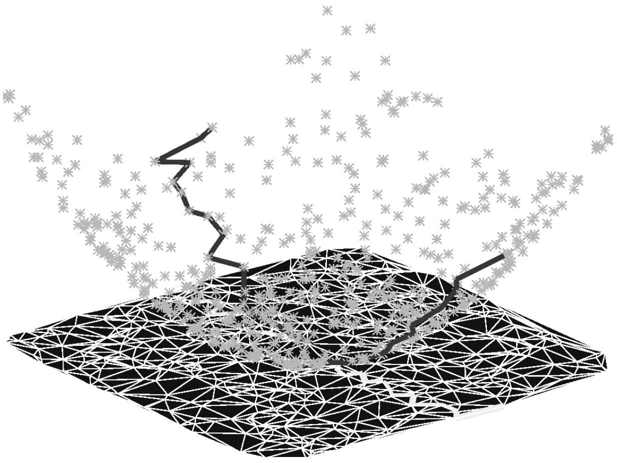
The entire algorithm, except for the merge phase, can be processed in parallel. Time needed for the merge phase is, however, negligible (thanks to interfaces) in comparison to other phases. Therefore, the algorithm achieved outstanding speed-up. For example, the tested uniform data sets with 96K points achieved speed-up 1.57 – 4.95 for 2 – 8 PEs at IBM SP2 with High Performance Fortran.

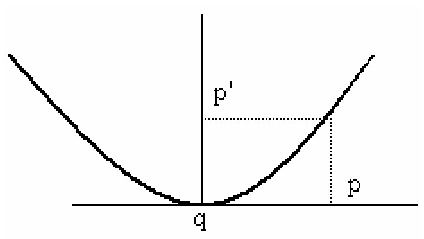
### 5.3.3 Hardwick

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 5.6a.* 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”.



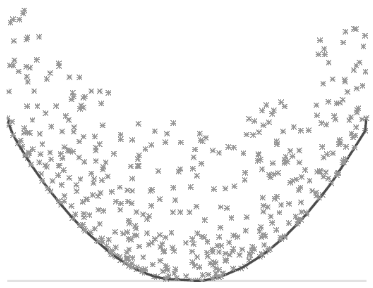
*a) the median line L going through the median point q*

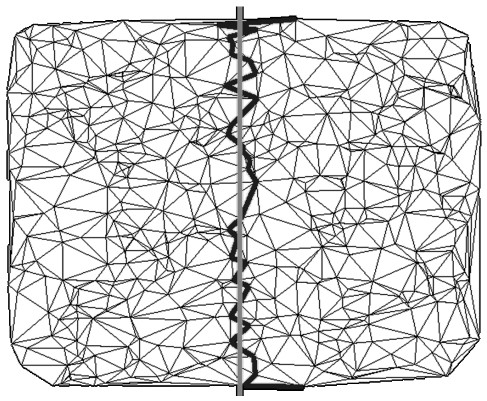




*b) the projection of the point p on the vertical plane using the paraboloid centered at q – perpendicular view*

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





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

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

**Figure 5.6:** *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 5.6b*.

The lower convex hull of the projected points is found by a parallel modification of simple quickhull [Pre85] – see *Figure 5.6c* and the back projection of the resulting lower convex hull into the plane *xy* gives a set of line segments – see *Figure 5.6d*. 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 5.6e*.

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 extremely 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 – 5.8 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 work-load. 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; 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.

# 6 Incremental Insertion with Local Transformations

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. In this section, we describe an incremental insertion algorithm with local improvements that was chosen as a base for our parallel solution.

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. All input points do not need to be available at the beginning of computation (although their range of coordinates is required) and it 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.

## 6.1 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 4) 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 contrary, 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 6.1*. More detailed description is given by Žalik and Kolingerová in [Žal03].

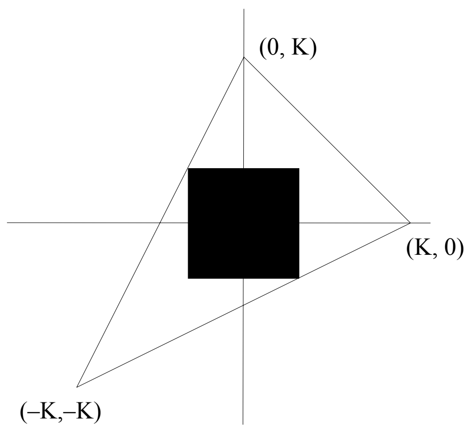


**Figure 6.1:** *The selection of the auxiliary simplex in $E^2$. The black rectangle is the bounding box.*

## 6.2 Location – The DAG

We use the Directed Acyclic Graph (DAG) to locate the simplex to be subdivided. Each node describes one simplex. If we want to find a simplex containing the current point to be inserted, we take the root (it describes the auxiliary large simplex) 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 (it explains why the DAG structure is not a tree, although it resembles it). 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. More details about the structure can be found in [Ber97]. *Figure 6.2* 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 exists 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.

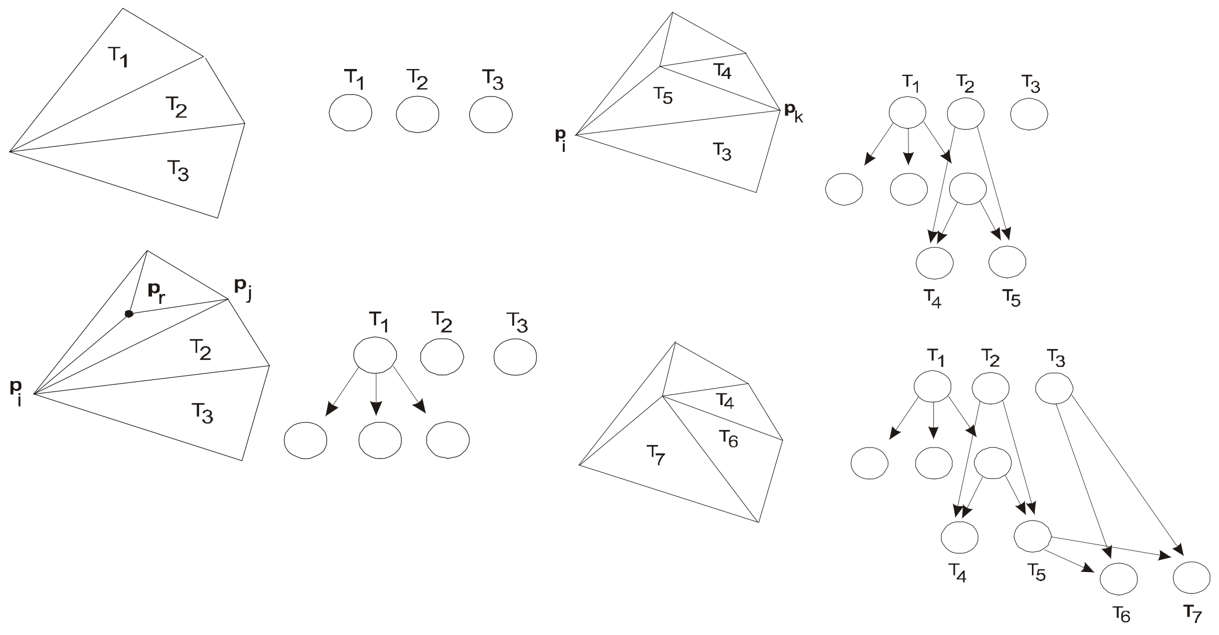


**Figure 6.2:** *The changes in the planar Delaunay triangulation caused by the insertion of the point $p_r$ and the corresponding changes of the DAG structure.*

## 6.3 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 6.3a*) and four tetrahedra in $E^3$ (see *Figure 6.4a*).

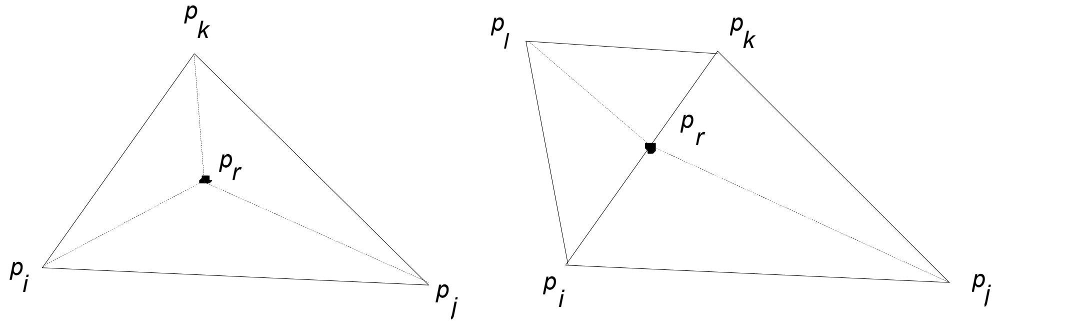
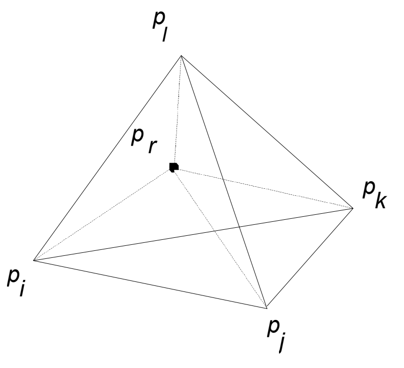

$p_k$
$p_r$
$p_i$
$p_j$
$p_l$
$p_k$
$p_r$
$p_i$
$p_j$
a) the point to be inserted lies strictly inside
b) the point to be inserted lies on an edge

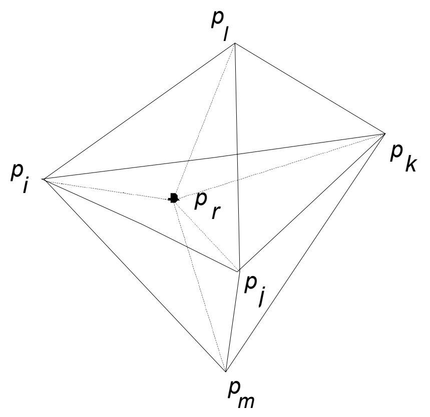
**Figure 6.3:** *Subdivision in $E^2$.*

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 6.3b*) or six tetrahedra (see *Figure 6.4b*).

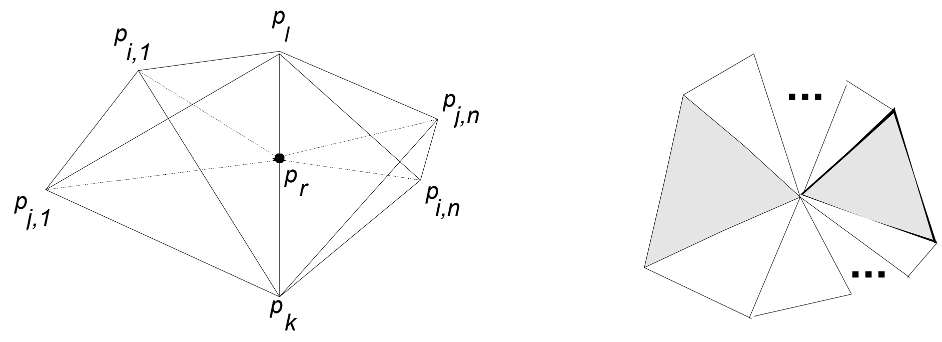
If the point to be inserted lies on an edge of the simplex in $E^3$, all simplices sharing this edge have to be subdivided. In the worst-case all simplices in the triangulation will be subdivided. Each simplex is subdivided into two new simplices (tetrahedra) – see *Figure 6.5*. Let us note that this case is difficult to be handled.



*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 6.4:** *Subdivision in $E^3$*.

## 6.4 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 6.5*, is simple: the edge is just swapped (see also *Figure 4.2*).

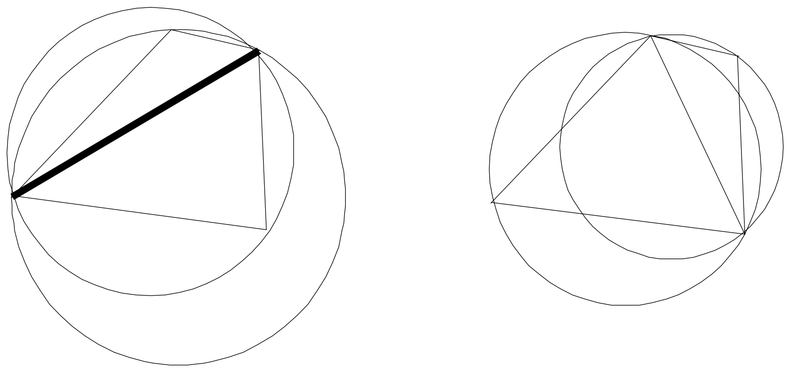


**Figure 6.5:** *Local transformations in $E^2$. The edge is swapped.*

After that, indeed, we have new outer edges (or faces) that have to be tested. *Figure 6.6* shows an example of the propagation of the local transformations in $E^2$. The located triangle is subdivided into three new triangles (*Figure 6.6a* – 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, therefore, flipped. As the circum-circle of the just created triangle $T_2$ is not empty, the flipping has to continue – see *Figure 2b*. Finally, the Delaunay triangulation is achieved (*Figure 6.6c*).

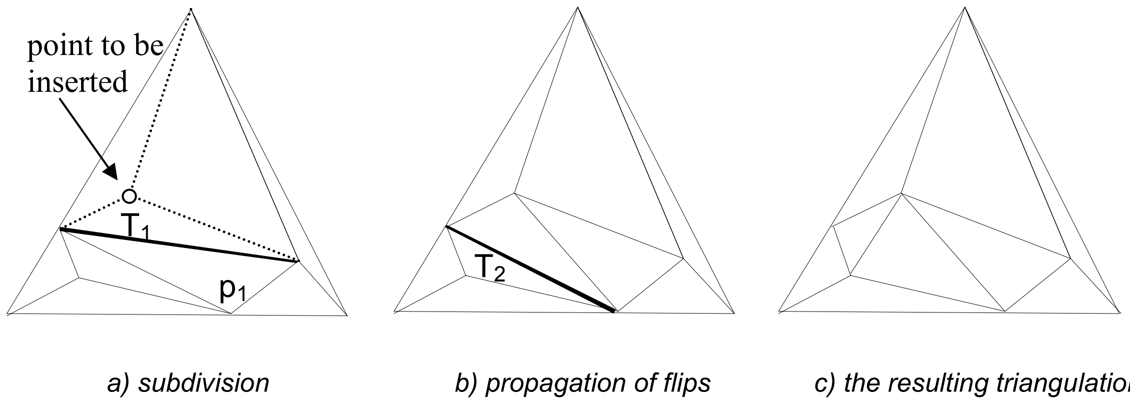


**Figure 6.6:** *The incremental insertion in $E^2$. Edges that should be flipped are bold.*

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 6.7*.

