Parallel Processing Letters, Vol. 11, Nos. 2 & 3 (2001) 341–352
© World Scientific Publishing Company

# AN IMPROVED PARALLEL ALGORITHM FOR DELAUNAY TRIANGULATION ON DISTRIBUTED MEMORY PARALLEL COMPUTERS*

SANGYOON LEE, CHAN-IK PARK and CHAN-MO PARK

*Department of CSE/PIRL, POSTECH, San 31, Hyoja-dong, Nam-gu*
*Kyoungbuk, Pohang, 790-784, Korea*
*E-mail: cipark@postech.ac.kr*

Received September 2000
Revised January 2001
Accepted by Y. Muraoka

ABSTRACT

Delaunay triangulation has been much used in such applications as volume rendering, shape representation, terrain modeling and so on. The main disadvantage of Delaunay triangulation is large computation time required to obtain the triangulation on an input points set. This time can be reduced by using more than one processor, and several parallel algorithms for Delaunay triangulation have been proposed. In this paper, we propose an improved parallel algorithm for Delaunay triangulation, which partitions the bounding convex region of the input points set into a number of regions by using Delaunay edges and generates Delaunay triangles in each region by applying an incremental construction approach. Partitioning by Delaunay edges makes it possible to eliminate merging step required for integrating subresults. It is shown from the experiments that the proposed algorithm has good load balance and is more efficient than Cignoni et al.'s algorithm and our previous algorithm.

## 1. Introduction

A triangulation for a given set of points is a well known topic of computational geometry [1, 2] and has many applications such as finite element analysis, solid modeling, shape representation, terrain modeling, volume rendering and computer vision [3, 4, 5]. Delaunay triangulation (DT) has been most attractive for triangulation due to its special feature that the circumscribed circle of a triangle must not constain other points than points of the triangle.

There have been much research in DT. The time complexity of constructing Delaunay triangles for a set of $n$ points is $\Omega(n \log n)$ under uniprocessor environment [12]. Reducing the time complexity is made possible by using more than one processor [6, 7, 8, 9, 10]. Saxena et al. [6] have proposed two parallel algorithms on the orthogonal tree network of $n \times n$ processors; one is for 2-dimensional DT with

*The earlier version of this paper has been presented at 1997 Advances in Parallel and Distributed Computing Conference.

the time complexity of $O(\log^2 n)$ and the other is for 3-dimensional DT with the time complexity of $O(m^{1/2}\log n)$, where $n$ represents the number of points and $m$ represents the number of tetrahedra constructed in case of 3-dimensional DT. Even though their algorithms are quite efficient, they require special architecture.

In general distributed memory parallel computers, Cignoni et al. have proposed two parallel algorithms in [8]; one (called 'ParDeWall') from a sequential algorithm (called 'Delaunay Wall[9]') based on divide & conquer paradigm and the other (called 'ParInCoDe') from a sequential algorithm (called 'Incremental Construction of Delaunay Triangulation[9]') based on an incremental construction method. ParDeWall algorithm exploits parallelism found when a set of points is recursively partitioned by some cutting planes and each subregion containing only one cutting plane is assigned to one processor. Each processor generates a set of triangles called simplex wall whose edges are crossing its corresponding cutting plane and then partitions its subregion by a newly generated cutting plane. Though this algorithm is simple, the degree of parallelism is very limited since subproblems are generated in the form of binary tree and parallelism comes from subproblems of the same level. We cannot obtain good scalability of it due to small degree of parallelism. The parallelism exploited in ParInCoDe algorithm comes from geometric partitioning of a set of points. Geometric partitioning means that the bounding box containing all points can be divided by subboxes (subregions) and each subregion is assigned to one processor. Each processor builds all Delaunay triangles possible for any points inside its assigned region. Then, we get the final set of Delaunay triangles by simple merging partial results generated by each processor. However, this algorithm suffers from overlapping processing because each processor generates many identical triangles. And their method causes load imbalance among processors and, even worse, it does not work in a certain case. In addition, this type of algorithm like ParInCoDe requires good load balance among processors in order to get nontrivial speedup. However, the load balancing issue is completely ignored in [8].

