Parallel 3D Delaunay Triangulation

P. Cignoni[†], 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

Abstract

The paper deals with the parallelization of Delaunay triangulation algorithms, giving more emphasis to pratical issues and implementation than to theoretical complexity. Two parallel implementations are presented. The first one is built on DeWall, an E^d triangulator based on an original interpretation of the divide & conquer paradigm. The second is based on an incremental construction algorithm. The parallelization strategies are presented and evaluated. The target parallel machine is a distributed computing environment, composed of coarse grain processing nodes. Results of first implementations are reported and compared with the performance of the serial versions running on a Unix workstation.

Keywords: Delaunay triangulation, divide & conquer, uniform grids, parallel processing, distributed computing.

1 Introduction

Triangulation is a well known topic of computational geometry. It is routinely used in a broad range of applications, such as robotics, computer vision and image synthesis, as well as in mathematics and natural sciences. Delaunay Triangulation (DT) is a particular type of triangulation well known in Computational Geometry; many algorithms have been proposed for the DT of a set of sites in E^2, E^3 or E^d [2].
Volume Rendering [16] is one of the latest applications of DT. A *volume* dataset consists of sampled points in E^3 space, with one or more scalar or vector values associated with each point. The spatial arrangement of pointsets 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 execution of a class of surface reconstruction or direct rendering algorithms. The large number of sites Volume Rendering applications have to manage imposes strong efficiency constraints on the triangulator used. The same constraints are also common to other applications, such as, for example, digital terrain modeling [12] where digital maps with O(100K) triangular elements are common.
High efficiency of a triangulator can be achieved by the use of optimization techniques, which can reduce the complexity of the basic algorithm, and by the exploitation of parallelism.
This paper describes the parallelization of two Delaunay triangulation algorithms whose sequential implementations were presented in detail in a previous paper [5]. The first algorithm is based on a particular reinterpretation of the *divide & conquer* paradigm, which makes it possible to specify both a generalized solution to DT in any dimension and a simple parallel implementation. The second is an optimized implementation of the *incremental construction* method.

The 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 regarding parallelism and Delaunay triangulation. In Section 3 we describe the sequential divide & conquer algorithm and the optimization techniques used, followed in Section 4 by the description of the parallel divide & conquer solution. In Section 5 we present and evaluate a parallel incremental construction algorithm. Concluding remarks are in Section 6.

2 Delaunay Triangulations: definition and algorithms

Given a pointset P in E^d, *k–simplex* is defined as the convex combination of $k+1$ affinely independent points in P, called vertices of the simplex (e.g., a triangle is a 2-simplex and a tetrahedron is a 3-simplex). An *s-face* of a simplex is the convex combination of $s+1$ vertices of the simplex (i.e., a *2-face* is a triangular facet, *1-face* is an edge, *0-face* is a vertex).

The **Delaunay Triangulation** (DT(P)) [21] of the E^d space defined on a pointset P is the set of d-simplices such that:

1. a point p in E^d is vertex of a simplex in DT(P) iff $p \in P$;
2. the intersection of two simplices in DT(P) is either an empty set or a common face;
3. the hyper-sphere circumscribed around the $d+1$ vertices of each simplex contains no other point of the set P.

The duality between DT and Voronoi diagrams is well known [21] and therefore algorithms are given for the construction of the former from the latter, but direct methods for constructing DT are generally more efficient. DT algorithms [2] can be classified as follows:

- *local improvement*
- *on-line* (or *incremental insertion*)
- *incremental construction*
- *higher dimensional embedding*
- *divide & conquer*

On-line [10] methods start with a simplex which contains the pointset, then insert the points in P one at a time: the simplex containing the currently added point is partitioned into sub–simplices by inserting it as a new vertex. All the simplices adjacent to the new ones are then recursively tested for equiangularity in topological order and, if necessary, their faces are flipped. This method in its naive version is extremely simple to program and can be generalized to manage pointsets in E^d space. Moreover, this approach has a lower asymptotic mean time complexity of $\mathcal{O}(n\ log\ n + n^{\lceil \frac{d}{2} \rceil})$ [13].

Incremental construction methods use the empty circle property to construct the DT by successively building simplices whose circum-hyperspheres contain no points in P [19]. Their efficiency is low, in the naive implementation, ($\mathcal{O}(n^3)$ in the worst case in E^3), but effective acceleration techniques can be devised [5].