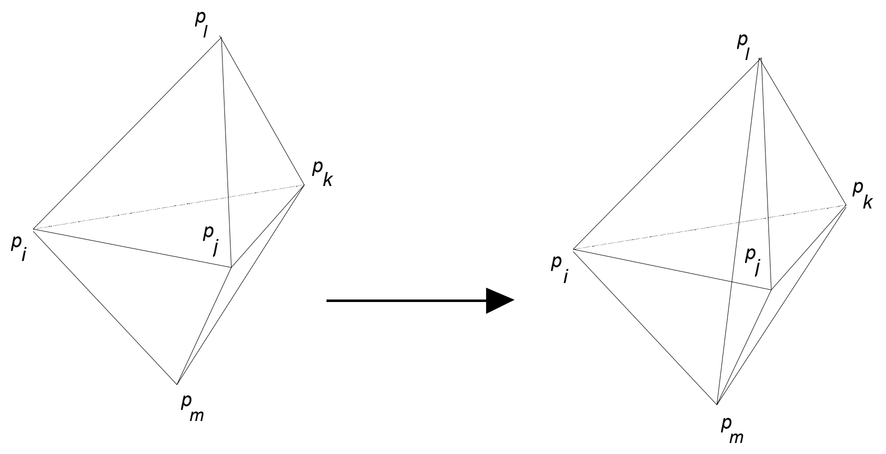


**Figure 6.7:** *Local transformation of two adjacent tetrahedra $p_i\, p_j\, p_k\, p_l$ and $p_i\, p_j\, p_k\, p_m$ sharing illegal face $p_i\, p_j\;\, p_k$ into three tetrahedra $p_i\, p_j\, p_l\, p_m$, $p_i\, p_k\, p_l\, p_m$ and $p_j\, p_k\, p_l\, p_m$.*

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 6.8*.

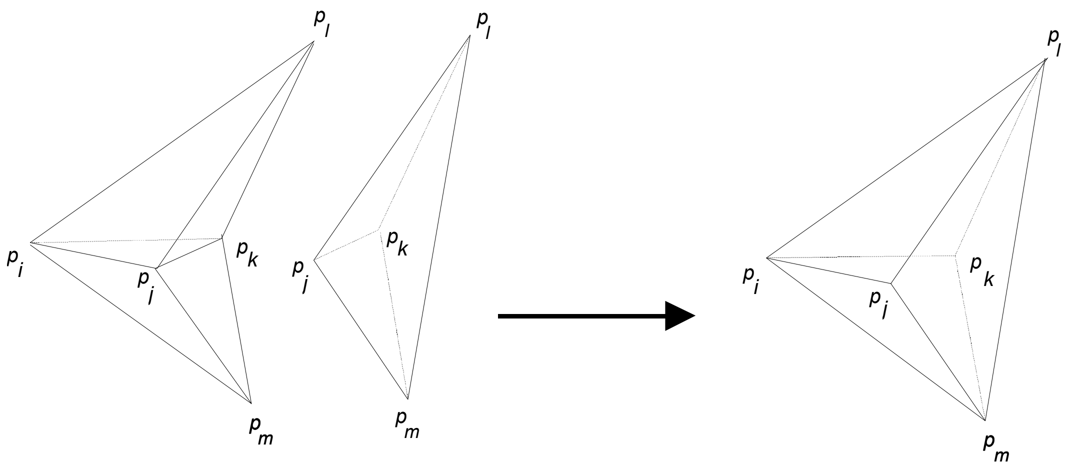
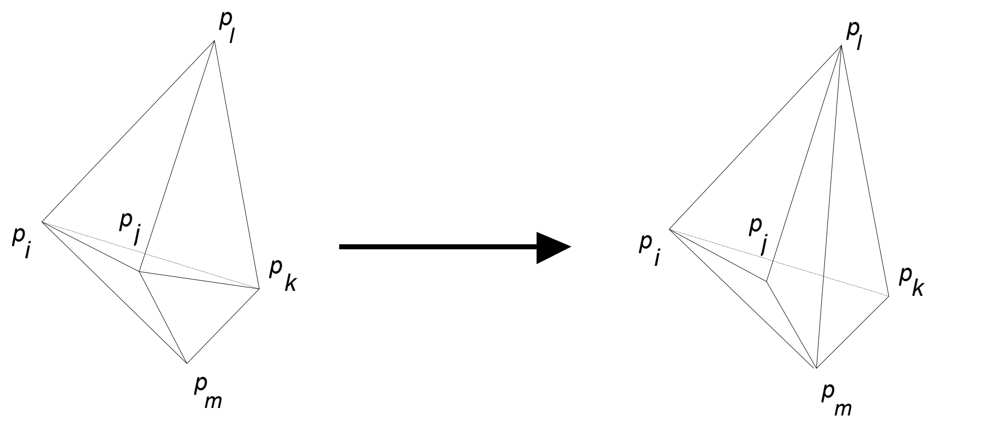
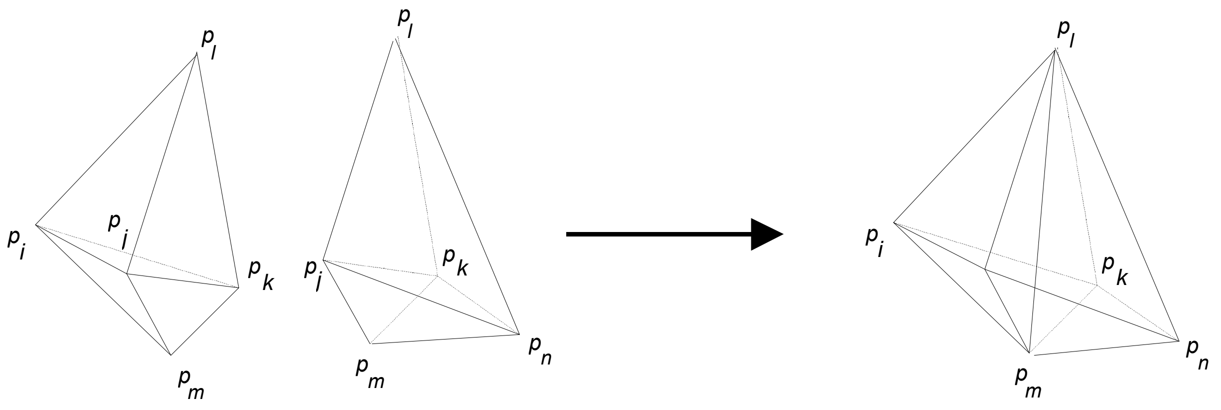


**Figure 6.8:** *Local transformation of three adjacent tetrahedra (tetrahedron $p_j\, p_k\, p_l\, p_m$ is shown separately to increase readability) $p_i\, p_j\, p_k\, p_l$, $p_i\, p_j\, p_k\, p_m$ a $p_j\, p_k\, p_l\, p_m$ into a pair of tetrahedra $p_i\, p_j\, p_l\, p_m$ a $p_i\, p_k\, p_l\, p_m$.*

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 are not available, 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 6.9* 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 6.9:** *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 node of the DAG is the simplest and also the most efficient solution.

## 6.5 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.

# 7 Parallelization

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 millions of points. In such cases, a parallel algorithm is useful and welcome. Quite a big set of parallel algorithms exists (see Section 5), however, they 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. In the last few years, multiprocessors with several processors and shared memory, especially two-processors, have come into consideration 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, more suitable for architectures with a limited number of processors, typically 2 or 4, and with a shared memory (such an architecture is commonly used and widely available nowadays in the computer graphics laboratories).

We have chosen the randomized incremental insertion algorithm described in Section 6 as a base for our parallel algorithms because this algorithm has many advantages: e.g. simplicity or robustness (see the previous section for more details). Let us assume we use parallel architecture with shared memory. The main idea is to let several threads to insert the points simultaneously into the one triangulation stored in the shared memory. Indeed, we have to prevent a thread to perform a non-consistent modification somehow. Let us note that the developed algorithms are simple to be implemented even by a person focused on computer graphics without a deep knowledge of parallel computations.

## 7.1 The Shared DAG Structure

We start several threads (usually one per each processor) and let them to insert points simultaneously into the shared triangulation accessing nodes of the DAG structure. A node can be accessed simultaneously by several threads if all threads need it for read-only purpose. When any thread needs to modify the node, some synchronization among the threads has to be implemented, otherwise the concurrency in this case could produce artefacts in the resulting Delaunay triangulation or even could lead to the collapse of the program. *Figure 7.1* shows one of the possible results of the unsynchronized subdivision phase when two points are to be inserted into the same simplex.

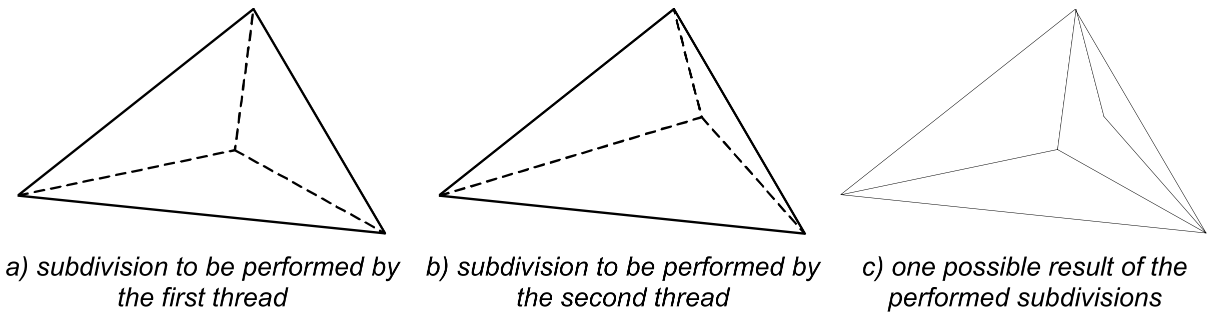

a) subdivision to be performed by the first thread
b) subdivision to be performed by the second thread
c) one possible result of the performed subdivisions

**Figure 7.1:** *Simultaneous unsynchronized insertion in $E^2$.*

It is clear that the node $N_0$ has to be modified if the simplex $S_0$ described by this node has to be subdivided (regardless if it is performed in the subdivision phase or in the legalization phase). However, the node stores also the pointers on the nodes describing the simplices adjacent to the simplex $S_0$. Therefore, when of any of these adjacent simplices is subdivided, the node $N_0$ is modified as well. As the leaves are in a bidirectional list, another modification of the node $N_0$ occurs if its 'next' or 'last' node is subdivided. *Figure 7.2* shows the content of two nodes of the DAG structure from the last step in *Figure 6.2*.

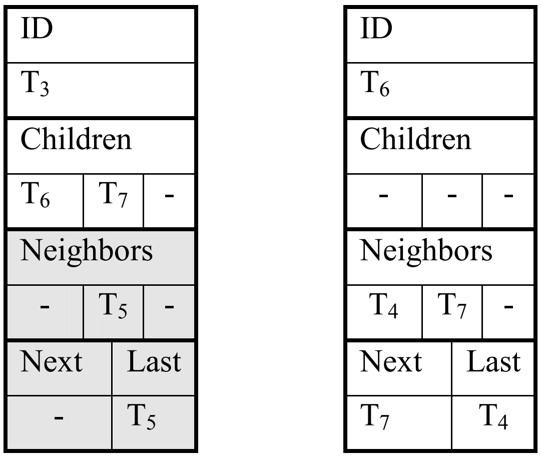


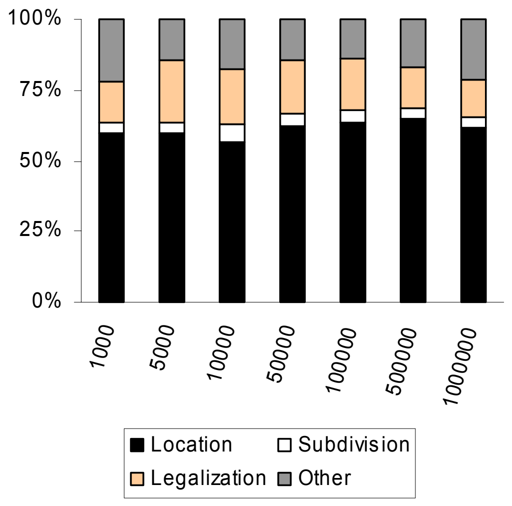
**Figure 7.2:** *The content of two nodes of the DAG structure from Figure 6.2.*

There is another hidden synchronization among the threads: the allocation of memory for a new node has to be performed exclusively, otherwise the memory blocks could overlap which causes a crash of the application in the most cases. Naive solution uses a critical section whenever a node is to be allocated. As the application needs to allocate new node frequently, it is not an efficient choice. Therefore, we allocate, in a critical section, a group of nodes at once and store their pointers in a local cache. When a new node is required, first free pointer from this cache is picked up without any further synchronization. If the cache is empty (i.e. there is no free pointer), then another group of nodes has to be allocated. Experimentally, we decided to use the cache length of 512 nodes.

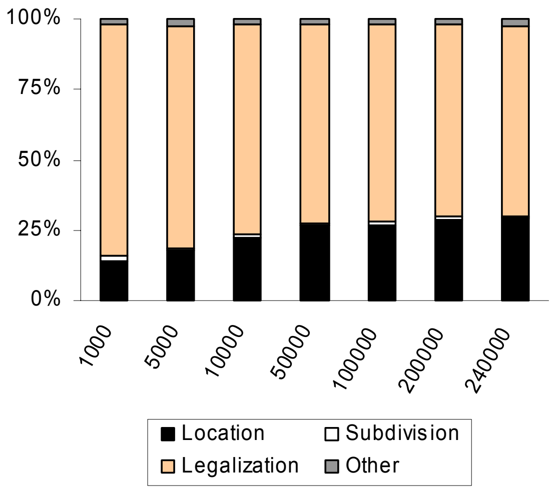
## 7.2 Analysis of the Sequential Algorithm

Let us first analyze the run of the sequential algorithm. All three phases of the algorithm need to access to the DAG structure, however, each one in a different way. In the location, DAG is accessed read-only in such a manner to find the triangle containing the point to be inserted. Then all corresponding triangles are subdivided, and new nodes are added to the DAG structure. The DAG is also modified during the legalization.

Typical runtimes needed for these parts in $E^2$ are in *Figure 7.3a*. The majority of time 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% – see *Figure 7.3b*.



a) $E^2$



b) $E^3$

**Figure 7.3:** *Typical runtimes needed for the sequential algorithm. Uniform data sets were tested. The data sizes N are on x-axis.*

## 7.3 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 *i*-th bit in the value of this 3-bits (4-bits in $E^3$) long parameter is set to one then *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.

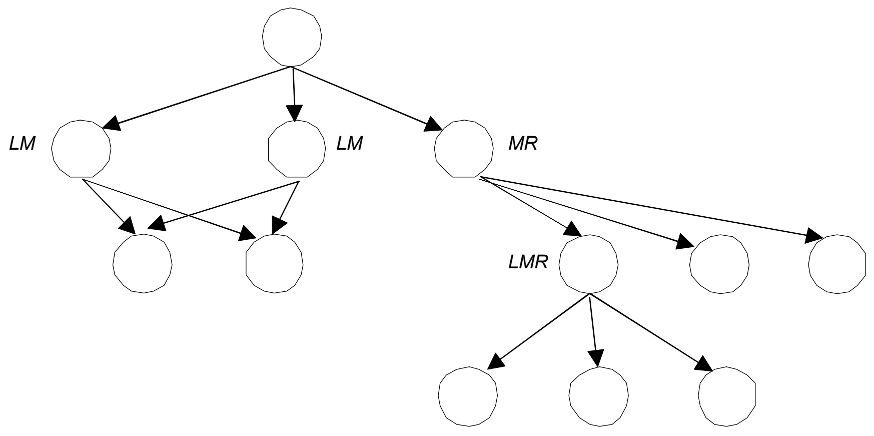


**Figure 7.4:** *The classification of the nodes in the DAG.*

*Figure 7.4* 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.