Another parallel algorithm has been proposed by Guy E. Blelloch et al. in [10]. The algorithm is based on projections and a divide & conquer paradigm. Each subproblem is determined by a region which is represented by a border consisting of Delaunay edges and the set of points inside or on the border. At each call the region is divided into two subregions by using a median line cut of the internal points and a corresponding path of Delaunay edges which is obtained through projections and a lower convex hull. Once the subproblem has no more internal points (all points of the region is on the border), Delaunay triangles in the region are generated by the special end-game strategy. This parallel algorithm is easy to implement and is of practical use. However, the degree of parallelism is very limited for the same reason in ParDeWall algorithm. Moreover, the convex hull of points must be given in addition to points as inputs.

We have proposed a parallel algorithm in [11] which archieves nontrivial speedup by way of good load balance and elimination of overlapping processing in ParInCoDe algorithm. The main pitfall of our previous algorithm is that it requires a merging scheme on partial results and the scheme is executed sequentially on only one processor, resulting in limited speedup. In addition, it is required that the first Delaunay triangle generated by each processor must exist entirely in its own subregion.

In this paper, we propose an improved parallel algorithm which eliminates all pitfalls of our previous algorithm proposed in [11]. The proposed algorithm uses the partitioning method which is based on projections, and generates Delaunay triangles in the subregion by applying an incremental construction approach.

## 2. Delaunay Triangulation and the Incremental Construction

Given a finite set $S$ of points in the plane, a triangulation is defined as joining the points of $S$ by nonintersecting straight line segments so that every region internal to the convex hull is a triangle. In addition, a triangulation is called Delaunay if it satisfies the empty circumcircle property: the circumcircle of a triangle in the triangulation does not contain any points of $S$ in its interior [2]. The concept of a triangulation can be extended to $E^d$ space. For instance, the triangulation in $E^3$ is to obtain tetrahedra in the convex hull of points. In this paper, we consider Delaunay triangulation in $E^2$ space, e.g. a plane.

The duality between Delaunay triangulations and Voronoi diagrams is well known [1, 2], and thus algorithms are given for the construction of the former from the latter. However, it is generally more efficient to directly construct the triangulation, and in fact the construction time for a Delaunay triangulation from a Voronoi diagram is $O(n)$ [2].

Incremental construction method was originally proposed by McLain[12] to construct Delaunay triangulation, which directly use the empty circumcircle property of the triangulation.

Our proposed algorithm uses a variant of McLain's original method, which was proposed by Cignoni et al. in [9]. Given a Delaunay edge $e$, the method constructs a new Delaunay triangle by selecting the point which minimizes Delaunay distance defined as $dd(e,p)$ for each point $p$ :

$$dd(e,p) = \begin{cases} r & \text{if } c \in \text{the outer half space of } e \\ -r & \text{otherwise} \end{cases} \tag{1}$$

where $r$ and $c$ are the radius and the center of the circumcircle around $e$ and $p$, respectively (see Fig. 1).

The efficiency of the incremental construction method is low ($O(n^2)$ in the worst case), but the method is practically good by using some speedup techniques such as hashing and the uniform grid [13].