Higher dimensional embedding methods transform the points in the E^{d+1} space and then compute the convex hull of the transformed points; the DT is obtained by simply projecting the convex hull back in E^d [2].

Divide & Conquer (D&C) algorithms have been proven to be optimal in the E^2 space in both mean and worst–case time complexity [17]. 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. D&C algorithms presented in the literature manage E^2 pointsets only. This, because specification of the merge phase is simple in E^2 due to the explicit ordering of the edges incident on a vertex. But this simple ordering does not hold in E^d ($d > 2$) and specification of the merge phase is thus difficult.
A new algorithm has been proposed by the authors which bypasses this problem by reversing the order between resolving the sub–problems and the merging phase [5]. Classic D&C algorithms recursively subdivide the input dataset, construct the two DT's and then merge them. DeWall, the new algorithm, first subdivides the input dataset, then builds that part of the DT that should 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 merge triangulation. This makes it possible to specify a general E^d D&C triangulator, with a simple structure that permits an efficient implementation using some well known optimization techniques.

2.1 Parallel Delaunay Triangulation

The increasing complexity of the input dataset, together with current technological trends gives increasing importance to parallel solutions. In this context, *on-line* algorithms are penalized because their parallel implementation involves considerable overheads due to the high degree of interprocess communications and synchronization needed. On the other hand, *divide & conquer* methods can be easily parallelized. To date only few papers have been published reporting the implementation of parallel algorithms for Delaunay triangulation.

Davy and Dew [6] have proposed a parallel DT for 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 allocated to coarse–grained processing elements. Each processor uses an *on–line* algorithm to build a 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 a Meiko Computing Surface, and the reported speedups range from 4.09 to 5.7 on eight processing nodes.

Some first results on parallel Delaunay triangulation were reported by Beichl and Sullivan [3]. Their paper deals with implementation of an incremental construction algorithm on a Thinking Machine, CM2, a data parallel architecture. The complexity of the parallel algorithm is $\mathcal{O}(M(n)T(n))$ in any dimension, where $M(n)$ is the complexity for determining the min/max of n items, and $T(n)$ the number of tetrahedra in the resulting triangulation. But the reported times and speedups are not impressive: the triangulation of 1000 sites in E^3 runs on a CM2, a machine with thousands of processors, only twice as fast as the sequential implementation of the same algorithm on a workstation.

Some other papers deal with the parallel computation of both Voronoi diagrams [11] [14] [15] and convex hulls [7] [4]. It is well–known that DT can be derived from Voronoi diagrams, therefore parallel Voronoi algorithms can be used to accelerate Delaunay computation. Moreover, *higher dimensional embedding* algorithms build the Delauny triangulation by first computing the convex hull of the pointset trasformed in E^{d+1} space; this phase can therefore be made parallel by the use of 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 overhead of the solution.

3 The DeWall triangulation algorithm

An algorithm for DT of a pointset P in E^d is presented in this section. The algorithm, described in detail in [5], is based on the divide & conquer paradigm applied differently from the usual D&C Delaunay algorithms [17] [9].
In our problem the pointset P can easily be split, using a cutting plane α, in two subsets P^- and P^+ of comparable cardinality. In D&C triangulation algorithms the problem is the logical complexity of the merge phase, i.e. the union of the two triangulations S^- and S^+ built on P^- and P^+. It requires a number of local modifications to the partial solutions S^- and S^+. It has been mentioned in Section 2 that the merge phase can be efficiently solved in E^2 [17], but there are as yet no proposals for D&C Delaunay triangulations in E^d, $d > 2$.
Our approach to divide & conquer DT is slightly different and has come about as the result of designing a simple and generalized to E^d way to merge subproblem solutions (i.e. local triangulations).
The main idea is simple. During the splitting phase we subdivide both the pointset and the problem solution; the latter is divided in three subset of simplices: the first, known as *simplex wall*, is directly built during the splitting phase, while the other two are computed recursively by the D&C algorithm.

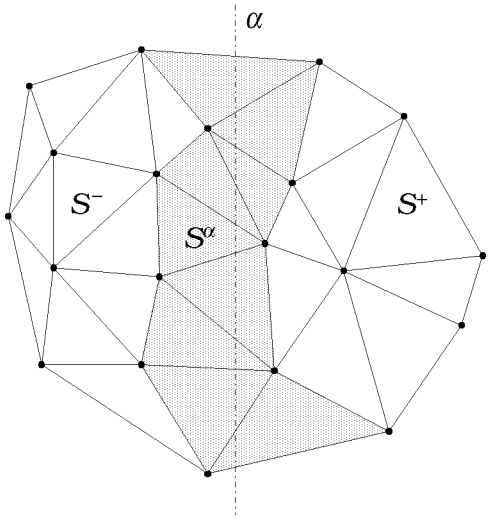

Figure 1: A triangulation in E^2 (S^α: gray triangles).

