Evaluation of Parallelization Strategies for an Incremental Delaunay Triangulator in E^3

P. Cignoni[†], D. Laforenza[†], C. Montani[‡], R. Perego[†], R. Scopigno[†]

[†] CNUCE – Consiglio Nazionale delle Ricerche, Via S. Maria 36, 56126 Pisa, ITALY

[‡] I.E.I. – Consiglio Nazionale delle Ricerche, Via S. Maria 46, 56126 Pisa, ITALY

E-mail: scopigno @ cnuce.cnr.it

June 29, 1994

Abstract

This paper deals with the parallelization of Delaunay triangulation, a widely used space partitioning technique. Two parallel implementations of a three–dimensional *incremental construction* algorithm are presented. The first is based on the decomposition of the spatial domain, while the second relies on the master–slaves approach.

Both parallelization strategies are evaluated, stressing practical issues rather than theoretical complexity. We report on the exploitation of two different parallel environments: a tightly–coupled distributed memory MIMD architecture and a network of workstations cooperating under the Linda environment. Then, a third hybrid solution is proposed, specifically addressed to the exploitation of higher parallelism. It combines the other two solutions by grouping the processing nodes of the multicomputer into clusters and by exploiting parallelism at two different levels.

Keywords: Computational Geometry, Delaunay Triangulation, Parallel Processing, Distributed Computing, Linda.

1 Introduction

Triangulations are a well known topic in Computational Geometry. They are routinely used in a broad range of applications, such as robotics, finite elements analysis, computer vision and image synthesis, as well as in mathematics and natural sciences [1]. Delaunay Triangulation (DT) is a particular type of triangulation and, together with its dual, the Voronoi diagram, is very extensively studied in Computational Geometry. Many algorithms have been proposed to compute DT over a set of sites in Euclidean spaces E^2, E^3 or E^d [22]. It gives a regular partition of the space in triangular cells, in E^2, or tetrahedral cells, in E^3, where these cells are as equilateral as possible.

Volume Rendering [19] is one of the latest applications of DT. A *volume* dataset consists of sampled points or sites in E^3 space, with one or more scalar or vector values associated with each point. The spatial arrangement of the pointset can be either structured, with explicit or implicit topological relations between the sites, or unstructured. In the latter case, the triangulation of the set of points in E^3 is a prerequisite for executing both surface reconstruction and direct volume rendering algorithms [26]. Volume Rendering applications generally cope with a large number of sites and this imposes heavy efficiency constraints on the triangulator used. This also holds for other applications, such as digital terrain modeling [15] where digital maps with $O(100K)$ triangular elements are common.

Thus, the high efficiency of triangulators is strategic and can be achieved both with optimization techniques, which can reduce the expected complexity of the basic algorithm, and by the exploitation of parallelism.

This paper describes the parallelization of a Delaunay triangulation algorithm in E^3 space based on an *incremental construction* approach; its sequential optimized implementation was presented in detail in a previous paper [7]. Three parallel solutions are presented and discussed: the first and second solutions have been implemented both on a hypercube multicomputer, using its message–passing library, and on a network of IBM R6000/340 workstations, using Linda; the third solution, proposed in order to reach a greater scalability, is based on a hybrid approach that exploits parallelism at two different levels. Although it has not been implemented, and only an analytical estimate of the achievable results is reported here, the approach is interesting and certainly more suited than the above for highly parallel environments.

This paper is organized as follows. In Section 2 definitions and a classification of Delaunay triangulation algorithms are given, together with an overview of other works on parallel Delaunay triangulation. In Section 3 we describe the sequential incremental algorithm and the optimization techniques used, followed in Section 4 by the description and discussion of the three parallel solutions. Concluding remarks and some comments on the two parallel environments used are made in Section 5.

2 Delaunay Triangulations

The Delaunay Triangulation Σ [22] defined over a pointset P in E^3 space is the set of tetrahedra (or 3-simplices, using a more formal mathematical notation) such that:

1. a point p in E^3 is a vertex of a tetrahedron in Σ *iff* $p \in P$;
2. the intersection of two tetrahedra in Σ is either an empty set or a common face or edge;
3. the sphere circumscribed around the four vertices of each tetrahedron contains no other point from the set P.

Examples of a non–Delaunay and a Delaunay triangulation over a simple pointset in E^2 are shown in Figure 1. In the figure on the left a non–Delaunay triangulation is shown: for some triangles, the circumcircle passing on the vertices contains other points from the dataset. In this example we can build a Delaunay triangulation simply by "flipping" the adjacent edge of the two pairs of non–Delaunay triangles, as in the right-most part of Figure 1.

Many solutions have been proposed to compute DT [1]; most of them fall into three broad classes: *on-line* (or *incremental insertion*), *incremental construction*, and *divide & conquer*

On-line [12] methods start with a tetrahedron which contains the pointset, then they insert the points in P one at a time: the tetrahedron which contains the point that is currently being added is partitioned into sub–tetrahedra by inserting it as a new vertex. The empty circumsphere criterion is tested recursively on all the tetrahedra adjacent to the new ones and, if necessary, their faces are flipped. This method in its naive version is extremely simple to program and can be generalized to manage a pointset in E^d space. Moreover, this approach has the lower asymptotic mean time complexity, $\mathcal{O}(n \; log \; n + n^{\lceil \frac{d}{2} \rceil})$ [16].

Incremental construction methods use the empty circle property to construct the DT by successively building tetrahedra whose circumspheres contain no points in P [21]. The efficiency of naive implementation is low ($\mathcal{O}(n^3)$ in the worst case in E^3), but effective acceleration techniques can be devised [7],[14].

Divide & conquer (D&C) algorithms have been proven to be optimal in the E^2 space in both mean and worst–case time complexities [20]. These methods are based on recursive partitioning and local triangulation of the pointset, and then on a merging phase where the resulting triangulations are joined. A new D&C solution working in any space has been proposed in [7].

2.1 Parallel Delaunay Triangulation: previous works

The increasing complexity of input datasets generated by many applications, together with parallel architectures technological trends means that parallel solutions for Delaunay triangulation are increasingly important. To date few papers have been published on the implementation of parallel DT algorithms. In this context, *on-line* algorithms are penalized because their parallel implementation involves considerable overheads due to the high number of interprocess communications and synchronizations needed. Implementations of on–line solutions have been proposed by Saxena et al. [24] and Puppo et al. [23]. In the first paper, an algorithm for DT in E^2 and E^3 was proposed, designed for an orthogonal tree network (i.e. an NxN array of processors where each row and each column forms leaves of a binary tree). This proposal has more theoretical than practical interest, as it only gives computational complexities but does not cover implementation issues. Conversely, Puppo et al. [23] propose a parallel solution designed for data–parallel architectures; their solution has been implemented on a Thinking Machine CM2, and computes triangulations in E^2 space only.