## 3. Partitioning Methods

The proposed algorithm partitions a convex bounding region of given points into subregions according to paths consisting of Delaunay edges. A path which partitions subregions is constructed by a projection-based method on Delaunay edges [10]. We

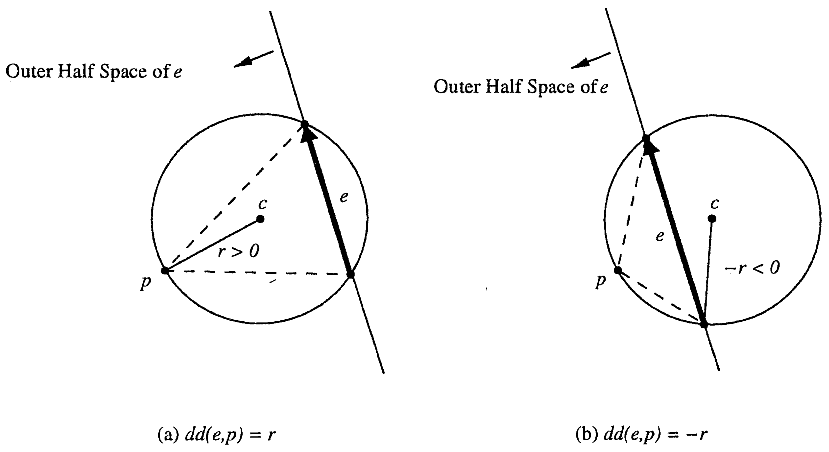


Fig. 1. Delaunay distance($dd(e,p)$) where $e$ is a Delaunay edge and $p$ is a point.

consider two partitioning methods using the paths. One divides a region into two subregions according to a vertical path, and then each subregion divides according to a horizontal path. This is recursively continued until the number of subregions equals to that of processors. The other divides a region into $m$ subregions according to just vertical (or horizontal) paths where $m$ is the number of processors.

## *3.1. The Projection-based Construction of a Path Consisting of Delaunay Edges*

Given a set $P$ of points in $E^2$ space, a vertical (or horizontal) path, which consists only of Delaunay edges and passes through a point $p \in P$ and divides the convex hull of $P$ into two regions, can be constructed by the following method[10]:

1. Let $L$ be the line $x = p_x$ (or $y = p_y$) where $p = (p_x, p_y)$.

2. Project all points of $P$ on a 3-D paraboloid centered at $p$, and then project them on the vertical plane (or horizontal plane) along the line $L$. The set $P'$ of projected points is:
$$\{(q_y - p_y, ||q - p||^2) | q \in P\}$$
$$(\text{or}\{(q_x - p_x, ||q - p||^2) | q \in P\}).$$

3. Construct the lower convex hull of $P'$. The set of edges in $P$ corresponding to this lower convex hull is a path consisting of Delaunay edges.

From now on, we assume that all paths are constructed by this method, thereby satisfying all these properties.

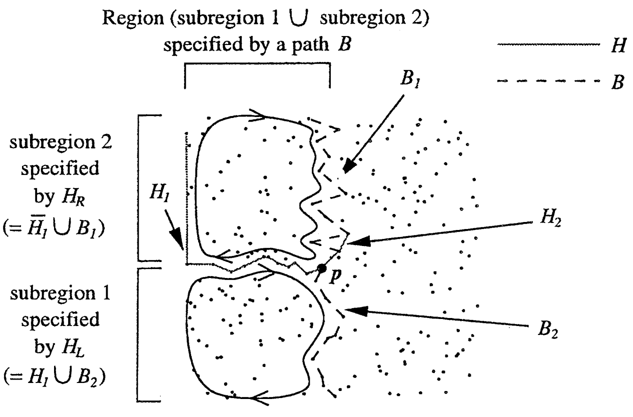


Fig. 2. An example of the merged paths at the 2nd level. $H_1 \cup B_2$ and $\overline{H}_1 \cup B_1$ are the merged paths $H_L$ and $H_R$ for the subregion 1 and 2, respectively.