## 7.4 Parallel Subdivision and Legalization Phase

In both the subdivision and the legalization phases a thread has to access one ore 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.

## 7.5 Batch Principle and Batch Method

In the batch principle, there exist 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.

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

```
for r := 0 to m-1 do begin
   Locate the triangle $T_0$ containing $p_r$ on the level of the parents
   of the leaves in the DAG structure;
   if the shared queue is full then wait;
   put {$T_0$ , $p_r$ } into the queue
end;
```

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

```
while not all points inserted do begin
   if the shared queue is empty then wait;
   get { $T_0$ , $p_0$ }from the queue;
   Locate the triangle $T_1 \in DT(S)$ containing $p_0$; //start at $T_0$
   Subdivide $T_1$;  // in the case where $p_0$ lies on the shared edge, say
                     //between $T_1$ and $T_2$ then subdivide also $T_2$.
   Legalize all new triangles;
end;
```

**Figure 7.5:** *Parallel construction - the batch method*

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 (see *Figure 7.6*) the queue length of 1024 seems to be optimal. Algorithms for searching thread(s) and a specialized thread are described in *Figure 7.5*.

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

**Figure 7.6:** *The ideal queue length. The queue lengths of 16, 32, 64, 128, 256, 512, 1024, 2048 and 4096 on a Dell P 410 (2 PEs) were tested.*

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 analysis of the sequential algorithms, 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.

## 7.6 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 algorithm for inserting threads of pessimistic method is described in *Figure 7.7*.

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 which has just finished the location) has to start its waiting. It allows using this method even in a case when the location phase does not consume more than 50%, i.e. pessimistic method is also available in $E^3$ but substantial speed-up cannot be expected.