On the other hand, *divide & conquer* methods can be easily parallelized, but suffer from limited scalability. Davy and Dew [9] have proposed a parallel DT for $\mathbf{E}^2$ datasets based on the *D&C* paradigm. The two dimensional space is partitioned into a number of equally sized disjoint stripes and all the sites contained in each stripe are assigned to a node of a multicomputer. Each processor uses an *on–line* algorithm to build the local triangulation. The disjoint triangulations are then merged pairwise, giving rise to a task graph in the shape of a binary tree. The algorithm

was implemented on an eight node Meiko Computing Surface, and the reported speedups range from 4.09 to 5.7. Clematis and Puppo [8] proposed a parallel $D\&C$ triangulator working in E^2 space on a MIMD coarse grain architecture, an nCUBE 2 multicomputer, obtaining similar speedups and using a $D\&C$ solution on the leaf nodes as well.
Cignoni et al. proposed a parallel implementation of a *D&C* triangulation algorithm in $\mathbf{E}^3$ [6]. The sequential algorithm [7] uses an original approach: it subdivides the input dataset, then first builds the part of the DT that would be built in the merge phase of a classic D&C algorithm, and then recursively triangulates the two half–spaces, taking into account the border of the previously computed partial triangulation. The parallel implementation of this algorithm is straightforward, but runs on eight processing nodes of an nCUBE 2 hypercube produces limited speedups.

A more promising approach is *incremental construction*, whose parallelization appears simple and more efficient, although it has the disadvantage of being a static algorithm, i.e. the sites to be triangulated have to be known at algorithm starting time.
A data–parallel implementation of a E^3 incremental construction algorithm was recently proposed by Teng et al. [25]. The algorithm is designed for data–parallel architectures: the search for the site with minimimum circumsphere radius, given an active face, is implemented in parallel by testing more sites at the time on different nodes. A bucketing technique is also used to reduce the number of sites to be tested. The implementation on a Thinking Machine CM-5 produces fast running times and good scalability (the triangulation of a 16K sites dataset requires 266 sec. on a 32 nodes CM-5 and the times reduce to 43 sec. on a 256 nodes CM-5). From a practical point of view, this approach results the more efficient and scalable of those presented in literature.
Some initial results regarding incremental construction algorithm parallelization have been presented by Cignoni et al. [6], reporting on an implementation on a MIMD multicomputer.

Some other papers deal with the parallel computation of both Voronoi diagrams [13] [17] [18] and convex hulls [10] [5]. The duality between DT and Voronoi diagrams is well known [22] and therefore algorithms are given for the construction of the former from the latter, therefore parallel Voronoi algorithms can be used to accelerate Delaunay computations. Moreover, *higher dimensional embedding* algorithms build the Delaunay triangulation by first computing the convex hull of the pointset transformed in E^{d+1} space; this phase can therefore be parallelized by a parallel convex hull implementation. However, the direct parallelization of the Delaunay algorithm is preferable in order to reduce the logical complexity and the overall overheads of the solution.

3 InCoDe, an incremental construction triangulator

A sequential algorithm for DT in E^2 based on the incremental construction approach was originally proposed by McLain [21]. In this section we briefly present **InCoDe** (**In**cremental **Co**nstruction of **De**launay triangulation), which was formerly proposed as a generalized and optimized E^d extension of McLain's algorithm [7]. A similar approach was also applied by Dobkin and Laszlo [11] for E^3 subdivisions, and recently revised and implemented by Fang and Piegl to build triangulations in E^2 space [14].

The algorithm starts from a tetrahedron s in Σ and it incrementally builds Σ by adding a new tetrahedron in each step (Figure 2).

It is well known that for each face f which does not lie in the $ConvexHull(P)$ there are exactly two tetrahedra s_1 and s_2 in Σ sharing such face. In each step, the tetrahedron lying on the face of one of the previously computed tetrahedra is built and added to the current triangulation Σ. All of the faces of each new tetrahedron are used to update an **Active Face List** (**AFL**). Updating the AFL is as follows: if the new face f is already contained in AFL, then f is removed from AFL; otherwise, f is inserted in AFL because, by construction, one of the two tetrahedra sharing f has not yet been built.

The algorithm starts by constructing a first valid Delaunay tetrahedron s_1 over the pointset P. Then, Σ is initialized with this first tetrahedron s_1 and all of its faces are inserted into AFL. The process continues iteratively (extract a face f' from AFL, build the tetra s_i adjacent to f', insert the faces of s_i in AFL, and then again extract another face from AFL) until AFL is empty. The algorithm is specified in pseudo–Pascal in Figure 3.

Two main functions are used: *MakeTetra*, which builds the tetrahedron adjacent to a face f, and *MakeFirstTetra*, which builds the initial tetrahedron.

MakeTetra Function

Given a face f, the adjacent tetrahedron can be built simply using the DT definition. For each point $p \in P$, we compute the radius of the sphere which circumscribes p and the vertices of the face f. We choose the point p which, generally speaking, minimizes this radius to build the tetrahedron adjacent to f.

We limit our analysis of the points $p \in P$ by considering only points lying in the *outer* halfspace with respect to face f (i.e. the halfspace which does not contain the previously generated tetrahedron that contains face f). The outer halfspace associated with f contains any point *iff* face f is part of the convex hull of the pointset P; in this case the algorithm correctly returns no adjacent tetrahedron and, in this case only, $MakeTetra$ returns *null*. The faces on the convex hull are the only faces that belong to just one tetrahedron in Σ.

For each point p in the outer halfspace of f, the radius of the circumsphere is evaluated. $MakeTetra$ selects the point which minimizes the function dd (Delaunay distance):

$$dd(f,p) = \begin{cases} \mathbf{r} & \text{if } \mathrm{c} \subset \mathrm{OuterHalfspace(f)} \\ -\mathbf{r} & \text{otherwise} \end{cases}$$

with r and c the radius and the center of the circumsphere around f and p.

MakeFirstTetra Function

We adopt a simple solution to determine the first tetrahedron, i.e. the MakeFirstTetra function. First, a point $p_1 \in P$ is randomly chosen, then MakeFirstTetra searches for the point $p_2 \in P$ such that the Euclidean distance $d(p_1, p_2)$ is minimal; then, it searches for the point p_3 such that the circumcircle about the 1-face (p_1, p_2) and the point p_3 has minimum radius: the points (p_1, p_2, p_3) are a face of a tetrahedron contained in the Delaunay triangulation. Then the required first tetrahedron is built on the former face.

3.1 Algorithm optimization

The InCoDe algorithm is simple and easy to implement although both its asymptotic worst–case time complexity ($\mathcal{O}(n^3)$) and experimental performances are not optimal. The naive implementation shows an increase in the processing times proportional to the square of the dataset resolution. An analysis of the algorithm reveals two main bottlenecks:

- *MakeTetra* function: given the face f, computation of $dd(f,p)$ is required for each point $p \in P$; the cost of the function is therefore linear to the dataset resolution;
- *active face list* management: if the AFL is implemented as a sequential list, the insertion of a new face f requires an $\mathcal{O}(m)$ search for the presence of f, where m is the number of faces currently stored into the list.

The *MakeTetra* function can be made more efficient by using a bucketing technique, i.e. a regular, non–hierarchical 3D uniform grid (UG), which subdivides the dataset space without overlap or omission, and where the number of UG cells is nearly equal to the number of points in the dataset. In a pre-processing phase, we build for each UG cell the list of points of P contained in the UG cell.
To construct a new tetrahedra in expected constant time the *uniform grid* is used to quickly detect the dd-nearest point by examining the UG cells in order of increasing distance from face f. The analysis of the UG cells can be stopped when there are no more cells contained in the circumsphere around f and the current dd-nearest point.

Moreover, an AFL operations time complexity which is linear to the number of elements in the AFL considerably reduces the efficiency of InCoDe. Thus AFL has been implemented using hash coding; this makes it possible to access each face in the list in expected constant time (1.15 - 1.5 accesses for each search on the average).

