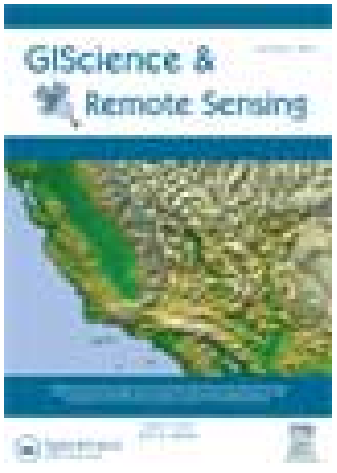
This article was downloaded by: [Nanjing University]
On: 29 September 2014, At: 04:10
Publisher: Taylor & Francis
Informa Ltd Registered in England and Wales Registered Number: 1072954 Registered office: Mortimer House, 37-41 Mortimer Street, London W1T 3JH, UK



GIScience & Remote Sensing
Publication details, including instructions for authors and subscription information:
http://www.tandfonline.com/loi/tgrs20

Novel parallel algorithm for constructing Delaunay triangulation based on a twofold-divide-and-conquer scheme

Wenzhou Wu[abc], Yikang Rui[b], Fenzhen Su[a], Liang Cheng[b] & Jiechen Wang[b]

[a] State Key Laboratory of Resources and Environmental Information System, Institute of Geographic Sciences and Natural Resources Research, Chinese Academy of Sciences, Beijing, 100101, China

[b] Jiangsu Provincial Key Laboratory of Geographic Information Science and Technology, School of Geographic and Oceanographic Sciences, Nanjing University, Nanjing, 210093, China

[c] University of Chinese Academy of Sciences, Beijing, 100049, China

Published online: 18 Aug 2014.

To cite this article: Wenzhou Wu, Yikang Rui, Fenzhen Su, Liang Cheng & Jiechen Wang (2014): Novel parallel algorithm for constructing Delaunay triangulation based on a twofold-divide-and-conquer scheme, GIScience & Remote Sensing

To link to this article: http://dx.doi.org/10.1080/15481603.2014.946666

PLEASE SCROLL DOWN FOR ARTICLE

Taylor & Francis makes every effort to ensure the accuracy of all the information (the "Content") contained in the publications on our platform. However, Taylor & Francis, our agents, and our licensors make no representations or warranties whatsoever as to the accuracy, completeness, or suitability for any purpose of the Content. Any opinions and views expressed in this publication are the opinions and views of the authors, and are not the views of or endorsed by Taylor & Francis. The accuracy of the Content should not be relied upon and should be independently verified with primary sources of information. Taylor and Francis shall not be liable for any losses, actions, claims, proceedings, demands, costs, expenses, damages, and other liabilities whatsoever or howsoever caused arising directly or indirectly in connection with, in relation to or arising out of the use of the Content.

This article may be used for research, teaching, and private study purposes. Any substantial or systematic reproduction, redistribution, reselling, loan, sub-licensing, systematic supply, or distribution in any form to anyone is expressly forbidden. Terms & Conditions of access and use can be found at http://www.tandfonline.com/page/terms-and-conditions

Downloaded by [Nanjing University] at 04:10 29 September 2014

GIScience & Remote Sensing, 2014
http://dx.doi.org/10.1080/15481603.2014.946666

Taylor & Francis
Taylor & Francis Group

Novel parallel algorithm for constructing Delaunay triangulation based on a twofold-divide-and-conquer scheme

Wenzhou Wu[a,b,c], Yikang Rui[b]*, Fenzhen Su[a], Liang Cheng[b] and Jiechen Wang[b]

[a]*State Key Laboratory of Resources and Environmental Information System, Institute of Geographic Sciences and Natural Resources Research, Chinese Academy of Sciences, Beijing, 100101, China;* [b]*Jiangsu Provincial Key Laboratory of Geographic Information Science and Technology, School of Geographic and Oceanographic Sciences, Nanjing University, Nanjing, 210093, China;* [c]*University of Chinese Academy of Sciences, Beijing, 100049, China*

(*Received 26 February 2014; accepted 9 July 2014*)

To increase the efficiency when processing large data sets, a novel parallel algorithm is proposed for constructing the Delaunay triangulation of a planar point set based on a twofold-divide-and-conquer scheme. This algorithm automatically divides the planar point set into several non-overlapping subsets along the *x*-axis and *y*-axis directions alternately, according to the number of points and their spatial distribution. Next, the Guibas–Stolfi divide-and-conquer algorithm is applied to construct Delaunay sub-triangulations in each subset. Finally, the sub-triangulations are merged based on the binary tree. All three sequential steps are processed using multitasking parallel technology. Our results show that the proposed parallel algorithm is efficient for constructing the Delaunay triangulation with a good speed-up.