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

for $r$ := 0 to $m-1$ do begin
   Locate the triangle $T_0$ containing $p_r$ on the level of the parents
   of the leaves in the DAG structure;
   Enter a critical section;
   Locate the triangle $T_1 \in DT(S)$ containing $p_0$; //start at $T_0$
   Subdivide $T_1$;  // in the case where $p_r$ lies on the shared edge, say
                  //between $T_1$ and $T_2$ then subdivide also $T_2$.
   Legalize all new triangles;
   Leave a critical section;
end;
```

**Figure 7.7:** *Parallel construction - the pessimistic method.*

## 7.7 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 7.8* 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 (see thick, dashed, and dotted circles in *Figure 7.8*) have to be modified. In the figure, these triangles are marked by the dark gray color. No other triangles are modified.

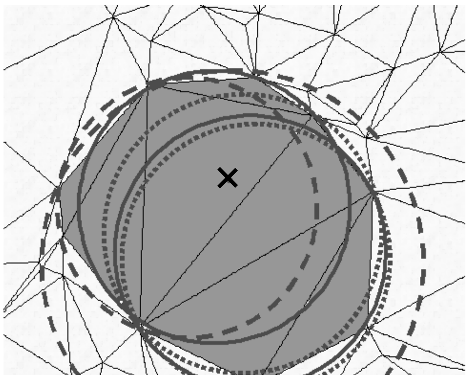


**Figure 7.8:** *The locality of changes in the triangulation in $E^2$. Inserted point is marked by a cross.*

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 more efficient strategy. In the shared memory, sets $M_t$ of accessed nodes ($t$ = 0 – $k$-$1$, 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 also for write purpose. 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 – $k$-$1$ 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.

### 7.7.1 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. *Figure 7.9* demonstrates the inefficiency of standard critical sections on the Microsoft Windows platforms. Although the optimistic method is supposed to be faster than the pessimistic one, the results of experiments show an opposite if standard critical sections are used.

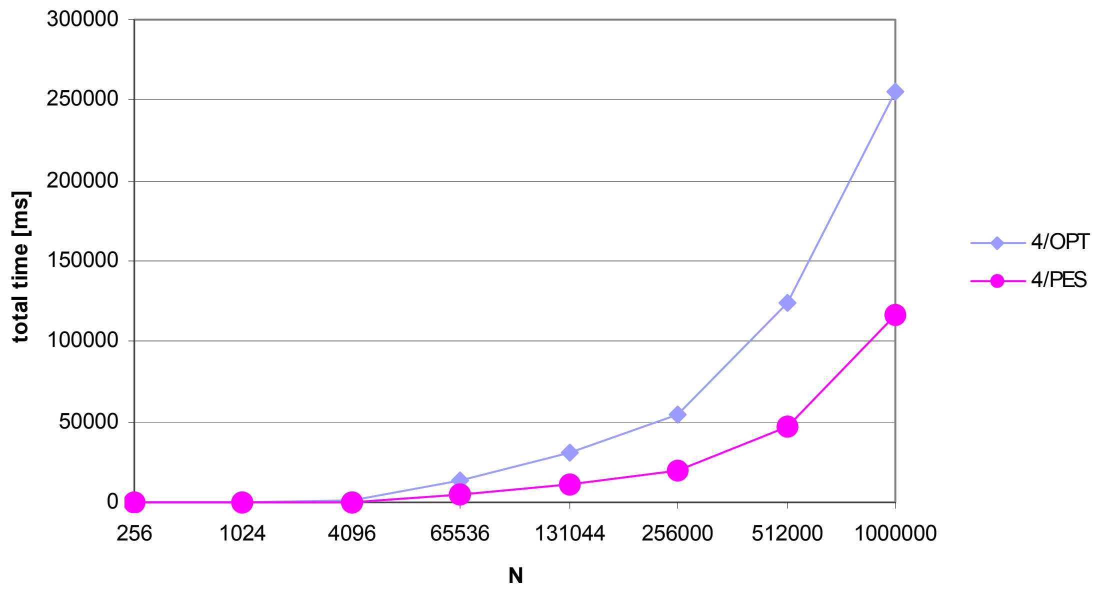


**Figure 7.9:** *The comparison of the pessimistic method (PES) and the optimistic method (OPT) in $E^2$, where standard synchronization object for critical section is used for the 'locking' of triangles. Uniform data sets on Dell Power Edge 8450 were tested.*

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 back 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.

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 enough 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, moreover, this value is not ensured to be a maximum, we use a dynamic solution similar to that one used for storing locked nodes.

*Figure 7.10* 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 7.10a*, 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, however, locked by the thread $T_2$ – *Figure 7.10b*. 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 7.10c*. Meanwhile, the thread $T_1$ tries to insert its point once again.

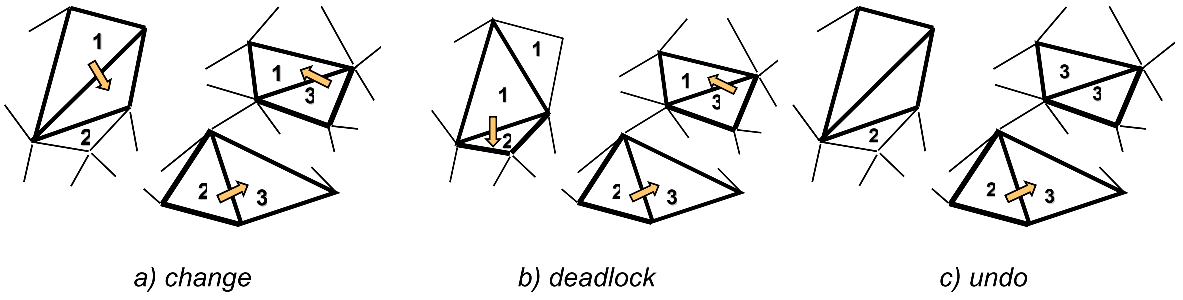
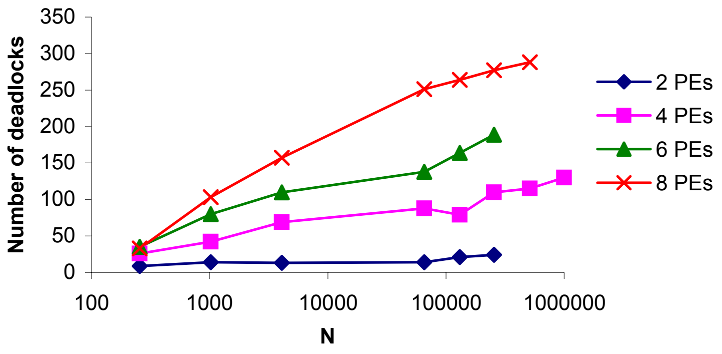


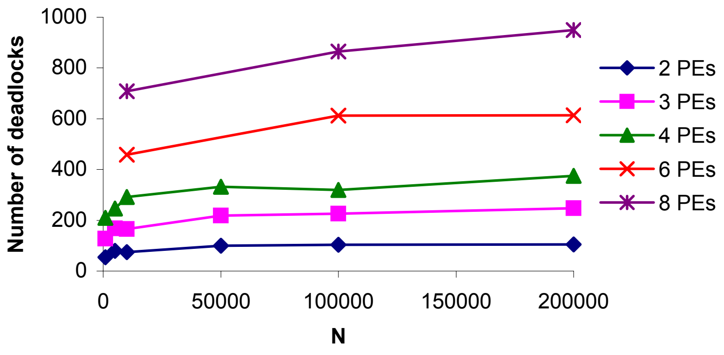
**Figure 7.10:** *The deadlock handling in the optimistic method – $E^2$ case.*

### 7.7.2 Burglary Method

There is no doubt that the transaction mechanism (needed for deadlock handling) negatively influences the performance of the parallel solution. *Figure 7.11* shows that the probability of the occurrence of deadlocks is almost negligible and, moreover, it decreases with the growing number of input points. It means that the transactions are in most cases a kind of luxury and, therefore, some strategy, which avoids the transactions, is welcome.



*a) $E^2$, Dell Power Edge 8450*



*b) E3, Dell Power Edge 7150 (2, 3 and 4 PEs) and ES5000 (6 and 8 PEs)*

**Figure 7.11:** *Number of deadlocks detected in the optimistic method for uniform data sets.*

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.

*Figure 7.12* 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 7.12a*, 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 7.12b*.

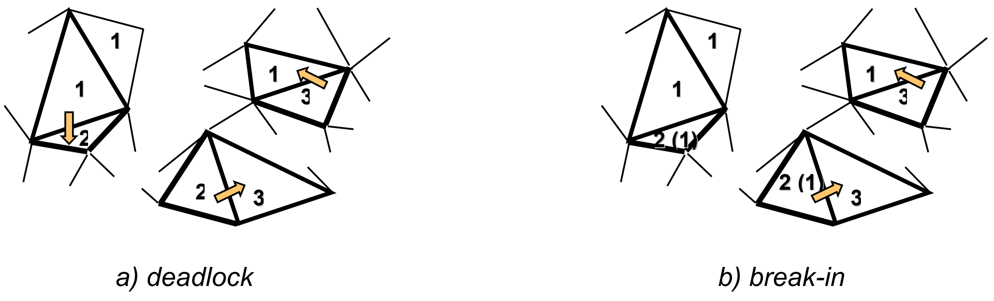
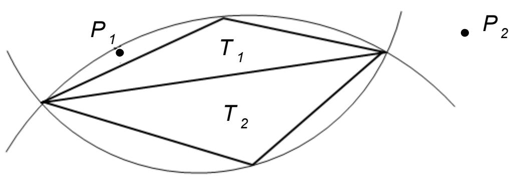


**Figure 7.12:** *The deadlock handling in the burglary method – $E^2$ case.*

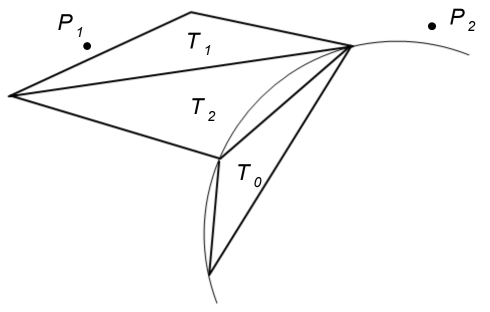
When the owner of the attacked nodes finally finishes its waiting, it has to check whether its original request is still valid because, although it is not very probable, the offending thread could have modified the currently requested node(s). In such a case, the thread cancels the request and continues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

## 7.7.3 Circum-Circle Method

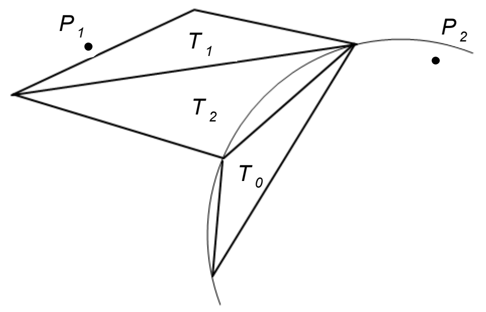
The circum-circle method is a modification of the burglary method. The decision whether to access a node or wait is, unlike the previous optimistic methods, solved by a geometric test. It can be proved that the subdivision and the legalization influence only such simplices where the input point lies in their circum-spheres – let us remind Bowyer-Watson algorithm and *Figure 7.8*. 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 7.13* shows an example of the use of just described geometric test.



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

There are two serious problems with the circum-circle approach. Both negatively influence the efficiency of this method and limit its use on 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.

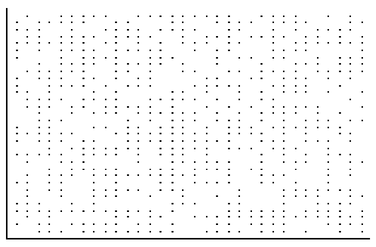
## 7.8 Experiments and Results

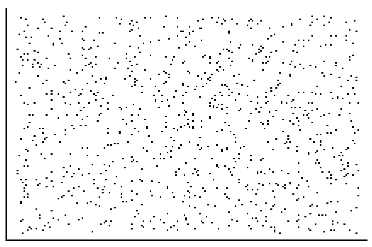
Parallel solution of Delaunay triangulation was implemented in Microsoft Visual Studio.NET 7.0 C++ using serial incremental algorithm implemented in Delphi 6. Unfortunately, we have only very limited access (approximately one per half a year) to any suitable computer with more than two processors and, therefore, the presented results are a collection of different experiments at various computers. The main tests were done on the following machines:

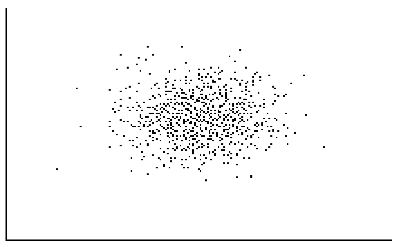
- Dell Precision 410 (2x Intel Pentium III 500 MHz, cache 512KB, 1GB RAM) with Microsoft Windows XP Professional operating system,
- Dell Power Edge 6400 (4x Intel Pentium III Xeon 550MHz, cache 1MB, 4GB RAM) with Microsoft Windows XP Advanced Server,
- 64-bits Dell Power Edge 7150 (4x Intel Itanium,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

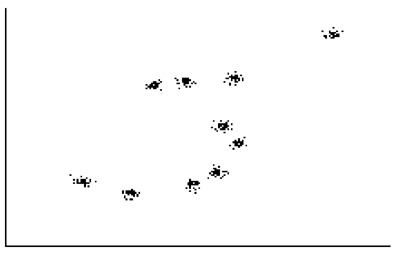
For additional tests, Shalla (2x Intel Pentium Celeron 533 MHz, cache 128KB, 512MB RAM) with Microsoft Windows 2000 Professional operating system was used.

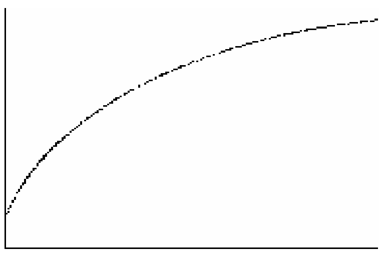
In our experiments, we used two kinds of testing data. The first group consists of artificially generated points with various distributions, such as grid, uniform, gauss, cluster, arc and sphere. The points were generated in a unit square. Points for grid distribution lie in a regular orthogonal grid. 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 which are singular for Delaunay triangulation (e.g. 5 or more points laying on a sphere in $E^3$). Examples of these distributions are shown in *Figure 7.14*. The tested number of input points, i.e. data size $N$, was between 1 000 and 1 000 000.


*a) grid data*


*b) uniform data*


*c) gauss data*


*d) cluster data*


*e) arc data - $E^2$ only*

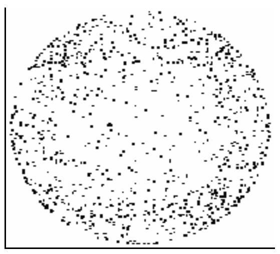

*f) sphere data, projection to the plane xy - $E^3$ only*

**Figure 7.14:** *Examples of tested distributions of the input points.*

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 7.15*. 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 computers with more processors (to which we have only a limited access) and we have chosen uniform data sets as representative.

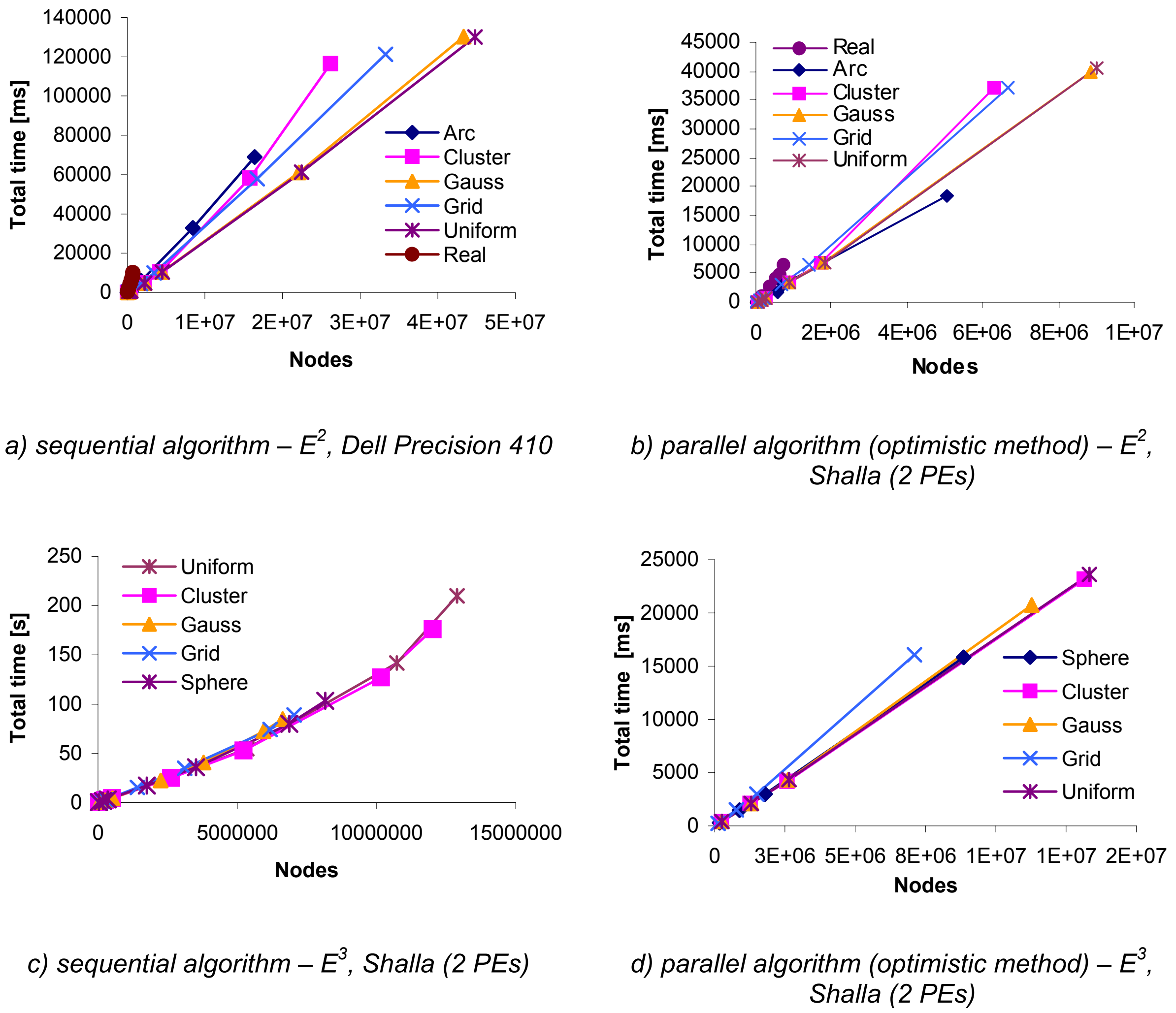


**Figure 7.15:** *The dependency of the total time on the number of nodes in the DAG structure.*

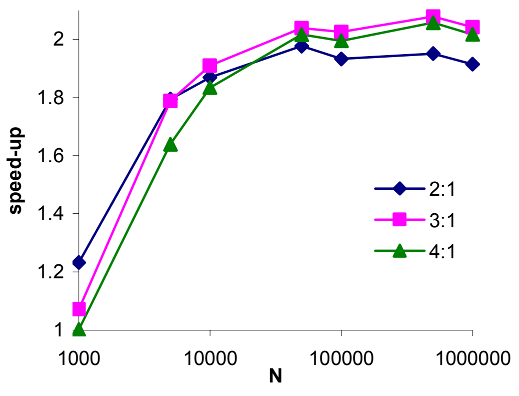
Besides the artificial data sets, we tested real data sets of terrain models (e.g. Crater Lake with 100 001 points) and some popular surface models (e.g. whale with 52 635 points, CTMayo with 98 869 or bell with 213 373). Most of these models were obtained from the Stanford repository [Sta99] and Žalik 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 speed-up divided by the number of used processing elements.

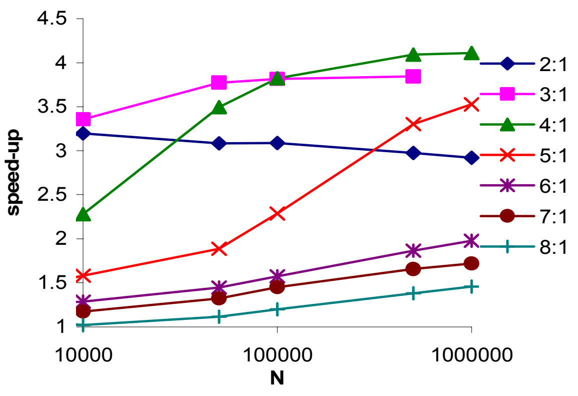
In the next text, we assume that points are subdivided randomly into *k* groups (*k* is the number of used threads) in such a manner to ensure equal number of points in each group. As two insertions of *m* points of the same point distribution take almost the same time, no dynamic load balancing is necessary (and also not used).

### 7.8.1 The Results of Batch Method

*Figure 7.16* shows the achieved speed-up for batch method for uniform data sets. Speed-up is mainly influenced by the number of input points. At first, speed-up quickly increases with the growing size of the input data set, then for some sizes it is almost constant and when larger data sets are processed, it tends to slowly decrease.



*a) Dell Precision 410 (2 PEs)*



*b) Dell Power Edge 6400 (4 PEs)*

**Figure 7.16:** *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 7.16a* 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 12*, 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 3.2. Use of five searching threads comes to consideration for data sets with more than approximately 1.5 millions of points. Probably, there exist such large data sets that would feed more searching threads (and also more PEs), however, computation of such data sets using the DAG structure needs more memory than is available in 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%.

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 Sen et al. [Sen96]. 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 7.17* 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.

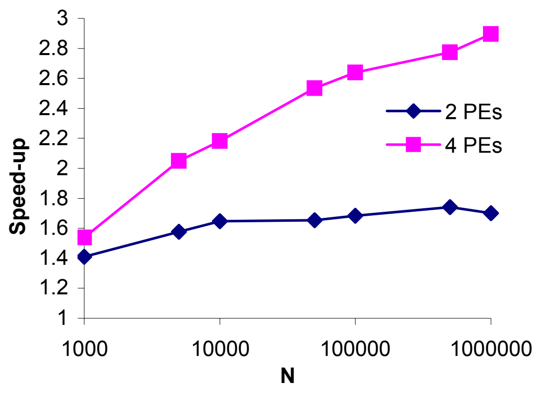
| **N** | **Speed-up** | | |
|---|---|---|---|
| | **Shalla 128 KB** | **P 410 1024 KB** | **PE 6400 4096 KB** |
| 1 000 | 1.132 | 1.072 | 1.642 |
| 5 000 | 1.698 | 1.787 | 2.218 |
| 10 000 | 1.606 | 1.910 | 2.391 |
| 50 000 | 1.695 | 2.039 | 2.342 |
| 100 000 | 1.749 | 2.025 | 2.422 |
| 500 000 | 1.745 | 2.079 | 2.399 |

**Figure 7.17:** *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).*

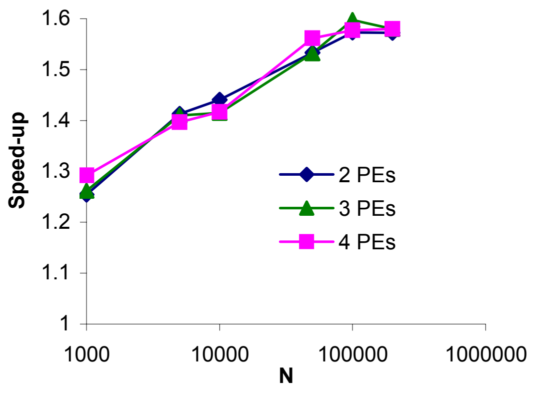
Caches are not the only reason for the noted super-linear behavior. Speed-up is also influenced by the internal parallelization of the kernel of the operating system. Our further experiments show that when a sequential algorithm runs only on the first processor (like in our testing), it takes about 4%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

### 7.8.2 The Results of Pessimistic Method

*Figure 7.18* shows the achieved speed-up for pessimistic method for uniform data sets.



*a)* $E^2$



*b)* $E^3$

**Figure 7.18:** *Speed-up of pessimistic method - Dell Power Edge 7150 (64 bits, 4 PEs).*

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, use of more than 2 PEs in $E^3$ makes no sense.

### 7.8.3 The Results of Optimistic Method and Burglary Method

*Figure 7.19* 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 also with the growing number of used threads. In burglary method, the transactions are avoided. It, indeed, has 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's why burglary method achieves slightly higher speed-up than optimistic method in $E^2$ and lower speed-up in $E^3$.

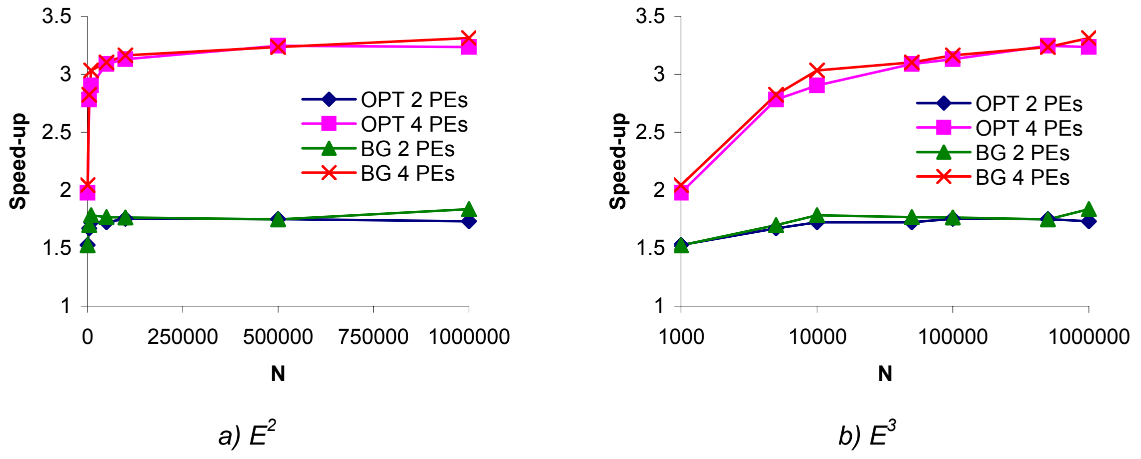
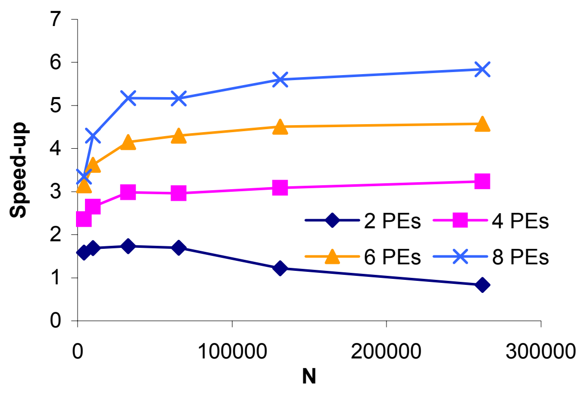
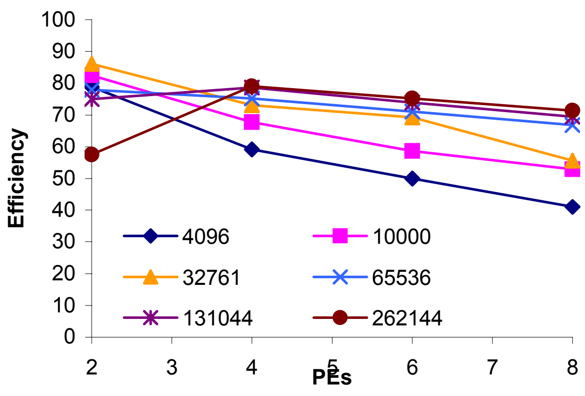


**Figure 7.19:** *Speed-up of optimistic method (OPT) and burglary method (BG) - Dell Power Edge 7150 (64 bits, 4 PEs).*

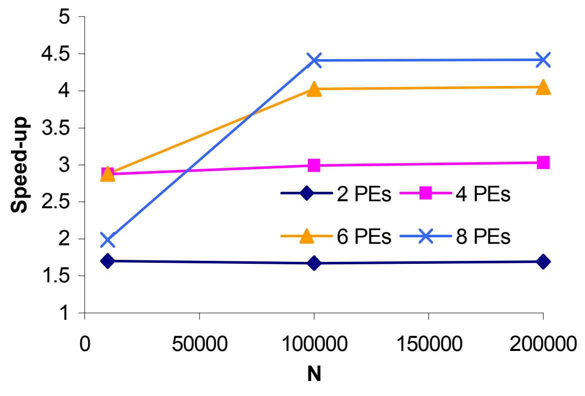
According to our previous experiments [Kol02, Koh03], both methods should be scalable at least up to 8 PEs. As we have no longer access to an appropriate multiprocessor, let us present the results of experiments with older versions of our algorithm. Since that time we have reduced the number of required synchronization and, therefore, one can expect better results using the current version. *Figure 7.20* acknowledges the scalability of the optimistic method.