The main characteristic of the simplex wall is that it can be considered as a valid *merging set* for the two partial solutions which are returned by the recursive invocation of the algorithm.
The efficiency of this approach depends on the complexity of constructing such Delaunay simplicial complex. Moreover, efficiency is increased if the computations needed to built the simplex wall in the splitting phase are exploited in the subsequent recursive invocations of the algorithm.

The **DeWall** (**De**launay **Wall**) algorithm is based on the above approach.
The splitting plane α divides the space E^d into two halfspaces, called PosHalfspace(α) and NegHalfspace(α); α divides a triangulation DT(P) into three subsets: the simplices that are intersected by the plane, which we call *simplex wall* S^α, the simplices S^+ that are completely contained in the PosHalfspace(α) and the simplices S^- contained in the NegHalfspace(α) (Figure 1).
S^α is a suitable candidate for the required S_{Merge}: it is part of a valid DT(P) and for each simplex $s \in$ DT(P), s is intersected by α iff $s \in S^\alpha$.

The DeWall algorithm is composed of the following steps:

- select the splitting plane α;
- construct S^α and the two subsets P^- and P^+;
- recursively apply DeWall on P^-, starting from S^α, and build S^-;
- recursively apply DeWall on P^+, starting from S^α, and build S^+;
- return the union of S^α, S^- and S^+.

An example of DT in E^2 space is given in Figure 1: α is the splitting plane, and S^α (the gray triangles) is the associated simplex wall; S^- and S^+ are the triangulations returned by recursive invocations of the DeWall algorithm on the two pointset partitions.

SimplexWall construction

The merging triangulation can be built in a very simple manner using a constructive rule similar to that proposed by Mc Lain in his **incremental construction** algorithm [19]. Mc Lain's algorithm uses the empty circle property to construct the DT by successively building simplices whose circum-hyperspheres contain no points in P. The algorithm starts from a simplex $s \in DT(P)$ and incrementally builds the $DT(P)$ by adding a new simplex in each step without having to modify the current triangulation.
The algorithm is based on the existence of a pair of adjacent simplices for each (d-1)face which does not belong to the $ConvexHull(P)$. Starting with the initial simplex, its (d-1)faces are visited and the simplex