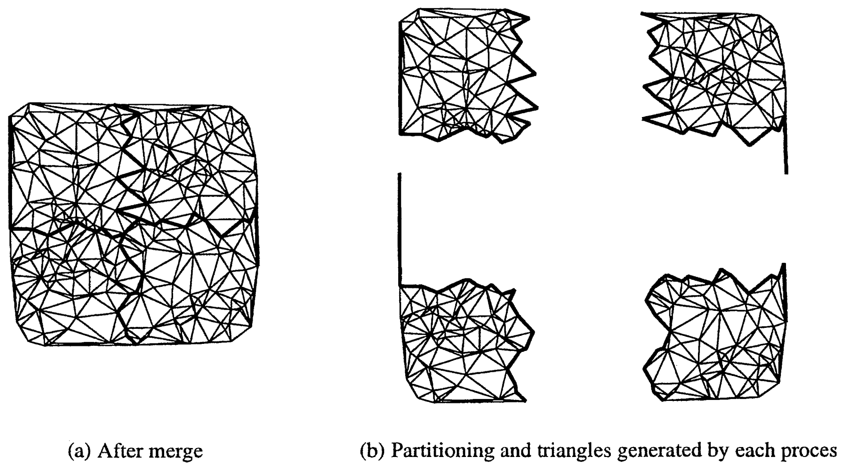


Fig. 3. An example of the partitioning using the method 1 for 4 processors.

### *3.2. The Partitioning Method 1*

Given a set $P$ of points, the convex bounding region of $P$ is divided into $2^k$ subregions, where the number of processors is $2^k$ for an interger $k$, by the following method:

1. Find the point $q$ that is the median along the $x$ (or $y$) axis of points of the region.

2. Construct a vertical (or horizontal) path $H$ passing $q$. Since a region is specified by a set of directed paths, we assume that $H$ has its own direction. Without loss of generality, assume that a vertical path has a direction from top to bottom and a horizontal path nas a direction from left to right. Let $\overline{H}$ be the path traversed in the opposite direction of $H$.

3. Divide the region into two subregions by the path $H$:

   (a) One subregion includes all the points on the left (or down) side of $H$ as well as the points on $H$. We can specify this subregion by the path $H_L$ obtained from $H$ and the path representing the given region.
   (b) The other subregion includes all the points on the right (or up) side of $H$ as well as the points on $H$. This subregion is specified by the path $H_R$ obtained from $\overline{H}$ and the path representing the given region.

4. Recursively repeat 1-3 for each subregion vertically or horizontally in turn until the number of subregions becomes $2^k$.

Checking the geometric relationships between Delaunay edges of $H$ and those of the current border is required when we construct merged paths $H_L$ and $H_R$ for a given $H$ because a portion of the path $H$ may be outside of the current region. For example of Fig. 2, consider a region specified by a path $B$. The region is partitioned by a path $H$ and we can see that a portion of the path $H$, $H_2$, is outside of the region, where $H$ and $B$ intersect at a point $p$ and are divided by the point into $H_1$, $H_2$, $B_1$ and $B_2$, respectively. This portion of the path $H$ is not needed to specify a subregion, i.e., subregion 1 can be specified by $H_L(= H_1 \cup B_2)$ where $H_2$ is not included.

Fig. 3 shows an example of a partitioning result using this method when there are 4 $(= 2^2)$ processors. Note that the construction of the merged paths $H_L$ and $H_R$ is not needed at the first level, since the convex bounding region of $P$ is completely divided into two subregions by the path $H$. This means that the partitioning method does not require the convex hull of $P$ as an input.

### *3.3. The Partitioning Method 2*

Given a set $P$ of $n$ points, the convex bounding region of $P$ is divided into $m$ subregions, where $m$ is the number of processors, by the following method:

1. Find the $(m-1)$ points $q_i$ that are the $\lceil \frac{ni}{m} \rceil$th largest points along the $x$ (or $y$) axis of all points of $P$ $(i = 1 \cdots (m-1))$.

2. Construct vertical (or horizontal) paths $H_i$ passing $q_i$. Let $\overline{H_i}$ be the path traversed in the opposite direction of $H_i$.

3. The 1st partition is the subregion on the left (or down) side of the path $H_1$, which is represented by $H_1$ and the points on the left (or down) side of or on $H_1$.