An empirically estimated complexity of nearly $\mathcal{O}(nlogn)$ was obtained using these optimization techniques [7]; on 8K site pointset, triangulation times reduced from 9023 to 75 seconds on an IBM R6000/340 workstation (Table 1). Two new measurement units appear in Table 1, as well in the last figure (Figure 11) regarding the parallel implementations. They are the *tetra–per–second* (**tps**), the mean number of tetrahedra produced in a second, and *tetra–per–MFlops* (**tpMFlops**), the former value divided by the performance in MFLOPS achieved on the Linpack benchmark by the actual processor/s used. These units are introduced to make simpler comparisons between implementations on different machines, and they resemble the polygon–per–seconds measurement which is very common in the field of computer graphics.

4 Parallel solutions

Three alternative parallelization strategies are presented in this section. The first two approaches have been implemented and evaluated on two quite different environments: a distributed memory MIMD architecture, which uses message–passing for exploiting interprocess communication, and a local network of workstations cooperating under the Linda programming environment.
The aim is therefore twofold: to show the effectiveness and shortcomings of alternative parallel solutions and to evaluate two different parallel environments on the same application.
The third solution, aimed at overcoming the low scalability common to the other two implementations, is specifically designed for highly parallel environments. It exploits the parallelism of the first two approaches through the clustering concept: the parallelism among the clusters follows the spatial decomposition approach (as in the first solution), while each cluster computes the triangulation of the assigned partition by exploiting master–slaves parallelism.
Below we briefly introduce the characteristics of the parallel environments used, and then we discuss the parallel solutions and the results achieved.

The nCUBE 2 system, a distributed memory MIMD architecture

The multicomputer used in this experiment is an nCUBE 2 hypercube model 6410, with 4 Mbytes of local memory per node. Running at a clock rate of 20 MHz, the nCUBE processor is rated at 2.4 MFLOPS peak performance in double precision (0.7 MFLOPS on the Linpack benchmark). The vendor message–passing libraries and programming environment have been used for programming the nCUBE parallel versions of the triangulator.

Distributed computing with Linda

The other experimented parallel environment is composed of eight IBM R6000/340 workstations (15 Linpack MFLOPS) interconnected by Ethernet and viewed as a single loosely–coupled distributed memory MIMD computer through the Network Linda programming environment[1].

[1] To avoid overheads due to other unrelated processes and collisions in the network, the Linda tests were run during the night when the workstations were unloaded and traffic on the local network was very low.

Linda model [3] is based on a virtually-shared associative memory called *tuple space* (TS). A *tuple* is a collection of typed fields, each holding a value of any type available in the sequential language. The processes of a Linda parallel program cooperate by putting tuples into and getting them from the *tuple space* via a few simple atomic operations:

- **eval(**t**)**: a new process is forked, which evaluates each field of tuple t; after evaluation, t becomes a passive tuple that can be read or consumed by another process.
- **out(**t**)**: the calling process evaluates each field of tuple t and adds the new tuple to TS.
- **in(**s**)**: a tuple t that matches the template s is removed from TS; the values of the actuals in t are assigned to the corresponding formal fields of s (variables prefixed by the symbol "?"). A template s matches a tuple t *iff*: (1) s and t have the same number of fields, and (2) each field in s has the same type as the corresponding field in t, and (3) actuals of s have the same values of corresponding actuals of t. If a tuple matching template s is not available in TS, the executing process is suspended until the next one is added by another process.
- **rd(**s**)**: has the same semantics as **in**, however the matching tuple t is not removed from TS.

In addition to these four basic operations, non-blocking predicate versions of **in** and **rd** are provided in the Linda implementation, named **inp** and **rdp**, respectively.

4.1 A spatial decomposition approach

The incremental Delaunay triangulator can be parallelized on m independent asynchronous processors by means of **pointset space subdivision**. We can easily partition the bounding box of pointset P into m rectangular regions, called $rr_i, 1 \leq i \leq m$. The same InCoDe process is replicated on each pe, together with the full pointset P. A rectangular region rr_i is assigned to each InCoDe process, which will compute all of the tetrahedra in Σ that are at least partially contained in rr_i. Although performing the triangulation of a region basically requires only local information, particular cases may exist in which a pe needs the full pointset in order to build all the tethraedra at least partially contained in the assigned region. Therefore the input

dataset cannot in general be partitioned among the *pe*'s to reduce memory requirements. On multicomputers characterized by local memories of limited sizes, large dataset can be eventually managed by adopting a pagination mechanism based on a simple LRU substitution policy.
The pseudo–code specification of the parallel InCoDe process is in Figure 5. The solution is algorithmically quite simple: the main difference from the sequential version is the management of the AFL list. For each new tetrahedron, we only insert into AFL the faces completely/partially contained in the local region rr_i. Each *pe* computes: (a) the tetrahedra within rr_i and (b) those shared with adjacent regions (in Figure 4 a 2D example is shown).
Storing replicated tetrahedra (i.e. the tetrahedra that cross region boundaries are computed once for each region intersected) in the output dataset can be simply prevented with a vertex–containment test. We include each new tetrahedron in the local output list *iff* its upper–left–frontmost vertex (*ULFVertex*) is contained in the local rr_i region.

The nCUBE and Network Linda versions essentially differ only in the initialization phase: in the nCUBE implementation each node synchronously reads the whole pointset P from secondary storage. In the Network Linda version the master workstation puts P into the tuple space and then it is read by all the InCoDe processes spawned in the network.
In both implementations the reported times also include the I/O times spent in reading and broadcasting the full pointset.

The times in Table 2 were obtained by running the nCUBE parallel version of the triangulator on two datasets generated randomly according to a uniform distribution; t_{min} and t_{max} are the times of the fastest and slowest process. Speedups and efficiencies are obviously computed using t_{max}. Column labeled $t_{I/O}$ reports I/O times. We chose to indicate separately I/O costs because they depend on the configuration of the I/O subsystem used (in our case a single disk driven by a single I/O node). A richer I/O configuration would allow to improve obtained efficiencies by shortening these times. $|\Sigma|$ is the number of tetrahedra in the resulting triangulation, while *dupl.factor* measures the number of tetrahedra built by more than one processing node (those on the borders of the regions rr_i) and can be used to quantify the overheads due to the parallelization strategy.
Table 3 shows the times of the Network Linda implementation which were run on the same input datasets on the workstation network. The table does not report the duplication factor because

it is clearly equal to that of the nCUBE version.

The times obtained with both the implementations are quite satisfactory although the low scalability jeopardizes the effective exploitation of highly parallel multicomputers. As can be seen from Table 2, load unbalance and the duplicated computation of some tetrahedra are the main sources of inefficiency in this solution. In these experiments a simple regular and uniform partition was applied to divide the load between *pe*'s.

An **adaptive partitioning** strategy, based on the definition of rr_i regions with an equidistribution of the points in P, has also been tested without notable benefits on datasets nearly uniformly distributed. In fact the computational costs of triangulating each rr_i region does not strictly depend on the number of points contained inside rr_i because of the high variability of the times required to build tetrahedra from points showing different geometrical disposition. Let consider, for example, the relative cost of triangulating points close to the dataset convex hull or having a large number of neighbor sites at similar euclidean distance. It may therefore happen that the *pe* which builds the largest number of tetrahedra is also one of the faster to end execution.