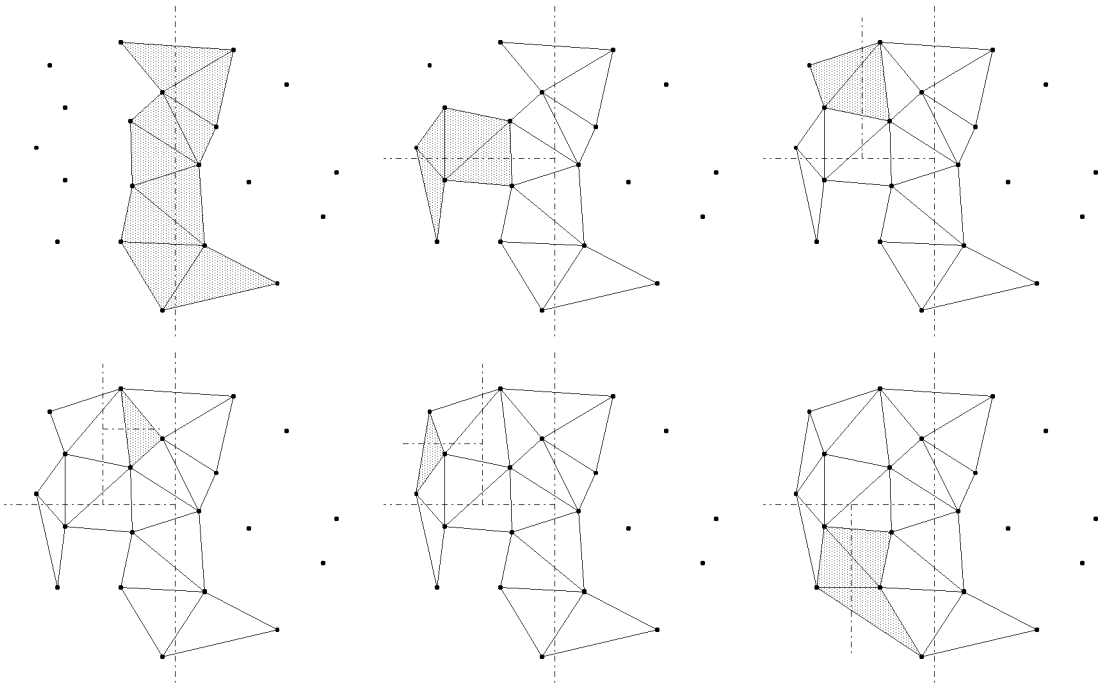

Figure 2: Individuation of the first six simplex walls on a E^2 dataset.

adjacent to each of them is built and added to the current list of simplicia. For each added simplex all of its (d-1)faces are inserted into an Active Face List (AFL). Insertion into 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 into AFL because the simplex adjacent to f has not yet been built. The process continues iteratively (extract a face f' from AFL, build the simplex s' adjacent to f', insert the (d-1)faces of s' in AFL, and then again extract a face from AFL) until AFL is empty.
The difference from the McLain algorithm is that DeWall must build first only the simplices in S^α. This can be done by applying a similar approach and using three different active face lists:

- AFL^α : (d-1)faces intersected by plane α;
- AFL^+ : (d-1)faces completely contained in PosHalfspace(α);
- AFL^- : (d-1)faces completely contained in NegHalfspace(α).

S^α is built incrementally (Figure 2). For each simplex s, the algorithm inserts each (d-1)face of s into one of the above face lists. Then it extracts a face (on which the next simplex will be searched for) from AFL^α only; this ensures that the next simplex is part of S^α.
The S^α construction terminates when the AFL^α is empty. The process returns both S^α and the active face lists AFL^+ and AFL^-. Then DeWall is recursively applied to the pairs (P^-, AFL^-) and (P^+, AFL^+).
The splitting plane α is cyclically selected as a plane orthogonal to the axes (X, Y or Z in E^3) in order to recursively partition the space with a regular pattern (Figure 2).

Given a face f, the adjacent simplex can be identified by using the Delaunay simplex definition: all the points $p \in P$ are tested by checking the radius of the hypersphere which circumscribes p and the d vertices of f. In the pseudo code in Figure 3 the function *MakeSimplex* implements the adjacent simplex construction. The analysis of the points $p \in P$ is limited by considering only those points which lie in the *outer* halfspace with respect to face f (i.e. the halfspace which does not contain the previously generated simplex that contains the face f). *MakeSimplex* selects the point which minimizes the function dd (Delaunay distance):

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

```
Function DeWall (P: point_set, AFL: face_list): simplex_tessellation;
    var f: face; AFL^α, AFL^−, AFL^+: face_list;
        t: simplex; DT: simplex_list; α: split_plane;
    begin
        AFL^α, AFL^−, AFL^+:=∅; DT:=∅;
        Pointset_Partition(P,α,P^−,P^+);
        if AFL=∅ then
            t:=Make_First_Wall_Simplex(P,α);
            AFL:=Faces(t); Insert(t,DT);
        for each f ∈ AFL do
            if Intersection(f,α) ≠ ∅ then Insert(f,AFL^α)
            else if f ⊂ NegHalfspace(α) then Insert(f,AFL^−)
            else if f ⊂ PosHalfspace(α) then Insert(f,AFL^+);
        while AFL^α ≠ ∅ do begin
            f:=Extract(AFL^α);
            t:=MakeSimplex(f,P);
            if t ≠ null then begin
                Insert(t,DT);
                for each f': f' ∈ Faces(t) AND f' ≠ f do
                    if Intersection(f',α) ≠ ∅ then Update(f',AFL^α);
                    else if f' ⊂ NegHalfspace(α) then Update(f',AFL^−);
                    else if f' ⊂ PosHalfspace(α) then Update(f',AFL^+);
                end;
            end;
        if AFL^− ≠ ∅ then DT:=DT ∪ DeWall(P^−,AFL^−);
        if AFL^+ ≠ ∅ then DT:=DT ∪ DeWall(P^+,AFL^+);
        DeWall:=DT;
    end;

Procedure Update (f:face, L:face_list);
    begin
        if f ∈ L then Delete(f, L)
            else Insert(f, L);
    end;
```

Figure 3: DeWall algorithm specification in pseudo–Pascal.

with r and c the radius and the center of the circumsphere around f and p. The outer halfspace associated with f contains no point *iff*, face f is part of the Convex Hull of the pointset P; in this case the algorithm correctly returns no adjacent simplex and, in this case only, $MakeSimplex$ returns $null$.