4. The $m$th partition is the subregion on the right (or up) side of the path $H_{m-1}$, which is represented by $H_{m-1}$ and the points on the right (or up) side of or on $H_{m-1}$.

5. The other partitions are the subregions between the paths $\overline{H}_{i-1}$ and $H_i$, which are represented by the merged path $H_m^i$ of the two paths and includes all the points between or on the paths.

The construction of $H_m^i$ is more simpler than that of the merged path $H_L$ or $H_R$ in the partitioning method 1. $H_m^i$ is constructed by just selecting the Delaunay edges that exist on either the path $\overline{H}_{i-1}$ or $H_i$, but not both.

Fig. 4 shows an example of a partitioning result using this method when there are 4 processors.

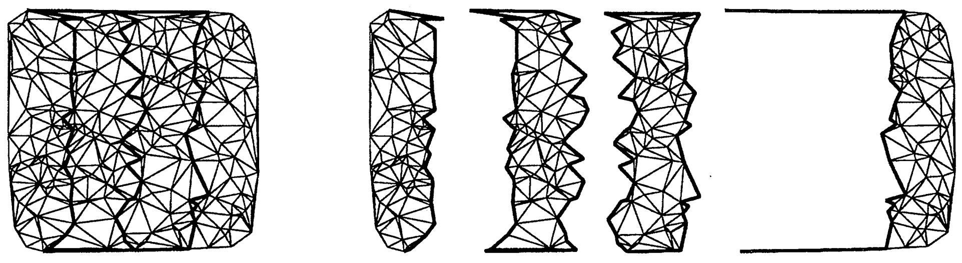


Fig. 4. An example of the partitioning using the method 2 for 4 processors.

## 4. The Proposed Algorithm

Given a set $P$ of $n$ points and the number of processors $2^k$ for an integer $k$, the proposed algorithm consists of the following three steps:

- **Step 1:** Partition the convex bounding region of $P$ into $2^k$ subregions by either the partitioning method 1 or 2 described in section 3 and assign each subregion to disjoint processor.

- **Step 2:** Each processor executes the procedure *ParDeTri* of Fig. 5 for its assigned subregion.

```
Procedure ParDeTri($P$ : pointset, $DE\_list$ : edge_list,
                   var $Partial\_DT\_list$ : triangle_list);
var
    $e$ : edge;
    $t$ : triangle;
begin
    while notempty($DE\_list$) do begin
        $e$ := Extract($DE\_list$);
        $t$ := MakeTriangle($e$, $P$);
        if $t \neq$ null then begin
            Insert($t$, $Partial\_DT\_list$);
            for each $e'$ : $e' \in$ edges($t$) AND $e' \neq e$ do
                Update($e'$, $DE\_list$);
        end;
    end;
end;

Procedure Update($e$ : edge, var $L$ : edge_list);
begin
    if $e \in L$ then Delete($e$, $L$)
    else Insert($e$, $L$);
end;
```

Fig. 5. The precedure *ParDeTri* executed on each processor.

- **Step 3:** Simply merge *Partial_DT_list* generated by each processor into one set of triangles, called *DT_list*.

In Step 1, the partitioning is performed on each processor, and each processor just constructs its own subregion which is represented by the points and a border of Delaunay edges. The border, called *DE_list*, is used for the input of the procedure *ParDeTri* in Step 2.

The function *MakeTriangle* in *ParDeTri* of Fig. 6 generates a Delaunay triangle from a Delaunay edge by selecting the point which minimizes the function *dd* for each point in the subregion, described in Section 2.

The procedure *ParDeTri* differs from the routines of Cignoni et al.'s *ParInCoDe* [8] and our previous algorithm[11] in that it does not have to specially generate the first triangle for the incremental construction since it directly uses *DE_list* for the construction, and it does not check if or not the edge exists in its own subregion. Thus it is released from the condition that the first triangle must be entirely in its own subregion. Note that this condition is essential in Cignoni et al.'s *ParInCoDe* and our previous algorithm.