On the other end, for datasets characterized by an unequal distribution of the points (see datasets *Bubbles* and *Bluntfin*[2] in Figure 6), the adoption of an adaptive strategy improves the load balance and results in sensibly lower execution times. The adaptive partitioning strategy (see Figure 7) is based on a simple orthogonal recursive partitioning algorithm which allows a good balanced distribution of the points in the rr_i regions. It applies a succession of three sorts of the points, one for each xyz axes; the dataset bounding volume is then recursively partitioned into rectangular subregions.

Table 4 compares the nCUBE times obtained by running the *adaptive* and *uniform* versions of the triangulator on geometrically different datasets. As expected, the adaptive version results in better performances on the *Bubbles* and *Bluntfin* datasets, while the equidistribution cost overwhelms the slight load balance benefits for nearly uniform datasets.

The adoption of dynamic load balancing strategies is under study, but the first results are

[2] *Bluntfin* dataset is the result of a fluid dynamics simulation on a curvilinear grid, produced and distributed by NASA-Ames Research Center.

not very comforting due to the large overheads involved. For example, the simple solution of making a finer partition of the bounding box and dynamically dispatching the rr_i regions on the *pe*'s on the basis of their loads has more shortcomings than advantages: the load is more balanced but the number of duplicated tetrahedra increases dramatically, resulting in longer execution times.

Another possible dynamic load balancing strategy, a dynamic shrinking/widening of the rr_i regions depending on the current load, is unlikely to lead to effective improvements due to the high costs of coordination/communication required. What is simple for other algorithms (e.g. balancing the load of a parallel ray tracer [2]) is not straightforward in a Delaunay triangulator, due to the unpredictable geometrical state of the computations in a given instant (e.g. all of the tetrahedra in the shrinking region may have already been computed, because it is not simple to maintain knowledge of how much and what part of each rr_i region has been triangulated in a given instant). Moreover, well known problems of dynamic load balancing techniques, such as how to choose the region to shrink or the periodicity of such load redistribution, discourage using a dynamic approach.

4.2 A master–slaves approach

The InCoDe algorithm can be also parallelized following the *master–slaves* paradigm: a master process (Face_Mgr - Face Manager) manages the centralized list of active faces and dispatches the faces contained in it over a set of independent worker processes (Tetra_Bld - Tetra Builder). Each process is mapped on a different processing node. The Tetra_Bld processes, on which the full input pointset P is replicated, repeatedly wait for the reception from the Face_Mgr process of a list of faces, call the *MakeTetra* function on each face, and return to Face_Mgr the resulting set of tetrahedra. The program ends when the AFL is empty, and all Tetra_Bld processes have terminated and sent their results to Face_Mgr process.

On the reception of a result from a Tetra_Bld process, the Face_Mgr process firstly sends it a new list of faces and then inserts the faces of the received tetrahedra into the AFL. One difference between the sequential and the master–slaves parallel version of InCoDe is the need for a replication test over the output data. In sequential InCoDe algorithm, tetrahedra can only be built once due to the strict ordering between constructing of each tetrahedron and AFL updates. Conversely, in the parallel version different faces, sent to different Tetra_Bld processes, may originate the same tetrahedron. This occurs even if the program is being run with a single Tetra_Bld process because, to decrease communication overheads, the Face_Mgr sends list of m faces to the worker nodes instead of a single face at a time, and initially puts two different lists of faces into the message queue of each Tetra_Bld process.

The optimal value of m (i.e. the number of faces sent in every message to the Tetra_Bld processes) has been experimentally fixed for the nCUBE 2 system as being equal to the number of processing elements used. Although this scheduling policy decreases both the number of exchanged messages and the times the Tetra_Bld processes spend waiting for the reception of the next set of faces, unfortunately it increases the probability that the same tetrahedra are computed several times and means that the Face_Mgr has to check if each tetrahedron received has been already built and inserted into the current output.

The Linda pseudocodes describing the Face_Mgr and Tetra_Bld algorithms implementing the same master–slaves paradigm are reported in Figures 8 and 9, respectively.

The strategy adopted in the Linda version to dispatch the work over the chosen number ($num\_workers$) of Tetra_Bld processes gives priority to the management of the AFL list. As soon as a Tetra_Bld process puts a tuple containing computed tetrahedra into TS, the Face_Mgr process reads and manages it, i.e. each tetrahedron not already present in the current output list is added to Σ and its faces are used to update the AFL. More jobs (lists of m faces, with m equal to the current value of the $faces\_per\_job$ parameter) are put into the tuple space when the current number of jobs (given by the value of $JobsNum$) is lower than a threshold ($num\_jobs * num\_workers$). This strategy is aimed at allowing a certain number of jobs is always present in TS, thus avoiding workers being blocked. The fine tuning of the two parameters num_jobs and $faces\_per\_job$ is essential to the performance of the Linda triangulator. Increasing the $faces\_per\_job$ (i.e. the number of faces dispatched at a time) or the num_jobs (i.e. the average number of jobs present in TS for each Tetra_Bld process) values, results in fewer communications and synchronizations among the parallel processes but also in an increase in the number of replicated tetrahedra (i.e. those which are computed more than once).

These parameters have been empirically fixed for the network environment used at faces_per_jobs

= 30 and nums_jobs = 3, which are clearly much larger then the corresponding values used in the multicomputer implementation.

Tables 5 and 6 show the results obtained by implementing the described parallel versions of the triangulation algorithm on the nCUBE 2 and on the network of workstations, respectively. Analogously to the previous Tables, $|\Sigma|$ is the number of tetrahedra in the resulting triangulation, while *dupl.factor* is the ratio between the number of tetrahedra actually built and $|\Sigma|$. In both Tables the reported times also include the I/O times spent in reading and broadcasting the full pointset.

Some comments can be made to explain the poor speedups of this parallel solution.
Firstly, by comparing the times of the sequential algorithm (reported in the rows labeled *seq.* of Table 6) with the times of the Linda parallel version ran on two workstations, one acting as Face_Mgr and one as Tetra_Bld, it is clear that due to the large amount of messages managed by the Face_Mgr process, communication overheads are very high, thus precluding the effectiveness of the solution especially on the workstation network environment. This is true also for the nCUBE version, although on two nodes of this architecture the communication overheads are lower than the gains obtained by overlapping the AFL management with the building of the tetrahedra (i.e. the nCUBE times on two nodes are slightly shorter than the sequential ones). Moreover, the scheduling policy adopted in both the implementations to decrease the number of messages exchanged by sending several faces at a time, results in a large number of tetrahedra being replicated.
Secondly, the execution profile of the sequential algorithm shows that, independently of the pointset cardinality, the management of the AFL constitutes about 10 percent of the total computational cost. If we also consider the cost of the replication test over the output list, this percentage significantly increases. According to Amdahl's Law, this high sequential cost gives an optimistic upper bound of ten to the speedup of any parallel algorithm which performs the AFL management in sequential.

On the other end, the *master-slaves* solution gives a better distribution of the load among the *pe*'s. As a result, its performances are nearly irrespective of the pointset distribution (we observed similar behavior on *Uniform*, *Bubbles* and *Bluntfin* datasets).