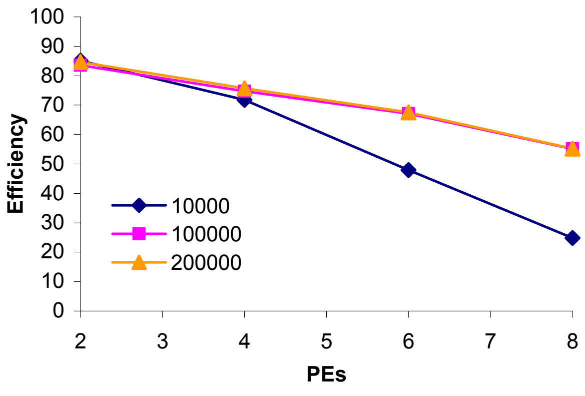
*a) the speed-up in $E^2$*



*b) the efficiency in $E^2$*



*c) the speed-up in $E^3$*



*d) the efficiency in $E^3$*

**Figure 7.20:** *The 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 $E^3$.*

### 7.8.4 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 7.21* presents 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 decreases with the increasing number of used threads. The reasons for the decrease are two: the complexity of the geometric tests and also 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.

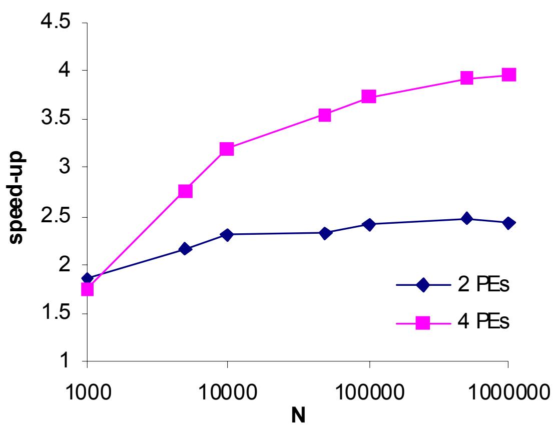


**Figure 7.21:** *Speed-up of circle-circum method - Dell Power Edge 6400.*

### 7.8.5 Experiments with Different Data Point Distributions

Let us discuss the influence of various distributions on the run of our methods. *Figure 7.22* shows samples of Delaunay triangulations for different data point distributions in $E^2$.

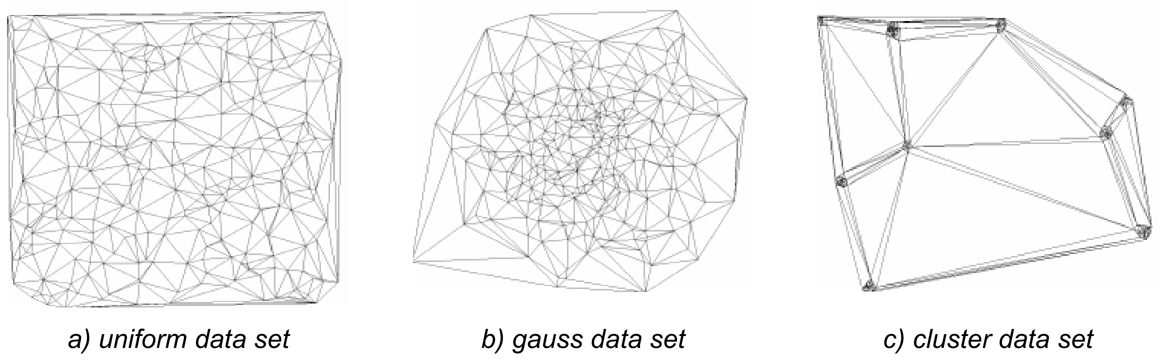


**Figure 7.22:** *Examples of triangulations of different types of data point distribution in $E^2$.*

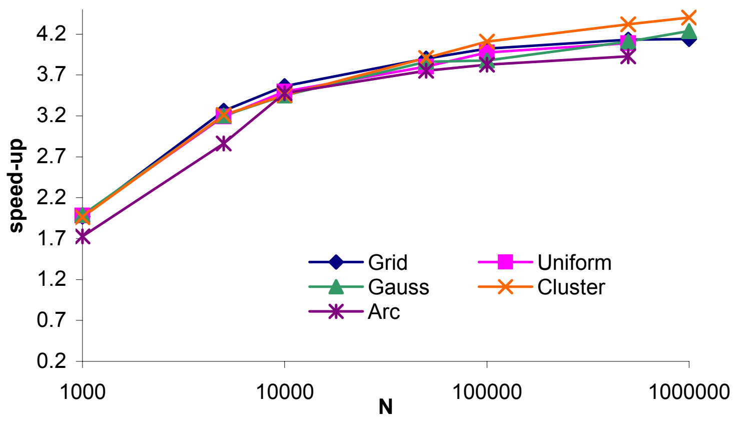


**Figure 7.23:** *Speed-up of the optimistic method in $E^2$ for different types of data point distribution - Dell Power Edge 6400 (4 PEs).*

*Figure 7.23* 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, Koh03a).

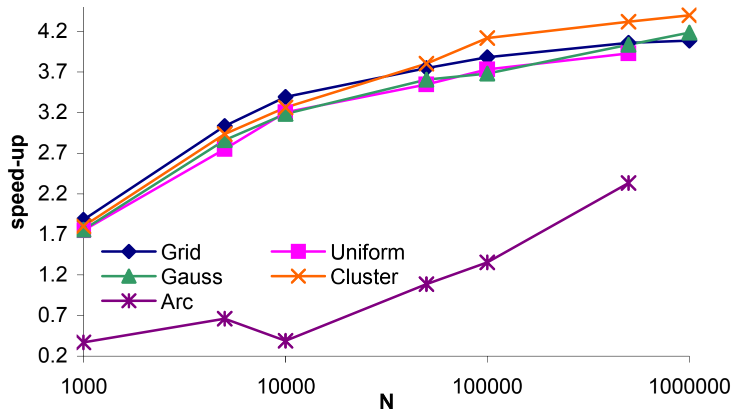
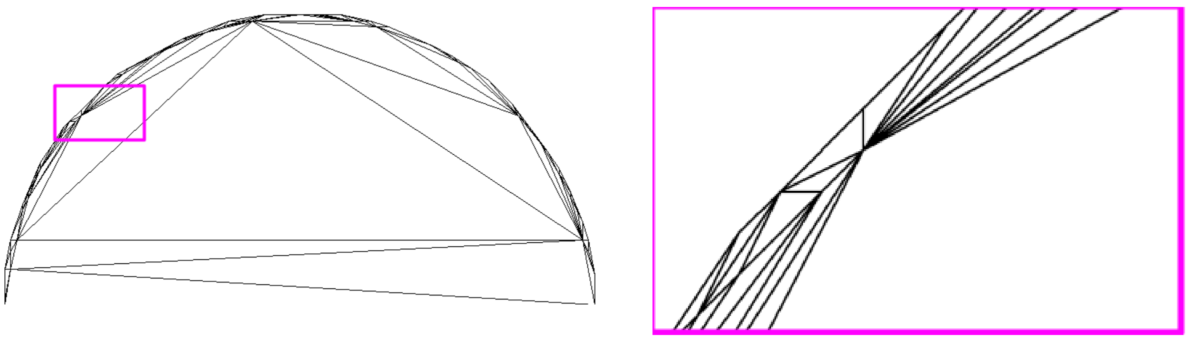


**Figure 7.24:** *Speed-up of the circum-circle method for different data point distributions - Dell Power Edge 6400 (4 PEs).*

A slightly different situation occurs when circum-circle method is tested – see *Figure 7.24*. 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 7.25* shows an example of the Delaunay triangulation of arc data set.



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

**Figure 7.25:** *An example of the Delaunay triangulation of 100 points lying on arc.*

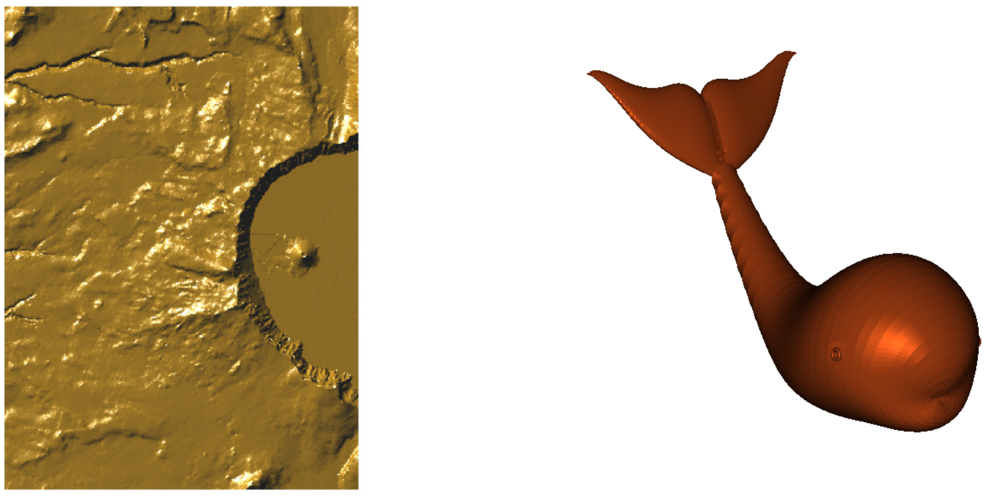
On the contrary, when dealing with cluster data sets, the possibility grows slower than when dealing with other distributions. It is caused by the fact that insertion of a point into one cluster barely influences another cluster, thus if the thread $T_0$ works with a cluster $A$ and the thread $T_1$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

Pessimistic, optimistic and also 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 7.8). However, the grid points are not a typical input - if the user prefers a structured mesh, he will probably not use Delaunay triangulation.

### 7.8.6 Experiments with Real Data Sets

Besides artificial data, we tested real data sets from [Sta99, Žal99]. 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 (data sizes) are compared.

*Figure 7.26a* 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].



a) $E^2$ b) $E^3$

**Figure 7.26:** *Examples of triangulations of real data sets.*

*Figure 7.26b* 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 7.26b* was obtained from the resulting triangulation by the method publicated in [Var03].

### 7.8.7 Subdivision of Input Points among Threads

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.

The optimal efficiency of all optimistic methods is reached when a thread does not need to access a node locked by another thread. To achieve efficiency as close to the optimal efficiency as possible, we have to minimize the total time spent in the waiting loop of the locking routine, i.e. we have to minimize the number of cases when a thread needs to access the node locked by another thread. Usually the thread accesses only the nodes that store simplices lying closely to the position of currently inserted point. Therefore, better strategy 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 thread then form just one fragment, i.e. a continuous area (volume).

The simplest way is to subdivide input points into *k* slabs in x-coordinate. To do this, we use a modified sequential algorithm for median computation[1]. Another possibility is to apply median subdivision on the distances of input points from the origin, i.e. $x^2 + y^2$ or $x^2 + y^2 + z^2$. *Figure 7.27* brings an example of the various strategies of the subdivision of input points among four threads.

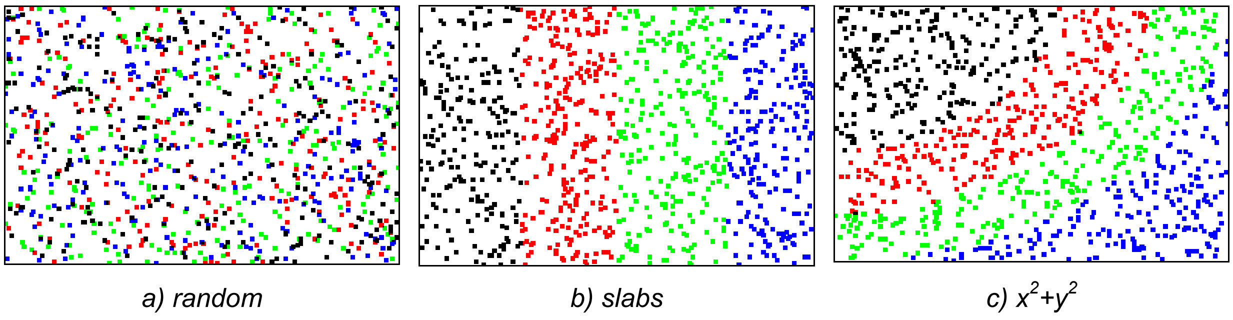


**Figure 7.27:** *Various strategies for the subdivision of input points among four threads.*

All possibilities, indeed, require additional time. *Figure 7.28* compares average number of fragments obtained using these strategies. We tested uniform data sets such that their computation does not need more than 512 MB of physical memory (because of used Shalla multiprocessor). For all that, the presented values are sufficiently representative. As *Figure 7.28* shows, the simplest way (i.e. slabs) should achieve also the highest efficiency.

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

| Threads | N | Number of fragments in $E^2$ | | | Number of fragments in $E^3$ | | | |
|---|---|---|---|---|---|---|---|---|
| | | random | slabs | $x^2 + y^2$ | random | slabs | $x^2 + y^2$ | $x^2 + y^2 + z^2$ |
| 2 | 1000 | 50.7 | 1.2 | 1.0 | 48.5 | 6.0 | 5.0 | 4.5 |
| | 10000 | 505.7 | 1.5 | 1.3 | 416 | 15.5 | 13.0 | 20.5 |
| | 25000 | 1289.0 | 1.5 | 1.5 | 1017.5 | 19.5 | 32.5 | 35.0 |
| | 50000 | 2527.0 | 1.7 | 1.8 | 2128.5 | 36.5 | 36.0 | 62.5 |
| | 100000 | 5101.5 | 2.3 | 2.2 | N/A | N/A | N/A | N/A |
| | 150000 | 7709.2 | 2.3 | 3.0 | N/A | N/A | N/A | N/A |
| | 250000 | 12742.0 | 3.7 | 3.7 | N/A | N/A | N/A | N/A |
| | 350000 | 17785.8 | 4.3 | 4.2 | N/A | N/A | N/A | N/A |
| | 500000 | 25329.3 | 5.2 | 4.7 | N/A | N/A | N/A | N/A |
| 8 | 1000 | 57.0 | 1.3 | 1.1 | 452.3 | 34.7 | 58.0 | 30.7 |
| | 10000 | 593.8 | 2.0 | 1.8 | 4910.7 | 80.7 | 83.0 | 67.0 |
| | 25000 | 1479.0 | 1.9 | 1.9 | 12211.7 | 143.3 | 142.3 | 119.0 |
| | 50000 | 2984.1 | 2.1 | 2.8 | 24698.0 | 211.3 | 247.7 | 196.0 |
| | 100000 | 5973.3 | 3.4 | 3.1 | N/A | N/A | N/A | N/A |
| | 150000 | 9025.9 | 4.3 | 3.8 | N/A | N/A | N/A | N/A |
| | 250000 | 15145.4 | 5.1 | 4.8 | N/A | N/A | N/A | N/A |
| | 350000 | 21230.5 | 5.0 | 6.7 | N/A | N/A | N/A | N/A |
| | 500000 | 30339.0 | 6.6 | 7.3 | N/A | N/A | N/A | N/A |

**Figure 7.28:** *Comparison of total number of fragments achieved by optimistic method with various strategies for the subdivision of input points among the threads for uniform data sets - Shalla (2 PEs).*

Our further experiments [Kol02, Koh03, Koh03a], however, show that the additional time needed for a more sophisticated subdivision is not counterbalanced in the computation and, therefore, the use of such possibilities brings worse results. *Figure 7.29* brings an acknowledgment of this behavior.

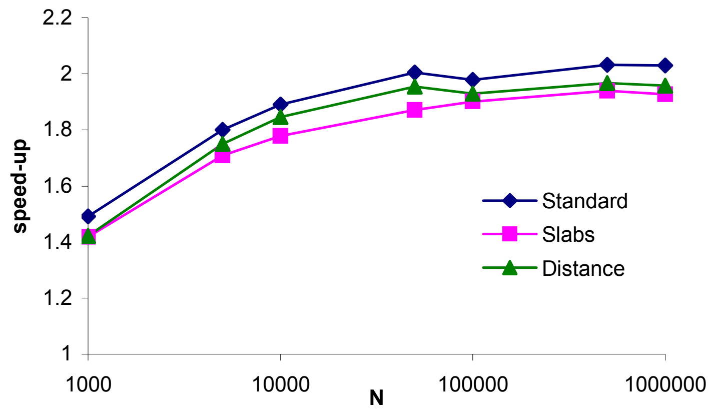


**Figure 7.29:** *The influence of subdivision of the input points among threads on the achieved speed-up for the circum-circle method. Uniform data sets were tested on a Dell P 410 (with 2 PEs).*

The explanation is simple: the number of cases when a thread has to wait is negligible (especially for large data sets), i.e. the total time spent in the waiting is lower than the additional time. For example, optimistic method in $E^3$ calls the locking routine about 27 times per one point (almost does not depend on the size of the data set). During these calls (MA) it accesses about 226 nodes; 62.8% of them were locked in some previous call (AL), 37.1% are locked without waiting (F) and only in about 0.03% cases a thread has to wait (W) for a short time. The details are given in *Figure 7.30*.

| N | MA/N | AL/N | F/N | W/N |
|---|---|---|---|---|
| 1000 | 25.80 | 134.62 | 79.95 | 0.202 |
| 10000 | 27.27 | 143.26 | 84.60 | 0.0321 |
| 25000 | 27.47 | 144.48 | 85.22 | 0.015216 |
| 50000 | 27.62 | 145.36 | 85.68 | 0.008452 |

**Figure 7.30:** *Analysis of the nodes accessed by optimistic method in E3 during the calls of the locking routine - Shalla (2 PEs). MA - the number of the calls of the locking routine, AL - the number of nodes already locked by the thread in any previous call of the locking routine, F - the number of free nodes, W - the number of nodes that are locked by another thread.*

Nevertheless, some of these sophisticated subdivision of input points among the threads (if a parallel version of median computation were used) could be a way how to further increase the efficiency of some methods (e.g. optimistic), although the increase would not be substantial.

### 7.8.8 Comparison and Summary

We have described several parallel methods for the construction of the Delaunay triangulation based on randomized incremental insertion with local improvements. All methods are designed for multiprocessors with shared memory and limited number of processors, typically 2 or 4. According to our experiments we found the methods almost insensitive to the point distribution of input data set. The implementation of the developed methods is available as a part of the Modular Visualization Environment [MVE] – a software package developed at our institute. The methods differ in the complexity of implementation, scalability and also efficiency. Batch method is limited to few processors only and usable in $E^2$ only, however, it is very efficient and, moreover, simple to be implemented. Pessimistic method is the simplest method to be implemented, however, inefficient. Optimistic 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 efficiency of all methods are summarized in *Figure 7.31* – 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.

| 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 7.31:** *Comparison of all methods (higher value means higher evaluation).*

According to these results, it is quite clear that each method has its pros and cons and, therefore, a researcher should use a method the most suitable for his or her purpose. If a general solution is required, optimistic method seems to be the best choice. The comparison of the best speed-ups achieved by optimistic method and speed-up achieved by other existing parallel algorithms is given in *Figure 7.32*.

| **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 7.32:** *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.*

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 5. 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 also differs, it is impossible to say without any doubt whether our optimistic method is more or less efficient than other one. However, let us claim that the results achieved by our optimistic method are at least comparable.

# 8 Conclusion and Future Work

The first task of this thesis was to investigate parallel techniques used in the computer graphics and computational geometry. It includes the possibilities of even distribution of the work over processors and internal data structures used in parallel algorithms. We focused on parallel rendering because the problem of rendering is well-known and there exist a lot of various parallel strategies. We tried to give a survey of the state of the art in the rendering including samples of current parallel systems.

Next, we discussed sequential principles how to construct the Delaunay triangulation in $E^2$ and $E^3$ – one of the fundamental problems in Computer graphics and described the best-known existing parallel algorithms. We proposed several parallel algorithms based on the incremental insertion with local transformations (i.e. the quality of the resulting mesh is ensured) for parallel systems 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. Some of the proposed algorithms are easy to be implemented but not very efficient (e.g. the pessimistic method), while some of them prove opposite characteristics (e.g. the optimistic method). Some of them are usable in $\mathrm{E}^2$ only (e.g. the batch method), other work in $\mathrm{E}^3$ as well. We implemented the 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 that the simplest one – i.e. our hint is: make your solution as simple as possible.