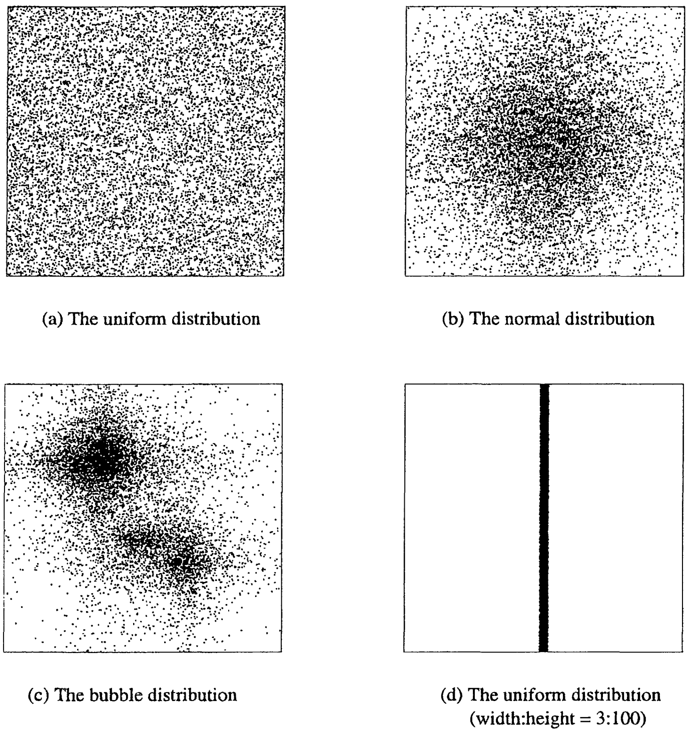


Fig. 6. The distributions for the experiment. The number of points are 10000.