4.3 Towards a more scalable parallel triangulator

The main shortcoming common to the parallel solutions depicted in the previous sections is low scalability. The scalability of the first solution is constrained by load unbalance and by the increase in duplicated computations coming from the finer partitions used. With the second solution, on the other hand, the satisfactory exploitation of more than a few processors is precluded by many factors such as communication overheads, duplicated computations and sequential costs associated with the centralized AFL management. As can be seen from Table 2, the efficiency of the spatial decomposition solution implemented on the multicomputer falls below the 50% when the 16 *pe*'s threshold is overcome. A comparable degradation occurs in the version implemented according to the master–slaves paradigm when more than 4 *pe*'s are exploited (see Table 5). Furthermore, the same tables show that, irrespective of the dataset cardinality, for a number N of *pe*'s greater than 16 the following relation holds:

$$\frac{Speedup_{spatial-dec.}(N)}{Speedup_{spatial-dec.}(N/4)} < Speedup_{master-slaves}(4) \tag{1}$$

i.e. the gain in terms of run times achieved by further partitioning each rectangular region rr_i into four subregions and, consequently, by using four times as many *pe*'s (e.g. 64 *pe*'s instead of 16) is lower than the speedup over 4 *pe*'s working with the master–slaves parallel solution. This suggests the adoption of a *hybrid* strategy that groups the processing nodes of the multicomputer into *clusters* of four *pe*'s each: a rectangular region rr_i of the partitioned triangulation space is assigned to each cluster, which will compute its subset of tetrahedra in parallel by executing the master–slaves algorithm (Figure 10).

This hybrid solution could be implemented quite easily and may allow us to raise the mentioned *pe*'s threshold over which the efficiencies of the previous parallel solutions become unacceptable.

From the execution profilings of the parallel implementations presented in Sections 4.1 and 4.2 it is possible to predict analytically, with a sufficient degree of accuracy, the expected behavior of the proposed hybrid solution. The estimate was made according to the following formulas:

$$Speedup_{hybrid}(20k, N) \approx Speedup_{spatial-dec.}(20k, N/4) * \frac{T_{seq}(20k) - T_{I/O} - T_{UG}}{T_{master-slaves}(20k, 4) - T_{I/O} - T_{UG}} \tag{2}$$

where $T_{I/O}$ and T_{UG} are the times necessary to perform I/O operations and to build the Uniform Grid over the pointset, respectively. These initializations are common to both space partitioning and master–slaves implementations and their cost have to be therefore charged only once.
Eq. 2 can be rewritten as:

$$\frac{T_{seq}(20k)}{T_{hybrid}(20k, N)} \approx \frac{T_{seq}(20k)}{T_{spatial-dec.}(20k, N/4)} * \frac{T_{seq}(20k) - T_{I/O} - T_{UG}}{T_{master-slaves}(20k, 4) - T_{I/O} - T_{UG}} \tag{3}$$

From Eq. 3 we can derive the following equation, used to estimate the times of the hybrid solution reported in Table 7:

$$T_{hybrid}(20k, N) \approx \frac{T_{spatial-dec.}(20k, N/4) * (T_{master-slaves}(20k, 4) - T_{I/O} - T_{UG})}{T_{seq}(20k) - T_{I/O} - T_{UG}} \tag{4}$$

As results form the analysis of Table 7 and of the graphs in Figure 11, the hybrid solution obtained by combining the two parallelization strategies gives a more effective exploitation of the processing power available on highly parallel architectures. The increase in performance is more than 20% and 45%, respectively on 64 and 128 *pe*'s. It is worth mentioning that such efficiency, estimated in 0.30 on 128 *pe*'s, could be considered not exciting in many scientific applications, but, in the particular case of not trivial geometrical parallel processing, these results are more than satisfactory. A certain degree of load umbalance is in fact structural with the problem addressed, depending on the characteristics of the geometrical disposition of the sites which is unknown until the final solution of the problem is reached; this largely prevents the design of cost–effective load balancing strategies.

5 Concluding remarks

Two different parallel solutions have been presented, which solve the general problem of computing a tessellation of the E^3 space into tetrahedral cells.
Two classical computational paradigms have been investigated for parallelizing a static incremental triangulation algorithm: *spatial decomposition* and *master–slaves*. In both cases, reducing communications and synchronizations has been one of the main goals of this project, in order to design solutions which could be ported onto different architectures at a low rewriting cost, while maintaining sufficiently high efficiency.
The first solution is based on the spatial decomposition of the pointset bounding box, i.e. by

dividing it into rectangular regions, and on independently computing the partition on each region. This solution does not require communications or synchronization among *pe*'s, due to some simple modifications in the sequential code.

The algorithm has been implemented on two different parallel architectures, a hypercube multicomputer and a network of workstations cooperating via Linda. The efficiency of these solutions is bounded by the cost of the initialization phase (e.g. input of the data, construction of the bucketing data structure needed to optimize geometrical computations) and by the cost of constructing the replicated tetrahedra, i.e. those which are on the borders of the regions. The efficiency of the multicomputer implementation remains higher than 0.5 when no more than 16 *pe*'s are used. Linda implementation shows an efficiency similar to that obtained on the multicomputer thus indicating that *Network Computing* is viable and effective especially if the application communication requirements are not so demanding.

The second parallel solution, based on the master–slaves approach, was less efficient than the first one and only reaches a very limited scalability; the best results were obtained while running on 4 *pe*'s. Performances in the second solution are mainly limited by communication overheads (faces sent to the workers and tetrahedra returned to the master), and these overheads are higher for Linda implementation, because in our configuration *pe* communications take place using standard Ethernet. The ratio between the communication bandwidth and computational power of the network of workstations used is in fact about two orders of magnitude lower than that of a multicomputer like nCUBE 2. The difference between nCUBE and Network Linda efficiencies is lower than expected. This shows that the overheads associated with the management of the shared tuple space on a distributed memory environment are limited due to the optimized implementation of Network Linda [4]. As a general consideration, we found the Linda environment easier and more practical than a message–passing one due to the greater expressive power and flexibility of its programming model.

Architectures with higher degree of parallelism could be exploited better by our hybrid solution, which is based on the spatial decomposition of the pointset bounding box and on the use of a master–slaves approach to solve each local sub–problem. The performances of this solution have only been estimated, on the basis of the processing times of the two previous implementations. This evaluation shows that effective improvements are obtained, in the case of a multicomputer environment, when 32 or more *pe*'s are used.

Acknowledgements

The authors would like to thank Nicholas Carriero who kindly spent some of his precious time in running our Linda codes at Yale, Nicola de Candussio and Alfredo Villani for the implementation of part of the parallel codes.
The work described in this paper has been partially funded by the Progetto Finalizzato "Sistemi Informatici e Calcolo Parallelo" of the Italian National Research Council.

References

[1] F. Aurenhammer. Voronoi diagrams - A survey of a fundamental geometric data structure. *ACM Computing Survey*, 23(3):345–405, September 1991.

[2] D. Badouel, K. Bouatouch, and T. Priol. Ray tracing on distributed memory parallel computers: strategies for distributing computations and data. In *Parallel Algorithms and Architectures for 3D Image Generation - ACM SIGGRAPH '90 Course Note No.28*, pages 185–198, July 1990.

[3] N. Carriero and D. Gelenter. How to write a parallel program: a guide to the Perplexed. *ACM Computing Surveys*, 21(3):323–358, September 1989.

[4] N. Carriero and D. Gelenter. Tuple analysis and partial evaluation strategies in the linda precompiler. In D. Gelenter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 114–126. MIT Press, 1990.

[5] N. Chandrasekhar and W.R. Franklin. A fast practical parallel convex hull algorithm. Technical report, Eletrical, Computer and Systems Engineering Department, Rensselaer Polytechnic Institute, Troy, NY, Nov. 1989.