The first objective of our future research is to develop a parallel system suitable to construct the Delaunay triangulation of very big data sets (hundreds million in $E^2$). This system should be composed only of the cheap components produced for a mass market, i.e. common computers interconnected via a network into a cluster. We need to ensure that the points may arrive on-line and the entire mesh can be simply extracted after a single insertion of a point and to ensure the quality of generated triangulation. Therefore, an algorithm by the incremental insertion is welcome. It is clear that the stress should be put also on the scalability and efficiency of the proposed algorithm.

There are two more objectives to be solved. One is theoretically oriented and another one is application oriented. The theoretical objective consists of three different goals. First, as our current results show, we cannot rely on speed-up evaluation of algorithms because the same implementation of a parallel algorithm with the same input data set may need different time on different computers (see Section 7) and, therefore, any comparison is difficult. Therefore, theoretically based complexity of our algorithms independent on the architecture and the platform of the operating system should be calculated. Moreover, if we are able to estimate the complexity of already existing parallel algorithms, e.g. Hardwick [Har97], the comparison of our algorithms with these will be more precise.

Next, all our parallel algorithms described in previous sections and distributed algorithms intended to be implemented (they will be described later) exploit the DAG structure. The key issue is what changes must be done in our algorithms if anyone decides to replace the DAG by another structure for the location (e.g. Devillers [Dev98]) and how this replace will affect the efficiency of algorithms. According to the preliminary results, the changes in the algorithm are usually not significant and, therefore, the efficiency is left intact. Further research is, however, still needed. We would like also to investigate the possibility of the use of the same techniques from our algorithms in any other algorithm that exploit the DAG or tree-like data structure.

Finally, there exist applications that need to construct the Delaunay triangulation in higher-dimension. Typically, these applications deal with the topology problems [Gar03, Krä02]. As it was shown in Section 6, the generalization of the incremental insertion with local transformations from $E^2$ to $E^3$ was straightforward. Is it possible to find out similar generalization for $E^4$ or even for any higher dimension? Is there any way to modify any of our parallel algorithms so that it is capable to handle any dimension? As this is a great challenge and an intensive research is expected, we are not sure that we will succeed in this matter.

The application oriented objective consists of two different goals. First, the proposed parallel system should be integrated into the algorithm for the surface reconstruction proposed by Varnuska et al. [Var03]. Let us briefly describe their approach. In the modeling and virtual archeology, the points on the surface of a real object (very often historical monuments or factory equipment) are obtained (e.g. by a laser scanner or from a sequence of photos). The goal is to create a digital model from these points scattered in space. One popular and relatively robust solution is to construct the Delaunay triangulation and then check all tetrahedra in order to select such tetrahedra that probably intersect the surface of the model. From the selected tetrahedra, the triangles on the surface are refined. The approach in [Var03] is based on the same sequential algorithm as our. The authors found out that the Delaunay triangulation consumes the largest part in their approach and, moreover, its hunger for memory limits the available size of input points to several hundred thousands. It is impossible for them to reconstruct surface of complex models such the statue of David [Mich] is (see *Figure 3.1*). Therefore, the integration of our distributed Delaunay triangulation could help them to overcome these drawbacks.

The last goal is to integrate our parallel system into the approach by Hlavatý et al. [Hla03]. The authors have developed a system for model classification. For a given model, they calculate a vector of its characteristic features and then the database of stored models is searched in order to find models with similar features. Such a tool is useful for designers or animators. In the Delaunay triangulation, an edge interconnects the nearest points. Let us assume we can map the vector of features to the coordinates of a point in $E^3$, then it is possible to construct the *DT* and we can very efficiently find all models related to the given one. In a case that we cannot find such a mapping function, we may use an implementation of the generalized *DT* (see theoretical objectives). At present, no further investigation of the classification problem was done and, therefore, we are not sure that we will succeed in this matter.

The first objective is the most important and also it refers to the nearest future. Let us, therefore, describe the possibilities how to solve this objective in more detail. Straightforward solution is to modify the optimistic method for a system with virtual shared memory [Kea94, Bad90]. *Figure 8.1* shows the schema of such a system. There are *k* processors and each of them has assigned a unique identification (numbered from one). The DAG structure is distributed over these processors and each node in this data structure is fully identified by the ID of PE and its local pointer (i.e. any 32-bits pointer in the original data structures was replaced by this pair). The pair {any ID, 0xFFFFFFFF} denotes the root of the DAG, which is always stored in the local memory of the first processor (i.e. the processor with the number one), and the pair {any ID, 0x00000000} denotes an invalid node.

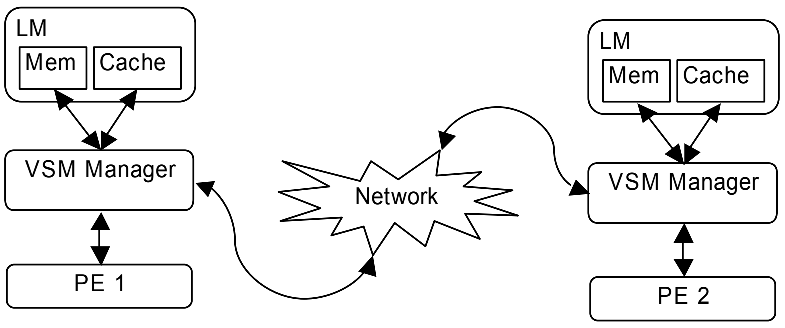


**Figure 8.1:** *Schema of a parallel system with virtual shared memory.*

Whenever a processor needs to access a valid node, it has to call the VSM layer to get an appropriate pointer on this node. In this call, the processor has to specify whether it requires the node for the read-only or for the read-write access. If the processor is going to modify the node, the VSM automatically tries to lock it. In a case that the deadlock has been detected, the processor gets NULL pointer and it has to perform rollback. When the processor finishes the insertion of its point, it calls the VSM for any locked node to complete the modification of the Delaunay triangulation and to unlock this node.

Let us assume that a processor has called the VSM to get the pointer on the node $S_0$ identified by a pair $\{ID, ptr\}$ for read-only purpose. The VSM checks whether *ID* is equal to zero or to the number assigned to the processor. If the outcome is positive, the requested node is stored in the local memory of this processor and, therefore, the VSM simply returns *ptr*. In the opposite case, the node is stored on a remote computer identified by *ID*. At first, the cache is searched in order to find a local copy of this node. If no appropriate entry is found in the cache, the VSM sends a request to the remote computer to get the missing node and waits until the reply comes. The received node is, afterwards, placed into the cache. Let us note that LRU (Least-Recently-Used) policy is used.

When the pointer on the node $S_0$ is needed for read-write purpose, the situation is more complex. If the node is stored locally, standard locking routine of the optimistic method is used. In the opposite case, the remote computer is contacted (the cache is never checked) and it uses the same standard locking routine to get an exclusive access to the given node for the initiating processor. The initiator waits until the reply comes. The received node is, afterwards, placed into the cache. The processor is responsible for calling the VSM for the node $S_0$ after the transaction (see the optimistic method) was committed. If the node is stored locally, it is just unlocked, otherwise it is sent to the remote processor, which updates the appropriate local node by the received data and unlocks this node.

There is no doubt that this is not very efficient solution because the intensive network communication cannot be avoided. Probably the amount of the network traffic could be reduced if the points are subdivided into *k* groups (where *k* is the number of processors) in such a manner that the equal number of points in each group is ensured and also the bounding boxes of these groups have minimal intersection – see Section 7.

Let us consider another distributed solution. The basic idea is to separate the input points into disjunctive regions and distribute these regions over processors. Bucketing strategy combined with Mueller's algorithm [Mue95] handle the problem how to subdivide evenly the input set of points. Each processor is responsible for the triangulation of its points. The synchronization among the processors for simplices at the boundaries of the regions is necessary.

Straightforward approach is to let each processor to triangulate its points independently and then merge the local triangulations into the result. There are two problems with this approach. First, the merging of two local triangulations is limited to just one processor and thus it negatively influences the overall efficiency of the algorithm. Next, it violates our request to have a valid triangulation in each step of the construction. It is also impossible to use strategy similar to DeWall algorithm, i.e. to construct the joint first, because we would like to ensure that the points may arrive on-line.

Therefore, we have to retain a distributed triangulation and update it simultaneously. Because of our good experience with the DAG, we use the distributed version of this data structure. Since now, let us consider the construction of the Delaunay triangulation in $E^2$ only; the generalization to higher dimension is simple.

Let us assume we already somehow obtained a distributed triangulation and that this triangulation is distributed over two processors labeled *Worker 1* (briefly $W_1$) and *Worker 2* (briefly $W_2$), where the processor $W_1$ is responsible for the left rectangular area and the processor $W_2$ for the right one – see *Figure 8.2*.

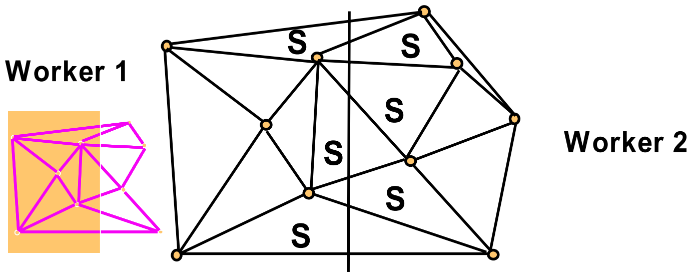


**Figure 8.2:** *An example of the Delaunay triangulation distributed over two processors (workers).*

From the point of view of the $W_1$, there are three kinds of triangles in this triangulation to be considered:

- The *local triangles* are fully inside the workspace of the processor $W_1$ and, therefore, they are stored in the local memory of this processor. No synchronization is required to access them. The local triangles are the most numerous group.
- The *shared triangles* are partially inside the workspace of the processor $W_1$ and partially inside the workspace of the processor $W_2$. They are stored in the local memories of both processors. Simultaneous modification of these triangles has to be avoided and, therefore, some synchronization is necessary (will be discussed later). There is about $\sqrt{N}$ shared triangles in the triangulation, where $N$ is the number of points in an uniform data set. *Figure 8.3* acknowledges this.