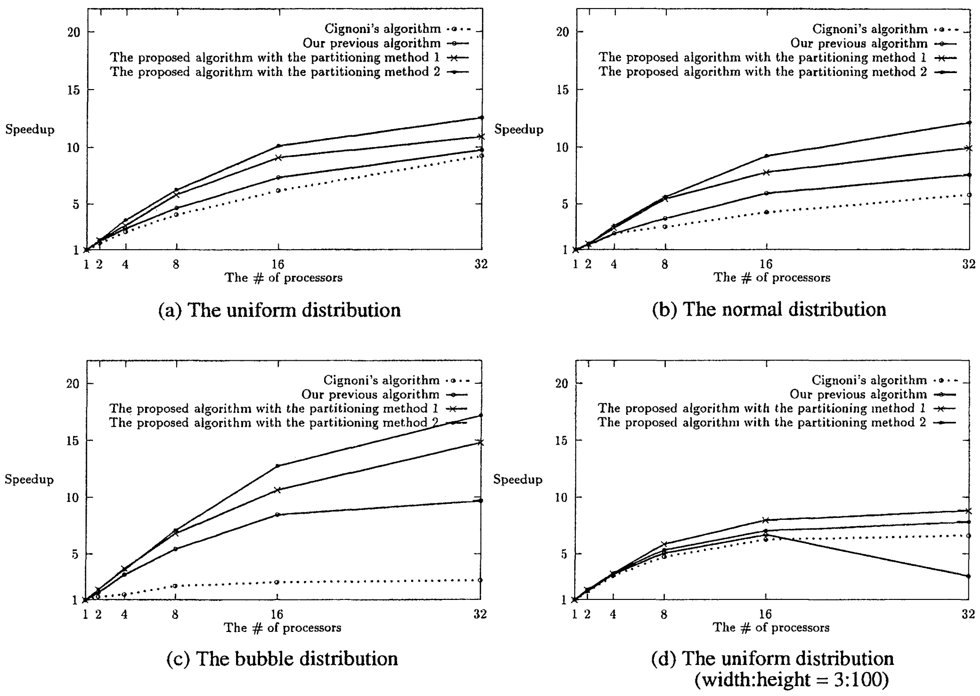
## 5. Experimental Results

The algorithm are implemented on the INMOS TRAM networks which has 32 T800 processors. The input point sets for the experiment are generated by using various distributions: the uniform distribution, the normal distribution, the bubble distribution and the uniform distribution with a much narrower width, which are shown in Fig. 6(a), 6(b), 6(c) and 6(d), respectively. In the experiment, the partitioning method 2 divides a region into subregions according to vertical paths.

The speedups of the algorithms on the INMOS TRAM networks are shown in Fig. 7. As compared with these, the proposed algorithm with the partitioning method 2 is always more efficient than the others except the last case. In the last case, the proposed algorithm with the partitioning method 1 is the best of the algorithms. This means that the proposed algorithm is more efficient than Cignoni et al.'s algorithm and our previous algorithm in the experiment.

Fig. 7(d) shows a problem of our previous algorithm that the algorithm have poor speedup with a number of processors since it should execute the sequential merging scheme on one processor (in this case, the algorithm has the more speedup with 16 processors than 32 processors).

Fig. 8 shows the breakdown of the execution time into the two components of the



(a) The uniform distribution

(b) The normal distribution

(c) The bubble distribution

(d) The uniform distribution (width:height = 3:100)

Fig. 7. The speedup graph for the distributions.

proposed algorithm (the partitioning time and the execution time of *ParDeTri*) for the distributions. As the figure shows, the execution time of *ParDeTri* is approximately the same across the partitioning methods and the distributions. For the partitioning time, the algorithm using the partitioning method 2 requires more time than the others in the uniform distribution with width:height = 3:100, and less time in the other distributions. It is because the partitioning method 2 uses just vertical paths and thus the work of the partitioning method 2 to get lower convex hulls of the projected points is more than that of the partitioning method 1 for the distribution. This explains the reason why the algorithm using the method 1 is faster than the algorithm using the method 2 in Fig. 7(d).

In order to show that the proposed algorithm has a good load balance, we calculate the standard deviation of execution times on processors for each case of the input sets. On executing the proposed algorithm on 32 processors, the standard deviations and averages of times for the distributions are shown in Table 1. Particularly, Fig. 9 shows the load of each processor for the bubble distribution.

## 6. Conclusions

In this paper, we described an improved parallel algorithm for Delaunay triangulation that eliminates the sequential merging step executed on one processor.

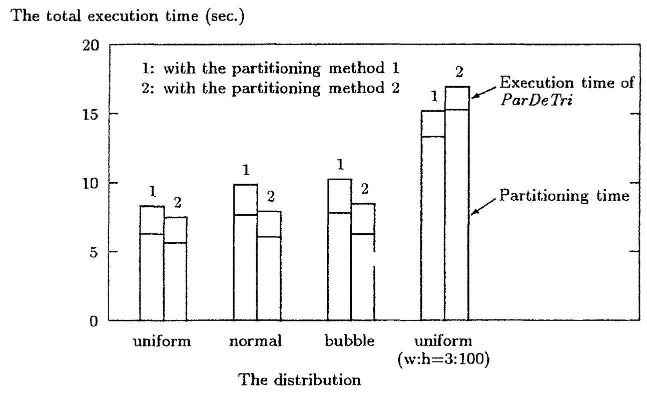


Fig. 8. The breakdown of the execution time into the two components of the proposed algorithm for the distributions.