[6] P. Cignoni, C. Montani, R. Perego, and R. Scopigno. Parallel 3D Delaunay triangulation. *Computer Graphics Forum*, 12(3):129–142, 1993.

[7] P. Cignoni, C. Montani, and R. Scopigno. A merge-first divide & conquer algorithm for E^d triangulations. Technical Report 92-16, Istituto CNUCE – C.N.R., Pisa, Italy (submitted paper), October 1992.

[8] A. Clematis and E. Puppo. Effective parallel processing of irregular geometric structure: an experience with the Delaunay triangulation. In *Proceedings of A.I.C.A. '93 International Section - Parallel and Distributed Architectures and Algorithms (Lecce, ITALY)*, pages 235–251, September 22-24 1993.

[9] J.R. Davy and P.M. Dew. A note on improving the performance of Delaunay triangulation. In *Proc. of Computer Graphics International '89*, pages 209–226, 1989.

[10] A.M. Day. Parallel implementation of 3D convex hull algorithm. *Computer–Aided Design*, 23(3):177–188, Apr. 1991.

[11] D.P. Dobkin and M.J. Laszlo. Primitives for the manipulation of three–dimensional subdivisions. *Algorithmica*, 4:3–32, 1989.

[12] H. Edelsbrunner and N. R. Shah. Incremental topological flipping works for regular triangulations. In *Proceedings of the 8th Annual ACM Symposium on Computational Geometry*, pages 43–52, June 1992.

[13] D.J. Evans and I. Stojmenovic. On parallel computation of Voronoi diagrams. *Parallel Computing*, (12):121–125, 1989.

[14] T.P. Fang and L.A. Piegl. Delaunay triangulation using a Uniform Grid. *IEEE Computer Graphics & Applications*, 13(3):36–47, May 1993.

[15] L. De Floriani. Surface representations based on triangular grids. *The Visual Computer*, 3:27–50, 1987.

[16] L.J. Guibas, D. E. Knuth, and M. Sharir. Randomized incremental construction of Delaunay and Voronoy diagrams. In *Lect. Note Comp. Science 443*, pages 414–431. Springer-Verlag, 1990.

[17] C.S. Jeong. An improved parallel algorithm for constructing Voronoi diagram on a mesh–connected computer. *Parallel Computing*, (17):505–514, 1991.

[18] C.S. Jeong. Parallel Voronoi diagram in L_1 (L_∞) on a mesh–connected computer. *Parallel Computing*, (17):241–252, 1991.

[19] A. Kaufman. *Volume Visualization.* IEEE Computer Society Press, Los Alamitos, CA, 1990.

[20] D.T Lee and B.J. Schachter. Two algorithms for constructing a Delaunay triangulation. *Int. J. of Computer and Information Science*, 9(3):219–242, 1980.

[21] D.H. McLain. Two dimensional interpolation from random data. *The Computer J.*, 19(2):178–181, 1976.

[22] F.P. Preparata and M.I. Shamos. *Computational Geometry - An Introduction.* Springer-Verlag, 1985.

[23] E. Puppo, L. Davis, D. De Menthon, and Y.A. Teng. Parallel terrain triangulation. In *Proceedings of 5th International Symposium on Spatial Data Handling (to appear also on the Int. J. of Geographical Information System)*, pages 632–641, August 3-7 1992.

[24] S.Saxena, P.C. Bhatt, and V.C. Prasad. Efficient VLSI parallel algorithm for Delaunay triangulation on orthogonal tree network in two and three dimensions. *IEEE Transactions on Computers*, 39(3):400–404, 1990.

[25] Y.A. Teng, F. Sullivan, I. Beichl, and E. Puppo. A data–parallel algorithm for three–dimensional Delaunay triangolation and its implementation. In *Proceedings of Supercomputing '93*, pages 112–121, November 15-19 1993.

[26] P.L. Williams. Interactive splatting of nonrectilinear volumes. In A.E. Kaufman and G.M. Nielson, editors, *Visualization '92 Proceedings*, pages 37–45. IEEE Computer Society Press, 1992.

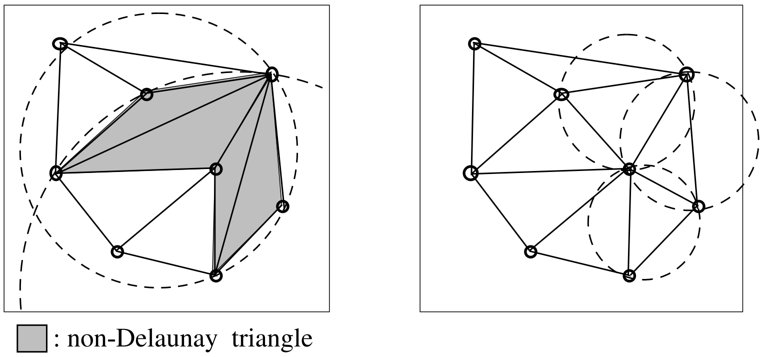


Figure 1: Triangulations over a simple pointset in E^2; on the right, the Delaunay triangulation.

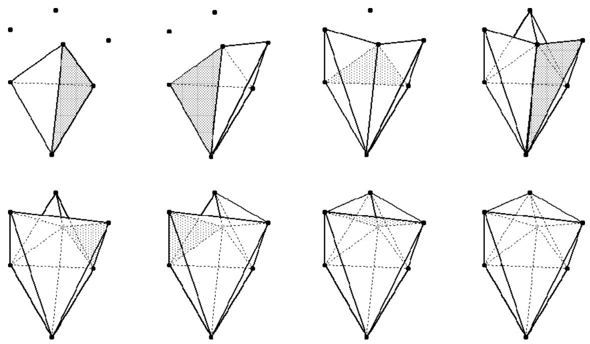


Figure 2: Incremental construction of a Delaunay triangulation in E^3.

```
Function InCoDe (P:pointset):tetrahedra_list;
    var f:face; AFL:face_list;
        t:tetrahedra; Σ:tetrahedra_list;
    begin
        AFL:=emptylist;
        t:=MakeFirstTetra(P);
        Insert(t,Σ);
        AFL:=faces(t);
        while notempty(AFL) do
            f:=Extract(AFL);
            t:=MakeTetra(f, P);
            if t ≠ null then
                Insert(t,Σ);
                for each f′:  f′ ∈faces(t) AND f′ ≠ f do
                    Update(f',AFL)
        InCoDe:=Σ;
    end;
```

Figure 3: InCoDE algorithm.

InCoDE (sites no.)		2000	4000	6000	8000	10000	20000
InCoDE	*seconds*	428	1838	4507	9023	-	-
Opt. InCoDE	*seconds*	14	32	51	75	96	223
Opt. InCoDE	*tps*	926	820	776	706	692	598
Opt. InCoDE	*tpMFlops*	61.7	54.6	51.7	47.0	46.1	39.8

Table 1: Performances of the sequential version of InCoDe running on a IBM R6000/340 workstation (Opt.InCoDe: optimized algorithm).