The function *Make_First_Wall_Simplex*, used in the pseudo–code of Figure 3, returns the first simplex on which S^α is then built. For correct construction of S^α, this first simplex must intersect the plane α. The*Make_First_Wall_Simplex* function selects the point $p_1 \in P$ nearest to the plane α. Then it selects a second point p_2 such that: (a) p_2 is on the other side of α from p_1, and (b) p_2 is the point in P with the minimum Euclidean distance from p_1. Then, it seeks the point p_3 at which the radius of the circum-circle about 1-face (p_1, p_2) and point p_3 is minimized: the points (p_1, p_2, p_3) are a 2-face of the DT(P). The process continues in the same way until the required first d-simplex is built.

3.1 Speedup techniques

Analysis of the algorithm reveals two main bottlenecks:

- *MakeSimplex* function: given the face f, computation of $dd(f, p)$ is required for each point $p \in P$;
- *active face list* management: if implemented as a simple list the operations have a complexity proportional to the number of faces currently stored.

Table 1: DeWall sequential run times on a SUN SPARC II ws, in seconds (Opt.: optimized algorithm).

Seq DeWall (number of sites)	2000	4000	6000	8000	10000	20000
DeWall	119	372	809	1292	1905	5930
Opt. DeWall	16	35	57	76	98	211

An empirically estimated complexity lower than $\mathcal{O}(nlogn)$ was obtained using the optimization techniques presented below, (on 20k site pointsets, triangulation times reduced from 5930 to 347 seconds on a SUN SPARC II workstation, Table 1).

3D Gridding

The basic concept of *local processing* is often adopted in computer graphics to achieve either speedup of sequential algorithms or parallelism. Delaunay triangulators can be made more efficient by using a 3D extension of the *Uniform Grid* [1]

$$UG = \{c_{ijk}\}; \qquad i, j, k \in [0..N] \tag{1}$$

i.e., a regular, not–hierarchical 3D grid, which subdivides the dataset space without overlap or omission, and where the grid resolution adapts to the data.
The key reason why uniform grid techniques are effective in geometric computations is that two points which are far apart have little or no effect on each other. A large class of geometric algorithms possesses this property [20] and this holds true for DT as well.

To construct a new simplex in expected constant time the *uniform grid* is used as an indexing scheme for fast detection of the *dd*-nearest point. Given the dataset P, it is possible to associate each cell c_{ijk} with the list of points in P that are contained in c_{ijk}. A preprocessing phase builds the UG structure. Analougous to Maus's approach [18], *MakeSimplex* function examines 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.
This optimization techinque can be simply extended to E^d triangulation algorithms.

Face lists management via hash tables

Operations need to be carried out on the AFL (active face list) data structure to insert, to extract, to remove a face or to test if a face belongs to the list.
A time complexity which is linear to the number of elements in the list considerably reduces the efficiency of the algorithm. The current implementation of the AFL data structure is based on hash coding; the hash code associated to each face makes it possible to access each face in the list in a nearly constant amount of time (1.15 - 1.5 accesses for each search).

4 Parallel DeWall

The parallelization of the DeWall algorithm is simpler, in comparison to other D&C algorithms. A usual D&C algorithm requires two synchronization points: the first at the partitioning of the input dataset before process splitting, and the second at merge time, when both split processes terminate. In the case of the DeWall algorithm, merging is not required, and the two subprocesses can then run asynchronously. This implies simpler implementation.
The parallelization strategy described here is designed for a parallel computing environment in which processes cannot be efficiently created at run time. The code is replicated on the processing elements (p.e.). Each p.e. reads in the pointset P and receives in input its identification number (pe_id). Then, on the basis of its pe_id, each node is able to selectively generate only a subset of the recursion tree. The k–th p.e. will execute all recursive calls in the path from the tree root to the root of the subtree associated to p.e. k; then, p.e. k executes all calls in the associated subtree.
The solution replicates part of the computations on the processing elements, but the parallelization of

Table 2: Parallel DeWall run times, in seconds, on an nCUBE 2 hypercube.