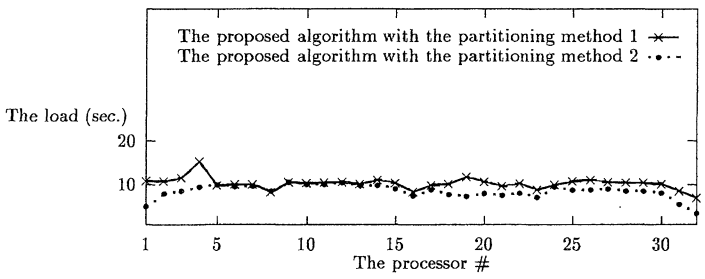


Fig. 9. The load of each processor for the bubble distribution.

Table 1. The standard deviations of running times obtained in executing the proposed algorithm with 32 processors.

(a) With the partitioning method 1

| Distribution | Standard deviation (average) |
|---|---|
| Uniform | 0.81(8.29) secs |
| Normal | 0.96(9.86) secs |
| Bubble | 1.32(10.25) secs |
| Uniform (w:h = 3:100) | 1.62(15.19) secs |

(b) With the partitioning method 2

| Distribution | Standard deviation (average) |
|---|---|
| Uniform | 1.13(7.51) secs |
| Normal | 1.38(7.92) secs |
| Bubble | 1.54(8.47) secs |
| Uniform (w:h = 3:100) | 1.69(16.95) secs |

The sequential merging step significantly limits the speed up. The algorithm is also released from the restriction that the first Delaunay triangle generated by each processor must exist entirely in its own subregion. All these improvements are possible by using projection-based partitioning methods.

We consider Delaunay triangulation of points in just $E^2$ space. However, it is remained as our future work to extend the proposed algorithm to $E^d$ spaces for $d > 2$.

## References

[1] J. O'Rourke, *Computational Geometry in C* (Cambridge University Press, 1994).
[2] F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction* (Springer-Verlag, New York, 1985).
[3] J. D. Boissonnat, Geometric Structures for Three Dimensional Shape Representation, *ACM Transactions on Graphics* **3** (1984) 266-286.
[4] J. C. Cavendish, Automatic Triangulation of Arbitrary Planar Domains for Finite Element Method, *Int. J. for Numerical Methods in Engineering* **8** (1974) 679–696.
[5] H. J. Choi, An Implementation of a Flight Path Visualization System using Optimization Algorithms of the Triangulated Irregular Network, MS. Thesis, POSTECH, 1996.
[6] S. G. Akl and K. A. Lynos, *Parallel Computational Geometry* (Prentice-Hall, 1993).
[7] S. Saxona, P. C. P. Bhatt and V. C. Prasad, Efficient VLSI Parallel Algorithm for Delaunay Triangulation on Orthogonal Tree Network in Two and Three Dimensions, *IEEE Trans. Comput.* **39** (1990) 400–404.
[8] P. Cignoni, C. Montani, R. Perego and R. Scopigno, Parallel 3D Delaunay Triangulation, in *Proc. Eurographics 93*, 1993.
[9] P. Cignoni, C. Montani and R. Scopigno, A Merge-First Divide & Conquer Algorithm for $E^d$ triangulations, TR 92-16, Istituto CNUCE-C.N.R., PIsa, Italy, 1992.
[10] Guy E. Blelloch, Gary L. Miller and Dafna Talmor, Developing a Practical Projection-Based Parallel Delaunay Algorithm, in *Proc. ACM Symposium on Computational Geometry*, May 1996.
[11] Sangyoon Lee, Chan-Ik Park and Chan-Mo Park, An Efficient Parallel Algorithm for Delaunay Triangulation on Distributed Memory Parallel Computers, in *Proc. The 1996 Int. Conf. PDPTA*, Aug. 1996.
[12] D. H. McLain, Two Dimensional Interpolation from Random Data, *The Comput. J.* **19** (1976) 178–181.
[13] V. Akman, W. R. Franklin, M. Kankanhalli and C. Narayanaswami, Geometric Computing and Uniform Grid Technique, *Computer-Aided Design* **21** (1989) 410–420.