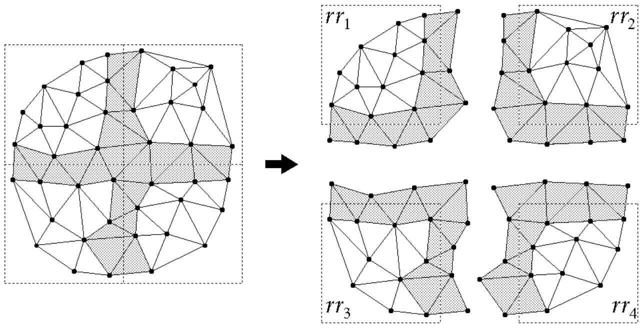


Figure 4: Partition of the triangulation space into four regions; the duplicated triangles are in gray.

```
Function ParInCoDe (P:pointset, rr_i:rect_reg):tetrahedra_list;
    var f:face; AFL:face_list;
        t:tetrahedra; Σ:tetrahedra_list;
    begin
        AFL:=emptylist;
        t:=MakeFirstTetra(P,rr_i); {the first tetrahedron must be in rr_i}
        Insert(t,Σ);
        AFL:=faces(t);
        while notempty(AFL) do
            f:=Extract(AFL);
            t:=MakeTetra(f, P);
            if t ≠ null then
                if ULFVertex(t) ∈ rr_i then Insert(t,Σ);
                for each  f':  f' ∈faces(t) AND f' ≠ f AND f' ∩ rr_i ≠ ∅ do
                    Update(f',AFL)
        ParInCoDe:=Σ;
    end;
```

Figure 5: Parallel version of InCoDE based on spatial decomposition.

nCUBE times	t_{min}	t_{max}	t_{mean}	$t_{I/O}$	*speedup*	$eff.$	$\|\Sigma\|$	$dupl.factor$
10000 sites								
1 *pe*	684.88			0.50	1	1	66445	1
2 *pe*'s	374.97	391.19	383.09	0.71	1.75	0.88		1.05
4 *pe*'s	190.07	226.23	207.80	1.39	3.03	0.76		1.11
8 *pe*'s	105.30	134.28	115.47	2.01	5.10	0.64		1.17
16 *pe*'s	58.36	85.08	68.45	2.69	8.05	0.50		1.30
32 *pe*'s	33.60	53.50	41.35	3.42	12.80	0.40		1.44
64 *pe*'s	19.19	35.56	25.09	4.11	19.26	0.30		1.58
128 *pe*'s	12.59	27.49	17.38	4.92	24.91	0.19		1.87
20000 sites								
1 *pe*	1670.25			0.88	1	1	133450	1
2 *pe*'s	913.13	929.69	921.41	1.38	1.80	0.90		1.04
4 *pe*'s	509.21	534.12	518.23	2.57	3.13	0.78		1.09
8 *pe*'s	252.48	312.52	276.58	3.48	5.34	0.67		1.14
16 *pe*'s	140.83	187.68	162.36	4.25	8.90	0.56		1.23
32 *pe*'s	77.03	121.60	97.17	5.17	13.73	0.43		1.34
64 *pe*'s	39.37	87.55	58.35	6.07	19.08	0.30		1.44
128 *pe*'s	26.26	80.64	39.24	7.04	20.71	0.16		1.66

Table 2: Parallel triangulation of *Uniform* datasets via spatial decomposition: nCUBE run times (in seconds), speedups and efficiencies.

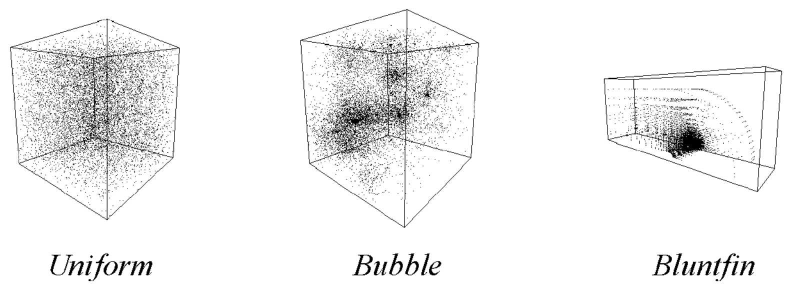
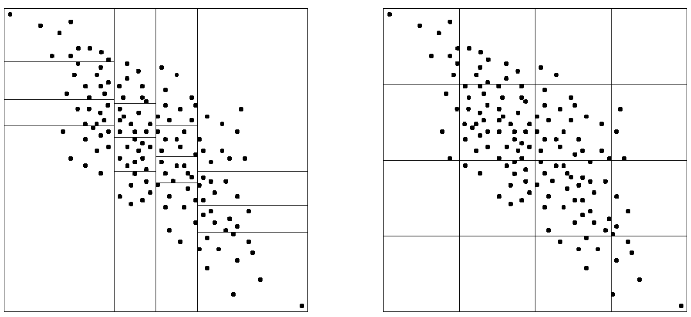


Figure 6: Visualization of the *Uniform*, *Bubbles* and *Bluntfin* datasets used in the experiments.

Linda times	*time*	*speedup*	*eff.*
10000 sites			
1 ws	96.35	1	1
2 ws	55.01	1.75	0.87
4 ws	32.40	2.97	0.74
8 ws	20.27	4.75	0.59
20000 sites			
1 ws	223.27	1	1
2 ws	123.82	1.80	0.90
4 ws	72.26	3.08	0.77
8 ws	43.54	5.12	0.64

Table 3: Parallel triangulation of *Uniform* datasets via spatial decomposition: Network Linda run times (in seconds), speedups and efficiencies.



a *b*

Figure 7: 2D example showing the region boundaries resulting from the adoption of the *adaptive* (a) and *uniform* (b) partitioning strategies.

nCUBE times	1 *pe*	2 *pe*'s	4 *pe*'s	8 *pe*'s	16 *pe*'s	32 *pe*'s	64 *pe*'s	128 *pe*'s
Uniform dataset (10k points)								
adaptive version	790.5	455.3	256.6	151.0	97.1	65.2	44.2	35.8
uniform version	790.5	452.1	261.7	155.5	98.7	47.1	41.3	31.9
Bubbles dataset (20k points)								
adaptive version	1972.3	1069.3	597.6	342.1	245.1	144.2	104.8	94.5
uniform version	1972.3	1537.1	868.9	498.0	330.5	240.2	173.8	124.0
Bluntfin dataset (10k points)								
adaptive version	4138.9	2856.2	1581.2	961.8	1145.6	756.8	498.2	371.7
uniform version	4138.9	3935.5	3883.3	3179.0	3158.8	3118.0	2480.0	2114.6

Table 4: Parallel triangulation via spatial decomposition of the datasets shown in Figure 6: comparison between the nCUBE run times (in seconds) of the uniform and adaptive versions of the triangulator.

nCUBE 2 times	*time*	*speedup*	*eff.*	$\|\Sigma\|$	*dupl.factor*
10000 sites				66445	
seq.	684.88	1	1		1
1 Face_Mgr, 1 Tetra_Bld	613.25	1.12	0.56		1.07
1 Face_Mgr, 3 Tetra_Bld	250.94	2.73	0.68		1.16
1 Face_Mgr, 7 Tetra_Bld	32.00	2.95	0.37		1.25
20000 sites				133450	
seq.	1670.25	1	1		1
1 Face_Mgr, 1 Tetra_Bld	1565.57	1.07	0.53		1.07
1 Face_Mgr, 3 Tetra_Bld	599.28	2.79	0.70		1.15
1 Face_Mgr, 7 Tetra_Bld	492.23	3.39	0.42		1.26