Parallel times	1 p.e.	2 p.e.	4 p.e.	8 p.e.	16 p.e.
2000 sites					
times	92.	54.4	39.2	33.2	31.5
speedup	1	1.69	2.34	2.77	2.92
efficiency	1	0.84	0.58	0.34	0.18
4000 sites					
times	209.75	123.43	87.41	73.39	68.14
speedup	1	1.69	2.39	2.85	3.07
efficiency	1	0.84	0.59	0.35	0.19
6000 sites					
times	325.47	190.75	132.88	107.1	98.65
speedup	1	1.70	2.44	3.03	3.29
efficiency	1	0.85	0.61	0.37	0.20
8000 sites					
times	445.36	260.87	180.68	145.69	132.82
speedup	1	1.70	2.46	3.05	3.35
efficiency	1	0.85	0.61	0.38	0.20

the sequential code is straightforward and makes it possible to write a parallel triangulator in just a few hours.

The parallel version of the DeWall algorithm can be concisely specified as follows:

```
Function ParDeWall (pe_id, rec_call:integer; P:pointset; AFL:face_list):simplex_list;
    var AFL^α, AFL^−, AFL^+:face_list;
        DT:simplex_list; α:splitting_plane;
    begin
        PointsetPartition(P, α, P^−, P^+);
        MakeWall(P, α, AFL, DT, AFL^α, AFL^−, AFL^+, P^−, P^+);
        if not must_save(pe_id) then DT:=∅;
        if (AFL^− ≠ ∅) and (on_path(pe_id, rec_call∗2))
            then DT:=DT ∪ DeWall(pe_id, rec_call∗2, P^−, AFL^−));
        if (AFL^+ ≠ ∅) and (on_path(pe_id, rec_call∗2 + 1))
            then DT:=DT ∪ DeWall((pe_id, rec_call∗2+1, P^+, AFL^+);
        ParDeWall:=DT;
    end;
```

where: *MakeWall()* is a function which hides most of the pseudo–code of the DeWall specification in Figure 3 (this part of the code remains unchanged in the parallel version); *on_path(p,i)* is a boolean function which returns true if the recursive call of index i is in the path assigned to processing node p. Recursive call indices are assigned as follows: 1 is the root, 2 is the first left recursion, 3 is the first right recursion, and so on.

An advantage of the parallel DeWall is its simplicity; it needs only to transform the plain recursive calls of the sequential code into conditional invocations depending on the current pe-id number. Moreover, no communications are needed between p.e.: the computations on different p.e. are completely independent. An evident disadvantage is the replicated computation of all the recursive calls at level $i < \log m$, with m the number of p.e. used. Each recursive call at level i is executed on $m/2^i$ p.e., and all of them produce identical results. The *must_save(pe_id)* boolean function prevents replication in the output data: only one of the $m/2^i$ p.e. actually returns its i-level tetrahedra wall.
Also in the case of a dynamic implementation of the D&C paradigm is used (where each process forks on a different p.e. before executing only one of the two recursive calls), the level of parallelism, i.e. the elapsed time, remains the same.

Run times on a number of uniformly distributed datasets are reported in Table 2. These times have been obtained using different number of p.e. on a hypercube multicomputer, an nCUBE 2 system model

Table 3: Run times, in seconds, required for the subphases of the DeWall algorithm on a SUN SPARC II workstation (20K sites).

Profiling 20000 sites	Part.&Grid	MakeWall	Subtrees
rec_call 1	0.51	31.48	174.87
rec_call 2	0.26	11.49	74.62
rec_call 3	0.26	13.45	74.78
rec_call 4	0.13	3.67	32.52
rec_call 5	0.13	3.95	33.16
rec_call 6	0.14	4.40	32.98
rec_call 7	0.14	4.77	33.38
rec_call 8	0.06	2.45	14.25
rec_call 9	0.06	2.49	14.20
rec_call 10	0.07	2.09	13.98
rec_call 11	0.07	2.19	14.16
rec_call 12	0.07	2.25	14.06
rec_call 13	0.07	2.31	14.24
rec_call 14	0.07	2.33	13.82
rec_call 15	0.07	2.46	14.20

Table 4: InCoDe sequential run times on a SUN SPARC II ws, in seconds (Opt.: optimized algorithm).