- The *remote triangles* are stored in the local memory of the processor $W_2$ only. However, when the processor $W_1$ modifies a shared triangle, it has to know the identification of the remote triangle adjacent to the shared one in order to update the neighborhood correctly. Again some synchronization is necessary.

N
√N
√N
a) E2
N
3√N
3√N
3√N
b) E3

**Figure 8.3:** *The cut of an object containing N uniformly distributed points has approximately square root of N points in its vicinity.*

The same categorization of the nodes can be done from the point of view of the second processor $W_2$. Now, let us assume that the processor $W_2$ wants to insert the point $p$ into the triangulation and this point lies inside the shared triangle $S_0$ as it is shown in *Figure 8.4*. Both processors have to synchronize their work, which involves sending of the index and the coordinates of the point $p$ and of the identification of triangle $S_0$ to the processor $W_1$. Afterwards, both processors subdivide $S_0$ into three new triangles and update the neighborhood with triangles $S_1$ and $S_2$. The $W_1$ has to update neighborhood with $R_0$ as well.

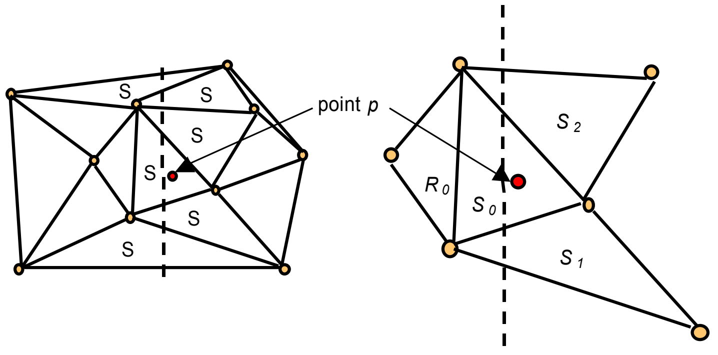


**Figure 8.4:** *An example of the insertion of the point p lying inside a shared triangle – see text.*

We have just introduced two problems. First, a processor has to store not only its local points but also remote points received during the process. Therefore, it is needed to have a hash table that maps the index of a point to a pointer on the structure storing the coordinates of this point. For a given point, it is also necessary to determine efficiently whether the point is local or remote one. Therefore, we use the highest bit of the indices of points to distinguish them. This allows us to apply an efficient test on a triangle in order to find out whether this triangle is shared (at least one of its vertices is a remote point) or local.

Another problem is how to identify a triangle. We cannot use the pair ID of PE and pointer on the appropriate node as in the VSM approach because of shared triangles. Let us, therefore, assign an integer to each node. Shared triangles are identified by negative values (assigned synchronously), local triangles by a pair ID of PE and positive values (assigned locally). We can either store the full 64-bits identification in all nodes or used only 32-bits identification and translate the short form of identification into the full identification only if it is necessary. As the DAG structure consumes a lot of memory generally and the most of triangles are local, the second possibility seems to be better.

Let us assume an extra processor called *Interface* responsible for retaining connectivity among local triangulations of the processors, i.e. for the translations. It needs to store shared triangles to perform the translations. *Figure 8.5* shows an example of the part of triangulation distributed over two workers and the interface.

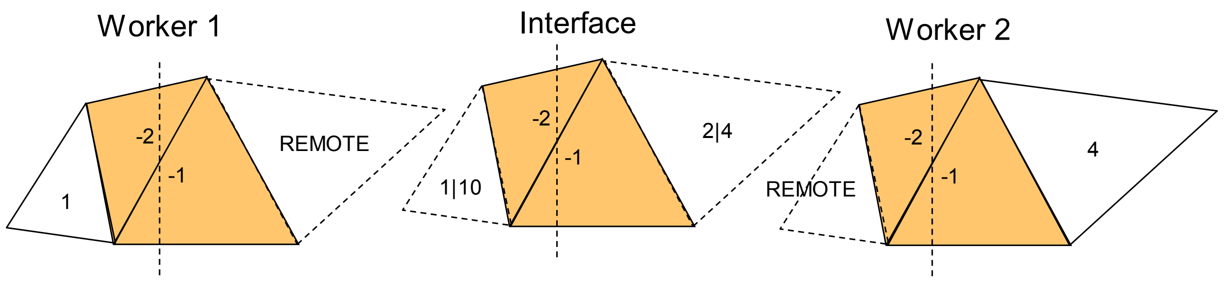


**Figure 8.5:** *An example of a part of triangulation. Dashed triangles are not physically stored in the local memory of that computer. A number (or label Remote respectively) in the triangle shows its identification known on that computer.*

The interface has also other functionality. At the beginning, this processor distributes the points over the processors called *Workers*, constructs the big auxiliary triangle (shared one) and broadcasts it to all processors. Whenever a worker needs to subdivide or legalize a shared triangle, it calls the interface. Interface enters a critical section (i.e. only one synchronized operation can be performed), performs the subdivision or the legalization of this triangle (the original triangle is removed from memory), contacts all participating workers sending them a command to perform the changes and leave a critical section. The worker has to wait until the operation is completed. The performance of the proposed algorithm can be increased if the worker processes the insertion of another point instead of the waiting. *Figure 8.6* shows a scheme of the DAG structure in this approach.

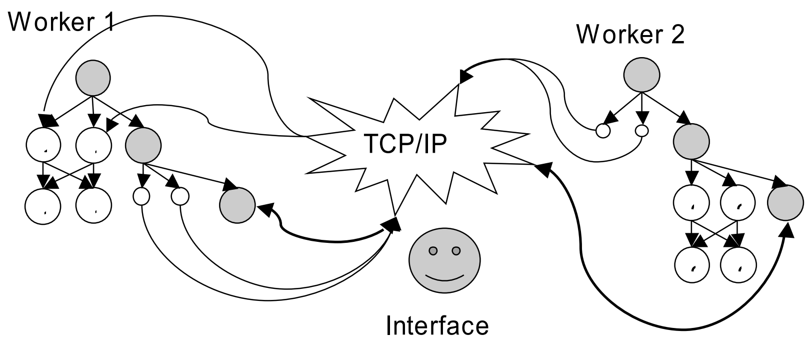


**Figure 8.6:** *The scheme of the DAG structure in the interface-workers approach. The gray nodes denote shared simplices.*

When the resulting triangulation is required, each worker extracts its local triangles and the interface extracts its shared triangles. The union of these triangles gets the requested Delaunay triangulation of up to $2 \cdot 10^9$ points.

The solution that we have just described suffers from several problems. First, the efficiency of the algorithm drops down if the requests on the interface cannot be handled immediately because the interface is busy by handling a previous request. As the number of requests grows with the increasing number of workers, the overall number of processors is very limited.

Another problem is that each shared simplex requires at least triple computational effort than any local simplex. The efficiency can be improved by reducing the number of requests generated on the interface by moving part of its functionality onto the processors.

Let us consider another approach. This, in our opinion very efficient, approach is based on the previously described solution. The main difference is that there is no interface, i.e. all processors have to handle all cases. For each triangle, we assign just one processor as an administrator. A triangle can be administrated only by a processor that contains at least one vertex of this triangle in its area. This means that while for a local triangle, there is only one candidate for the administrator, for shared simplices, we have several options – we prefer the processor that occupy minimum administration posts.

The processor can access the triangle only if it is an administrator of this triangle. In a case that the administrator has changed the shared triangle, it sends an asynchronous message about the change to all processors sharing this triangle. These messages are handled by the receivers when they are in an idle status or they have to face the problem of the inconsistency.

If a shared triangle is required by a processor that is not its administrator, this processor either sends asynchronously its point to be inserted to the appropriate administrator or it applies for the administrative post of the triangle in a synchronous way. Let us note that the first possibility is available for the subdivision phase only.

# 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.

[Ake01] Akeley K, Hanrahan P. Real-time graphics architectures. http://graphics.stanford.edu/courses/cs448a-01-fall/

[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.

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

[ATI] ATI Technologies Inc., http://www.ati.com

[Bad90] Badouel D, Priol T. An efficient parallel ray tracing scheme for highly parallel architectures. In: Advances in computer hardware v. rendering, ray tracing audiovisualisation systems. Springer-Verlag, New York, 1990. p. 93-106

[Bad94] Badouel D, Bouatouch K, Priol T. Distributing data and control for ray tracing in parallel. IEEE Computer Graphics and Applications 1994; 14(4):69–77

[Bat99] Battiato S, Cantone D, Catalano D, Cincotti G, Hofri M. An efficient algorithm for the approximate median selection problem. In: Proceedings of Fifth 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.

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

[Cas89] Caspary E, Scherson ID. A self-balanced parallel ray-tracing algorithm. In: Parallel Processing for Computer Vision and Display, Addison-Wesley, 1989. p. 408-419.

[Cor02] Correa WT, Klosowski JT, Silva CT. Out-Of-Core Sort-First parallel rendering for cluster-based tiled displays. In: Proceedings of Fourth Eurographics Workshop on Parallel Graphics and Visualization, 2002.

[Cha02] Chalmers A, Davis TA, Kato T, Reinhard E. Practical parallel rendering. In: Courses of SIGGRAPTH 2002, San Antonio, U.S.A, 2002. course 22

[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 5 th 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.

[Cor97] Cortés A, Ripoll A, Senar MA, Luque E. Dynamic load balancing strategy for scalable parallel systems. In: Proceedings of PARCO'97, 1997. p. 735-738

[Cox94] Cox M, Hanrahan P. A distributed snooping algorithm. IEEE Parallel and Distributed Technology 1994; 2(2):30-36

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

[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.

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

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

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

[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 2$^{nd}$ Annual Symposium on Computational Geometry, ACM, 1986. p. 276-284.

[Eld01] Eldridge M. Designing graphics architectures around scalability and communication. Dissertation, Stanford University, 2001.

[Ell94] Ellsworth D. A new algorithm for interactive graphics on multiprocessors. IEEE Computer Graphics and Applications 1994; 14(4):33-40

[Ell96] Ellsworth D. Polygon rendering for interactive visualization on multicomputers. Dissertation, University of North Carolina at Chapel Hill, 1996.

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

[Fuc89] Fuchs H, Poulton J, Eyles J, Greer T, Goldfeather J, Ellsworth D, Molnar S, Turk G, Tebbs B, Isreal L. Pixel-Planes 5: A heterogeneous multiprocessor graphics system using processor-enhanced memories. In: Proceedings of SIGGRAPH 1989, July 1989. p. 79-88

[Fus82] Fussel D, Rathi BD. A VLSI-oriented architecture for real-time raster display of shaded polygons. In: Graphics Interface '82. p. 373-380

[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.

[Gre90] Green SA, Paddon DJ. A highly flexible multiprocessor solution for ray tracing. The Visual Computer 1990; 6(2):62-73

[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.

[Han96] Hansen C, Krogh M, Painter J, De Verdiere GC, Troutman R. Binary-swap volumetric rendering on the T3D. In: Cray Users Group Conference, Denver, March 1995.

[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.

[Hei97] Heirich A, Arvo J. A competitive analysis of load-balancing strategies for parallel ray tracing. The Journal of Supercomputing 1998; 12(1&2):57-68

[Hla03] Hlavatý T, Skala V. 3D object classification and retrieval. Technical report of Department of Computer Science and Engineering, University of West Bohemia, Czech Republic, 2003

[Hol98] Larson GW. The Holodeck: A parallel ray-caching rendering system. In: Proceedings of Eurographics Workshop on Parallel Graphics and Visualization, 1998.

[Hop96] Hoppe H. Progressive meshes. In: ACM Computer Graphics Proceedings of Annual Conference Series, ACM Press, 1996. p. 99–108

[Hum01] Humphreys G, Eldridge M, Buck I, Everett M, Stoll G, Hanrahan P. WireGL: A scalable graphics system for clusters. In: Proceedings of SIGGRAPH 2001, July 2001.

[Hum02] Humphreys G. Big data in computer graphics. Lecture notes 2002, University of Virginia, Virginia, USA.

[Jev89] Jevans DAJ. Optimistic multi-processor ray tracing. In: Proceedings of CG International '89, Springer-Verlag, New York, 1989. p. 507–522

[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

[Kar94] Karia RJ. Load balancing of parallel volume rendering with scattered decomposition. In: Proceedings of Scalable High Performance Computing Conference, Knoxville, TN, May 1994. p. 252-258

[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

[Kea94] Keates MJ, Hubbold RJ. Accelerated ray tracing on the KSR1 virtual shared-memory parallel computer. Technical Report UMCS-94-2-2, Department of Computer Science, University of Manchester, Oxford Road, Manchester, UK, February 1994.