Table 5: Parallel triangulation of *Uniform* datasets under the master–slaves paradigm: nCUBE run times (in seconds), speedups and efficiencies.

```
Function Face_Mgr (P:pointset,num_workers,num_jobs,faces_per_job):tetrahedra_list;
    var f:face; AFL,Job:face_list;
        t:tetrahedra; Σ,result:tetrahedra_list;
    begin
        JobsNum:=0; AFL:=emptylist;
        t:=MakeFirstTetra(P);
        Insert(t,Σ);
        AFL:=faces(t);
        for i:=1 to num_workers do /* Tetra_Bld processes spawning*/
            eval("Worker", Tetra_Bld(P));
        while JobsNum≠0 OR NotEmpty(AFL) do
            while inp(?result) do /* get tetra from TS */
                JobsNum:=JobsNum-1;
                for each t :  t ∈ result do
                    if not Member(t, Σ) /* check for duplication */
                        then
                            Insert(t, Σ);
                            for each f:  f ∈faces(t) and f ≠starting_face(t) do
                                Update(f, AFL);
            while JobsNum< num_jobs*num_workers && NotEmpty(AFL) do
                Job:=M_Extract(AFL, faces_per_job);
                /* extract a number of "faces_per_job" faces from AFL */
                out(Job);
                JobsNum:=JobsNum+1;
        for i:=1 to num_workers do /* stop Tetra_Bld processes*/
            out(EndMessage);
        Face_Mgr:= Σ;
    end;
```

Figure 8: Pseudocode of the Network Linda implementation of Face_Mgr (Face Manager) process.

```
Function Tetra_Bld (P:pointset);
    var f:face; Job:face_list;
        t:tetrahedra; result:tetrahedra_list;
    begin
        in(?Job);
        while Job ≠ EndMessage do
            result:= ∅;
            for each f :  f ∈ Job do
                t:=MakeTetra(f, P);
                Insert(t, Result);
            out(Result);
            in(?Job);
    end;
```

Figure 9: Pseudocode of the Network Linda implementation of Linda Tetra_Bld (Tetrahedra Builder) process.

Linda times	*time*	*speedup*	*eff.*	$\lvert\Sigma\rvert$	*dupl.factor*
10000 sites				66445	
seq.	96.35	1	1		1
1 Face_Mgr, 1 Tetra_Bld	145.39	0.66	0.33		1.38
1 Face_Mgr, 2 Tetra_Bld	81.17	1.19	0.39		1.44
1 Face_Mgr, 3 Tetra_Bld	59.60	1.61	0.40		1.51
1 Face_Mgr, 4 Tetra_Bld	51.79	1.86	0.37		1.53
1 Face_Mgr, 5 Tetra_Bld	47.83	2.01	0.33		1.52
20000 sites				133450	
seq.	223.49	1	1		1
1 Face_Mgr, 1 Tetra_Bld	355.49	0.63	0.31		1.32
1 Face_Mgr, 2 Tetra_Bld	192.71	1.16	0.39		1.43
1 Face_Mgr, 3 Tetra_Bld	137.57	1.62	0.41		1.48
1 Face_Mgr, 4 Tetra_Bld	112.24	1.99	0.40		1.51
1 Face_Mgr, 5 Tetra_Bld	102.56	2.18	0.36		1.50

Table 6: Parallel triangulation of *Uniform* datasets under the master–slaves paradigm: Network Linda run times (in seconds), speedups and efficiencies.

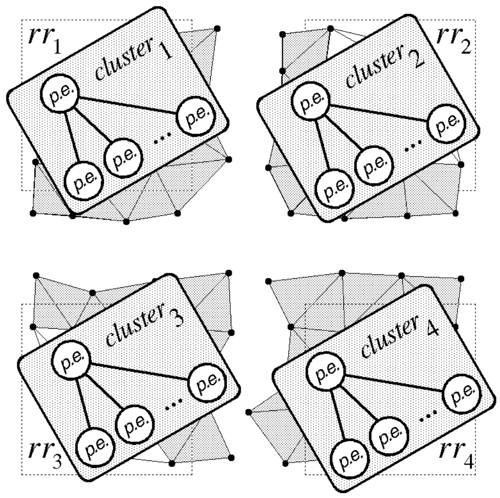


Figure 10: Increasing the scalability by adopting a hybrid strategy (spatial decomposition and master–slaves parallelism).

nCUBE estimated times	*time*	*speedup*	*eff.*
20000 sites			
1 *pe*	1670.25	1	1
8 *pe*'s	330.99	5.05	0.63
16 *pe*'s	190.16	8.78	0.55
32 *pe*'s	111.26	15.01	0.47
64 *pe*'s	66.82	25.00	0.39
128 *pe*'s	43.29	38.58	0.30
256 *pe*'s	31.17	53.58	0.21

Table 7: nCUBE 2 estimated run times (in seconds), speedups and efficiencies of the hybrid version of the parallel triangulator (four *pe*'s for each cluster, spatial decomposition among clusters and master–slaves parallelism inside each cluster).

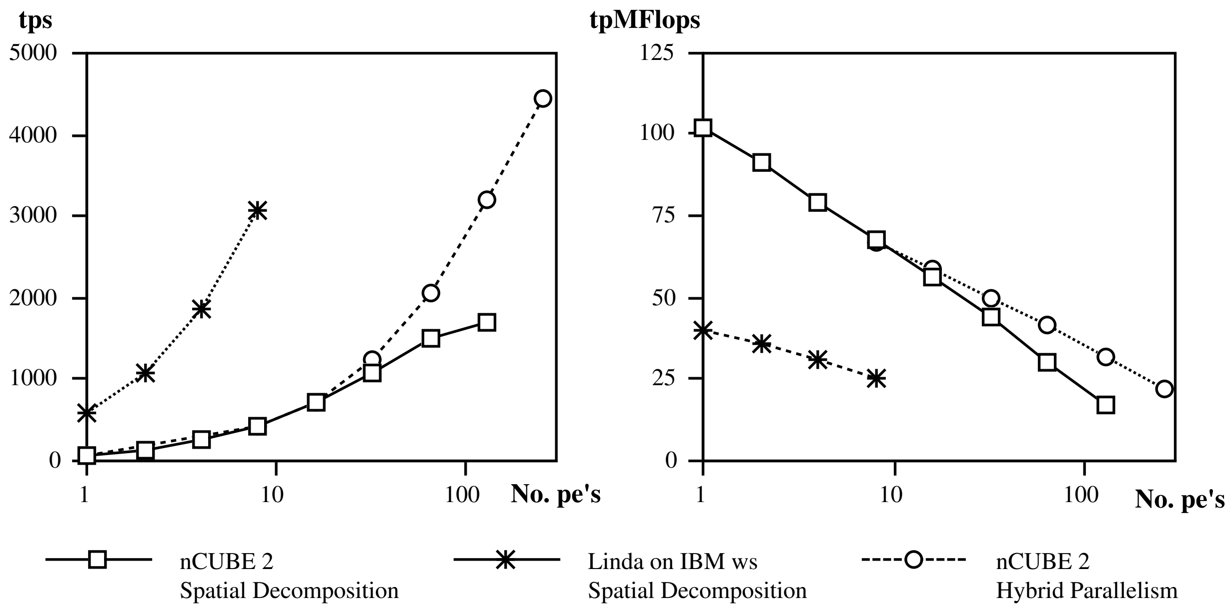


Figure 11: Behavior of some performance indexes in function of the number of *pe*'s used: tetrahedra per second (tps) and tetrahedra per MFLOPS (tpMFlops).