Seq InCoDE (number of sites)	2000	4000	6000	8000	10000	20000
InCoDE	796	3620	8838	16242		
Opt. InCoDE	18	42	71	103	137	327

6410. The nCUBE has a MIMD distributed memory architecture, with 4 Mbytes of local memory on each p.e. in the configuration used. Running at a clock rate of 20 MHz, the nCUBE node processor is rated at 7.5 MIPS and 3.3 MFLOPS, single precision, or 2.4 MFLOPS, double precision.
The results of parallel DeWall were below expectations in terms of effective speedup. It is well–known that the D&C paradigm does not scale efficiently on machines with a large number of p.e., but DeWall suffers from a further problem: the MakeWall phase involves a computational cost which is much higher in the first recursive calls than in the rest of the computations.
Table 3 reports profiling data which show how fast the MakeWall times decrease. The time refers to the triangulation of a 20K–site dataset on a SUN SPARC II workstation (28.5 MIPS and 4.2 MFLOPS in double precision). *Part&Grid* is the time required in each recursive call to partitionate the pointset and to build the local uniform grid. In each row i the *rec_call* column contains the time needed to execute all the recursive calls which originates from the *i-th* recursive call . The MakeWall time at the root level recursion is 31.48 seconds and reduces to a few seconds at the third level of recursion.
Load imbalance slightly reduces as the resolution of the dataset increases but, this notwithstanding, effective scalability is precluded by the computational characteristics of the DeWall algorithm. If the aim is to run a parallel Delaunay triangulator on a network with few cooperating workstations (4/8 nodes) or on a multi–cpu workstation, the absence of communications and the extremely low cost of parallelizing the sequential code justify the adoption of such an approach.
Whether the goal is the use of a distributed computing environment with tens of cooperating p.e., or of parallel MIMD architectures, DeWall cannot guarantee a good exploitation of the processing power.

5 A parallel incremental construction algorithm

A second parallel solution, aimed at increasing scalability, is presented in this section.
The solution is based on an *incremental construction* method [19] [8]; the characteristics of this class of algorithms were briefly introduced in Sections 2 and 3, and a detailed description of one particular implementation, the InCoDe algorithm, can be found in [5].

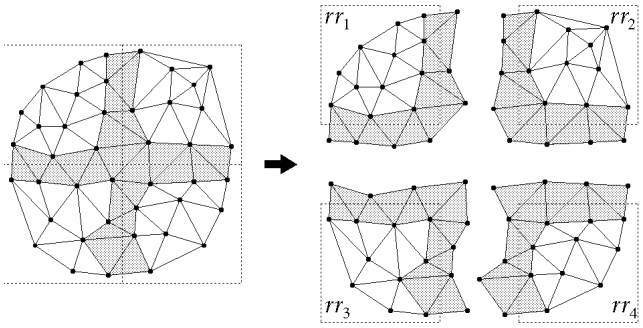

Figure 4: Partition of the triangulation space in four regions; in gray, the duplicated triangles.

Table 5: Parallel InCoDe: nCUBE run times, in seconds.

Parallel times	t_{min}	t_{max}	t_{mean}	$speedup$	$eff.$	$\|DT(P)\|$	$repl.tetra$	%
10000 sites								
1 p.e.	761.68			1	1	66445	0	0.
2 p.e.	417.49	436.26	426.87	1.74	0.87		3711	5.5
4 p.e.	210.52	252.44	231.34	3.01	0.75		7605	11.4
8 p.e.	116.67	149.96	128.34	5.07	0.63		11739	17.6
16 p.e.	64.07	95.32	75.83	7.99	0.49		20224	30.4
32 p.e.	36.59	59.92	45.52	12.71	0.39		29238	44.0
64 p.e.	20.39	39.67	27.26	19.20	0.30		39046	58.7
20000 sites								
1 p.e.	1676.25			1	1	133450	0	0.
2 p.e.	916.89	933.97	927.43	1.79	0.89		6016	4.5
4 p.e.	511.90	538.52	521.33	3.11	0.77		12197	9.1
8 p.e.	251.64	315.54	277.48	5.31	0.66		18587	13.9
16 p.e.	139.04	189.15	162.34	8.86	0.55		31418	23.5
32 p.e.	75.25	122.61	96.59	13.67	0.42		44957	33.6
64 p.e.	37.21	88.21	57.26	19.01	0.29		59139	44.3

```
Function ParInCoDe (P:pointset, rr_i:rect_reg):simplex_list;
    var f:face; AFL:face_list;
        t:simplex; DT:simplex_list;
    begin
        AFL:=emptylist;
        t:=MakeFirstSimplex(P,rr_i); {the first simplex must be in rr_i}
        Insert(t,DT);
        AFL:=faces(t);
        while notempty(AFL) do begin
            f:=Extract(AFL);
            t:=MakeSimplex(f, P);
            if t ≠ null then begin
                Insert(t,DT);
                for each f': f' ∈faces(t) AND f' ≠ f AND f' ∩ rr_i ≠ ∅ do Update(f',AFL)
                end;
            end;
        ParInCoDe:=DT;
    end;
```

Figure 5: Parallel InCoDE algorithm.