[Kim96] Kim HJ, Kyung CM. A new parallel ray-tracing system based on object decomposition. The Visual Computer 1996; 12(5):244–253

[Koh03] Kohout J, Kolingerová I. Parallel Delaunay triangulation in $E^3$: Make it simple. The Visual Computer 2003; in the press

[Koh03a] Kohout J, Kolingerová I, Žára J. Practically oriented parallel Delaunay triangulation in $E^2$ for computers with shared memory. *Computer & Graphics 2003; accepted for the publication*

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

[Kur97] Kurc TM, Aykanat C, Öcgüc B. A comparison of spatial subdivision algorithms for sort-first rendering. Lecture Notes in Computer Science 1997; 1225(1997):137-146

[Kur98] Kurc TM, Aykanat C, Öcgüc B. Object-space parallel polygon rendering on hypercubes. Computers & Graphics 1998; 22(4):487-503

[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.

[Lee96] Lee TY, Raghavendra CS, Nicholas JB. Image composition schemes for sort-last polygon rendering on 2-D mesh multicomputers. IEEE Transactions on Visualization and Computer Graphics, September 1996; 2(1996):202-217

[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.

[Li91] Li J, Miguet S. Z-buffer on trasputer-based machine. In: Proceedings of Distributed Memory Computing Conference, Portland, OR, April 1991. p. 315-322

[Ma94] Ma K, Painter JS, Hansen CD, Krogh MF. Parallel volume rendering using binary-swap compositing. IEEE Computer Graphics and Applications, July 1994; 14(4):59-68

[Mac92] Mackerras P. A fast parallel marching cubes implementation on the Fujitsu AP1000. Technical Report TR-CS-92 –10, Department of Computer Science, Australian National University, 1992.

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

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

[Mol91] Molnar S. Combining z-buffer engines for higher-speed rendering. In: Advances in Graphics Hardware III, Springer-Verlag, New York NY, 1988.

[Mol92] Molnar S, Eyles J, Poulton J. PixelFlow: High-speed rendering using image composition. ACM SIGGRAPH Computer Graphics 1992; 26(2):231-240

[Mol94] Molnar S, Cox M, Ellsworth D, Fuchs H. A sorting classification of parallel rendering. IEEE Computer Graphics & Applications 1994; 14(14):23-32

[Mon97] Montrym JS, Baum DR, Dignam DL, Migdal CJ. InfiniteReality: A real-time graphics system. In: Proceedings of SIGGRAPH 1997, August 1997. p. 293-302

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

[Mue97] Mueller C. Hierarchical graphics databases in sort-first. In: Proceedings of IEEE Symposium on Parallel Rendering, October 1997. p. 49-57

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

[Muu95] Muuss MJ. Towards real-time ray-tracing of combinatorial solid geometric models. In: Proceedings of BRL-CAD Symposium'95, June 1995. p. ????

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

[Neb98] Nebel JC. A concurrent dataflow algorithm for ray tracing. In: Proceedings of 16th Eurographics UK Conference'98, University of Leeds, 25-27 March 1998, Leeds, UK

[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.

[NVID] nVidia Corporate, http://www.nvidia.com

[OpenGL] Open Graphics Library, http://www.opengl.org

[Oka92] Okabe, Boots B, Sugihara K. Spatial tesselations: Concepts and applications of Voronoi diagrams. Wiley, 1992. ISBN 0 471 93430 5

[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

[ORT] Open RayTracing, http://www.openrt.de

[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.

[Par99] Parker S, Shirley P, Livnat Y, Hansen C, Sloan PP. Interactive ray tracing. In: Proceedings of Interactive 3D Graphics, April 1999. p. 119-126

[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.

[Ram97] Ramakrishnan CR, Silva CT. Optimal processor allocation for sort-last compositing under BSP-tree ordering. TR 9728, Department Applied Mathematics and Statistics, SUNY Stony Brook, October 1997.

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

[Rob88] Roble D. A load balanced parallel scanline z-buffer algorithm for the iPSC hypercube. In: Proceedings of Pixim '88, Paris, France, October 1988. p. 177-192

[Rom79] Roman GC, Kimura T. A VLSI architecture for real-time color display of three-dimensional objects. In: Proceedings of IEEE Micro-Delcon, 1979. p. 113-118

[Sam99] Samanta R, Zheng J, Funkhouser T, Li K, Singh JP. Load-balancing for multi-projector rendering systems. In: Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware, Los Angelos, California, August 1999. p. 193-197

[Sam00] Samanta R, Funkhouser T, Kai Li, Singh JP. Hybrid sort-first and sort-last parallel rendering with a cluster of pcs. In: Proceedings of SIGGRAPH/Eurographics workshop on Graphics hardware, New York, August 2000. p. 99–108.

[San97] Sano K, Kitajima H, Kobayashi H, Nakamura T. Parallel processing of the shear-warp factorization with the binary-swap method on a distributed-memory multiprocessor system. In: Proceedings of Parallel Rendering Symposium, Phoenix, AZ, U.S.A, October 1997. p. 87-94

[Sch88] Schneider BO, Claussen U. An architecture for rendering in object space. In: Advances in Graphics Hardware III, Springer-Verlag, Berlin, 1988. p. 121-140

[Sch92] Schroeder W, Zarge J, Lorensen W. Decimation of triangle meshes. Computer Graphics 1992; 26(2):65-70

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

[Sch00] Schikore DS., Fischer RA, Frank R, Gaunt R, Hobson J, Whitlock B. High-resolution multiprojector display walls. IEEE Computer Graphics & Applications 2000; 20(4):38-44

[Sch03] Schmittler J, Slusallek P. SaarCOR: A hardware architecture for realtime ray tracing. http://www.saarcor.de, January, 2003. http://graphics.cs.uni-sb.de/~jofis/SaarCOR/SaarCOR-Descr-Engl.pdf

[Sco93] Scopigno R, Paoluzzi A, Guerrini R, Rumolo G. Parallel depth-merge: A paradigm for hidden surface removal. Computers & Graphics 17(1993):583-592

[Sha88] Shaw CD, Green M, Schaeffer J. A VLSI architecture for image composition. In: Advances in Graphics Hardware III, Springer-Verlag, Berlin, 1988. p.183 –199

[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.

[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.

[Sta38] Stamminger M, Haber J, Schirmacher H, Seidel HP. Walkthroughs with corrective textures. In: Proceedings of the 11th Eurographics Workshop on Rendering, 2002. p. 377-388

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

[Tay02] Tay YC. A comparison of pixel complexity in composition techniques for sort-last rendering. Parallel and Distributed Computing 2002; 62:152-171

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

[Tek00] Tekalp AM, Ostermann J. Face and 2-D mesh animation in MPEG-4. 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.

[The89] Theoharis TA. Algorithms for parallel polygon rendering. Springer-Verlag, New York NY, 1989.

[Ude99] Udeshi T, Hansen C. Parallel multipipe rendering for very large isosurface visualization. In: Proceedings of Eurographics-IEEE TCCG Symp.Visualization, Vienna, Austria, May 1999. p. 99-108.

[UNC] UNC Walkthrough Project, http://www.cs.unc.edu/~walk/

[Ups89] Upstill S. The RenderMan companion. Addison-Wesley, Reading MA, 1989

[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

[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.

[Wal01] Wald I, Slusallek P. State of the art in interactive ray tracing. In: State of the Art Reports, Eurographics 2001

[Wal01a] Wald I, Slusallek P, Benthin C, Wagner M. Interactive rendering with coherent ray tracing. In: Procedings of Eurographics 2001, p. 153-164

[Wal03] Wald I, Benthim C, Slusallek P. Distributed interactive ray tracing of dynamic scenes. To appear in: Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics, 2003

[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.

[Wat00] Watt A. 3D Computer Graphics. Addison-Wesley, 2000, ISBN: 0-201-39855-9

[Wei81] Weinberg R. Parallel processing image synthesis and antialiasing. In: Proceedings of SIGGRAPH '81, Dalass, Texas, U.S.A, August 1981. p. 55-62

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

[Whi92] Whitman S. Multiprocessor Methods for Computer Graphics Rendering, AK Peters, Wellesley, Massachusetts, 1992.

[Wil93] Willebeek-LeMair M, Reeves AP. Strategies for dynamic load balancing on highly parallel computers. IEEE Transactions on Parallel and Distributed Systems 1993; 4(9):979-993

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

[Žar95] Žára J, Holeček A, Přikryl J, Buriánek J, Menzel K. Load balancing for parallel raytracer on virtual walls. In: Proceedings of WSCG'95, Plzeň, Czech Republic, 1995. p. 439-447

[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

[Koh03] Kohout J, Kolingerová I. Parallel Delaunay triangulation in $E^3$: Make it simple.Visual Computer 2003, Springer-Verlag Heidelberg; 19(7&8): 532-548

[Koh03c] 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

[Koh02] 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

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

[Kol01] 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

- accepted for the publication

[Koh03a] Kohout J, Kolingerová I, Žára J. Practically oriented parallel Delaunay triangulation in $E^2$ for computers with shared memory. Elsevier Pergamon, Computer & Graphics 2003*; accepted for the publication*

- submitted for the publication

[Koh03b] Kohout J, Kolingerová I, Žára J. Parallel Delaunay triangulation in $E^2$ and $E^3$ for computers with shared memory. Elsevier North-Holland, Parallel Computing; *submitted for the publication*

## Not reviewed publications

- related to this work

[Koh01] 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 as the best paper*

[Dou01] Doubek J, Kohout J. Spojenou silou, Systém pro distribuované zpracování - GSD. Chip 10/01 + CD ROM, 2001. p. 154 (*in Czech*)

- other

[Koh99] Kohout J, Mautner P, Zuzák F. Automatická detekce částic v digitálních mikrogramech. In: Proceedings of the *15th Konference s mezinárodní účastí - Výpočtová mechanika '99,* Nečtiny, Czech Republic, 1999. (*in Czech*)

[Koh99a] Kohout J, Mautner P, Zuzák F. Metody zpracování digitálního ferogramu pro tribodiagnostiku. In: Proceedings of the *15th Konference s* *mezinárodní účastí - Výpočtová mechanika '99,* Nečtiny, Czech Republic, 1999. (*in Czech*)

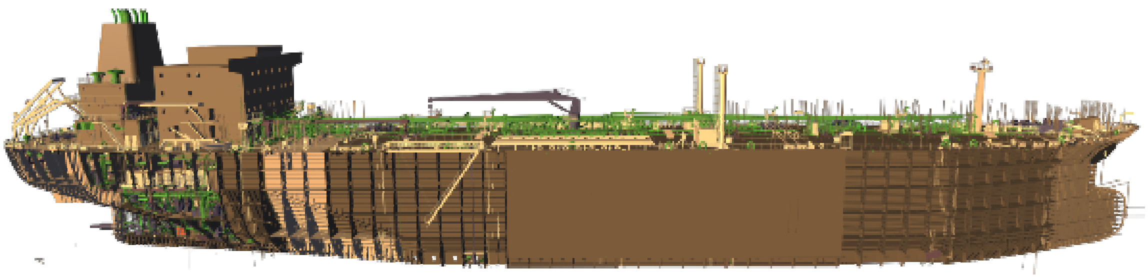
**Presentations and talks abroad**

- January 2004 – Paralelní rendering, presentation (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, TU Chemnitz, Germany

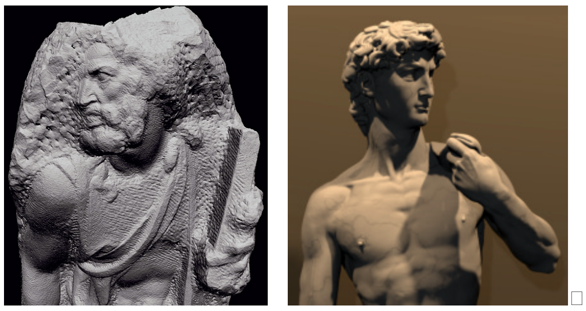
**Stays and other activities**

- September 2003 – passive participation on the EG 2003 conference
- October 2003 – two weeks stay at TU Graz, project Aktion
- May 2003 – one week stay at TU Chemnitz, Socrates-Erasmus teaching mobility
- Spring 2003 – one semester study abroad at Queen's University of Bath, UK

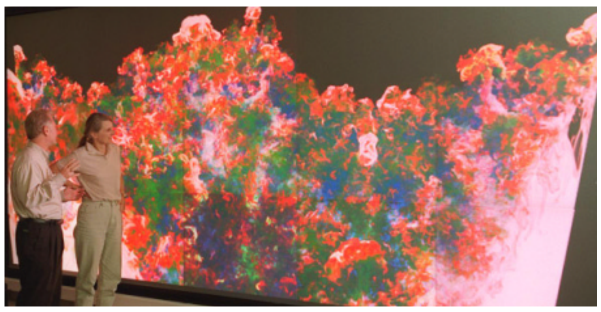
# Appendix: Color figures



*a) double eagle tanker model containing 83 million triangles [UNC]*



*b) scans of Saint Matthew (386 MPolys) and the David (2 GPolys) - Stanford Digital Michelangelo Project [Mich]*



*c) simulation of compressible turbulence (2K x 2K x 2K mesh) - Sean Ahern and Randal Frank, LLNL [Sch00]*

**Figure 3.1:** *An example of big data sets.*

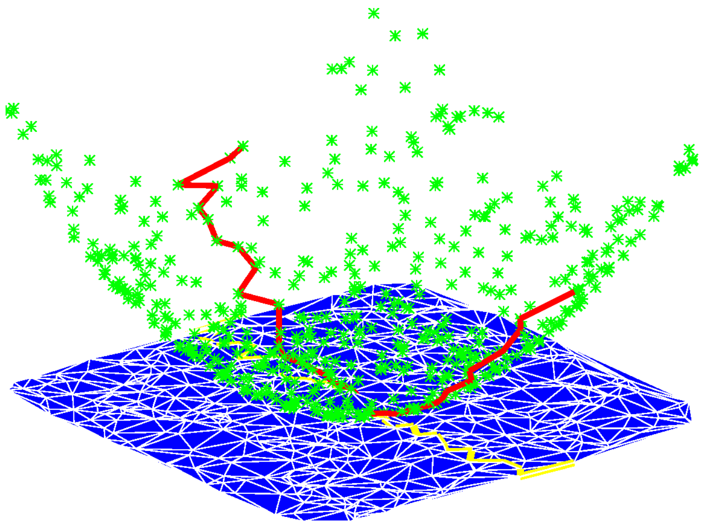


**Figure 4.6:** *The projected points and the corresponding Delaunay triangulation [Har97]. Only a tiny part of the 3D convex hull is shown (red bold line segments).*

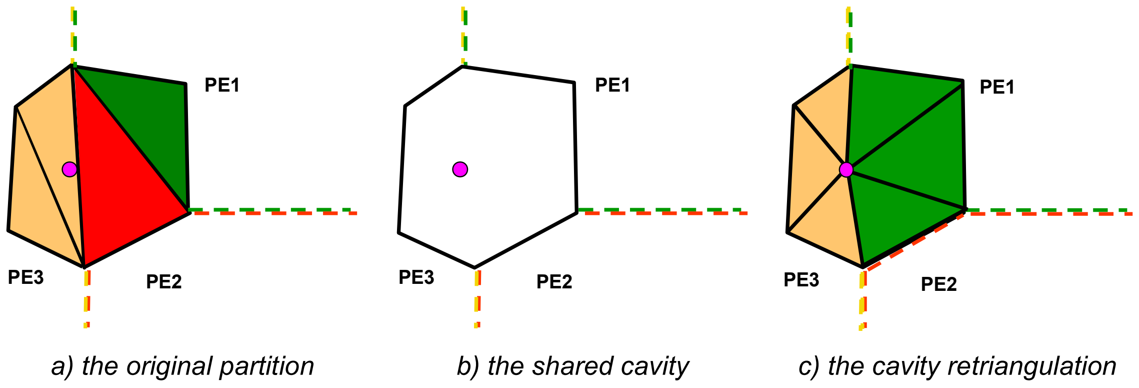


**Figure 5.2:** *The insertion of a point (big dot) into the triangulation in $E^2$. This insertion causes retriangulation of a cavity shared by three processors (PE1, PE2 and PE3) including the update of their regions' boundaries. The simplices belonging to the same region are shown in the same color.*

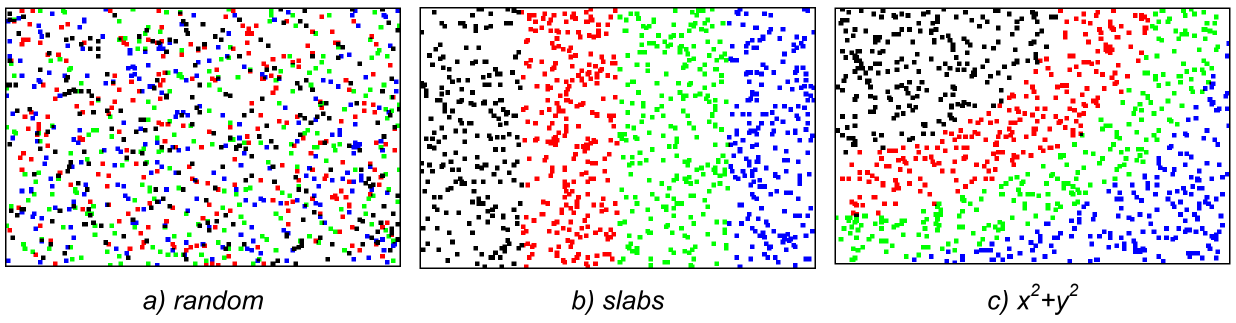


**Figure 7.27:** *Various strategies for the subdivision of input points among four threads.*