InCoDe (**In**cremental **Co**nstruction of **De**launay triangulations) is based on the classic incremental construction method, i.e. with the same approach used to build the simplex walls in the DeWall algorithm. InCoDe is optimized by means of the same techniques presented in Section 3.1, i.e. 3D gridding and hashing. Sequential run times are reported in Table 4.
The InCoDe algorithm can be simply parallelized on m independent asynchronous processors by means of pointset space subdivision. We can easily partition the bounding box of the pointset P in m rectangular regions, called $rr_i, 1 \leq i \leq m$. An InCoDe process is replicated on each p.e., together with the pointset P. A rectangular region rr_i is assigned to each InCoDe process, which will compute all of the tetrahedra in $DT(P)$ that are at least partially contained in rr_i.
The pseudo–code specification of the parallel InCoDe process is in Figure 5. The main difference from the sequential version is the management of the AFL list: for each new tetrahedron, we insert into AFL only the faces completely/partially contained in the local region rr_i. Each p.e. computes (a) the tetrahedra internal to rr_i and (b) those shared with adjacent regions (in Figure 4 a 2D example is shown).
Storing replicated tetrahedra in the output dataset can be simply prevented with a vertex–containment test. For each new tetrahedra, we include it in the local output list if and only if its upper–left–front most vertex is contained in the local rr_i region.

The times of the parallel InCoDe algorithm on the nCUBE 2 parallel machine are reported in Table 5. The table contains the times obtained with different numbers of p.e.; **tmin** and **tmax** are the times of the fastest and slowest processes. The **tmax** time is obviously used to compute speedup and efficiency. $|\mathbf{DT}(\mathbf{P})|$ is the number of simplices in the resulting triangulation, while **repl. tetra** is the number of tetrahedra built by more than one processing node (those on the borders of the regions rr_i); the column % reports the ratio between these two values and it therefore gives a measure of the overhead due to the parallelization strategy. The speedups obtained are quite satisfactory. A more balanced load could be achieved using a different partitioning rule. In this first experiment a simple regular and uniform partition was applied to divide the load between p.e., but a more sophisticated adaptive partitioning strategy can be applied as well, resulting in a more balanced load distribution.

6 Concluding remarks

Two different parallel Delaunay triangulators have been presented. Given the sequential code both the proposed parallel versions can be simply implemented on parallel environments with coarse grain processor elements, such as networks of workstation or multicomputers. The paper reports the efficiency of the implementations on different datasets.
The first algorithm, based on the divide & conquer paradigm, is faster but shows limited scalability; the best results, in terms of both performance and efficiency, have been obtained while running on 2-4 p.e. This solution is therefore well suited for multiprocessor workstations or for networks with a limited number of connected nodes.
The second algorithm is characterized, in its sequential implementation, by higher run times than DeWall, but it also demonstrates greater scalability. The resulting parallel times are therefore lower, and provide the possibility of exploiting the processing power of architectures with higher parallelism.

Acknowledgements
The authors would like to thank Leila De Floriani and Enrico Puppo for valuable suggestions and encouragements; they also gratefully acknowledge the support of our collegues in the Parallel Processing Group of CNUCE-C.N.R.
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] V. Akman, W.R. Franklin, M. Kankanhalli, and C. Narayanaswami. Geometric computing and uniform grid technique. *Computer-Aided Design*, 21(7):410–420, Sept. 1989.

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

[3] I. Beichl and F. Sullivan. Parallelizing computational geometry: First steps. *SIAM News*, 24(6):1–17, 1991.

[4] 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.

[5] 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, October 1992.

[6] 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.

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

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

[9] R.A. Dwyer. A faster divide and conquer algorithm for constructing Delaunay triangulations. *Algorithmica*, 2:137–151, 1987.

[10] 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.

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

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

[13] 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.

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

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

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

[17] 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.

[18] A. Maus. Delaunay triangulation and the convex hull of n points in expected linear time. *Bit*, 24:151–163, 1984.

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

[20] C. Narayanaswami. *Parallel Processing for Geometric Applications*. PhD thesis, Rensselaer Polytechnic Institute, Troy, NY, December 1990.

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