Keywords: Delaunay triangulation; twofold-divide-and-conquer scheme; adaptive subdivision; parallel computing

Downloaded by [Nanjing University] at 04:10 29 September 2014

1. Introduction

The Delaunay triangulation of a planar point set is a triangulation such that the circumcircle of any triangle contains no point in its interior. As an optimal triangulation, the Delaunay triangulation and its dual (the Voronoi diagram) have been used widely in terrain modelling, 3D scene visualization, computer graphics, pattern recognition, finite element analysis and cartographic generalization (Kolingerová and Kohout 2002; An and Trang 2012). However, identifying the best algorithm for constructing the Delaunay triangulation rapidly and efficiently has always attracted much interest in these application areas. Many classic algorithms are available such as incremental algorithms, divide-and-conquer algorithms and sweepline algorithms (Su and Drysdale 1997). The incremental algorithms add one point into the existing triangulation one by one and update the triangulation in a timely manner (Lawson 1977; Watson 1981). This implementation is relatively simple, but it runs slowly and the time complexity in the worst case is $O(n^2)$. Guibas, Knuth, and Sharir (1992) improved the algorithm by using a random insertion method, which reduced the time complexity to $O(n\log n)$. The divide-and-conquer algorithm proposed by Shamos and Hoey (1975) was applied to the construction of a Voronoi diagram with a time complexity of $O(n\log n)$. This algorithm was later used to construct the Delaunay triangulation by Lee and Schachter (1980). Guibas and Stolfi (1985)

*Corresponding author. Email: ruiyikang@gmail.com

© 2014 Taylor & Francis

simplified the process to reduce the time complexity to $O(n\log n)$. Dwyer (1987) modified the algorithm further for uniformly distributed data within a square, and the time complexity was reduced to $O(n\log\log n)$, although the maximum number of points in the experimental data set was 65,536. The sweepline algorithm proposed by Fortune (1987) was designed to construct a Voronoi diagram and its dual graph (Delaunay triangulation) for a planar point set. This algorithm uses a priority queue to store point events and circle events before tracing the status of the sweeplines based on these two events; thus, the data structure is complex and the time complexity in the worst case is $O(n\log n)$.

In many cases, the volume of data that need to be processed has grown, for example LiDAR point cloud data for various applications (Petroselli 2012; Chen and Zhu 2013); thus, the significant increase in the computational time required for processing has become one of major problems in existing sequential algorithms. Meanwhile, the frequency of a single processor is also close to the limit; thus, mainstream CPU manufacturers have started to develop multicore processors such as the very popular dual-core, quad-core and eight-core processors, which provide parallel programming environments.

Parallel construction of the Delaunay triangulation has been studied by many researchers in recent years, and many interesting strategies have been proposed (Batista et al. 2010). Teng et al. (1993) designed an algorithm for data-parallel architectures with a bucketing technique, which was up to five times slower with non-uniform data sets compared with uniform data sets (on a 32-processor CM-5). Puppo et al. (1994) designed a parallel algorithm for computing a triangulated irregular network on massive distributed machines. However, the resulting triangulation did not necessarily satisfy the empty circle property; thus, the standard edge flip procedure (Lawson 1977) was applied concurrently to all non-Delaunay edges. This algorithm achieved an 80-fold speed-up when triangulating 16-K points on a Connection Machine CM-2 with 16-K processors. A spatial decomposition approach was also proposed for performing subdomain and interface triangulation based on incremental construction (Cignoni et al. 1995). Lee, Park, and Park (2001) improved the Delaunay triangulation algorithm by exploiting a projection-based partitioning scheme (Blelloch et al. 1999) and by eliminating the merging phase. The algorithm was implemented on an INMOS TRAM network of 32 T800 processors, and it achieved a speed-up of around 6.5 on 8 processors using 10,000 randomly distributed points. However, the best partitioning method required 75% of the total elapsed time. To accelerate merging, a parallel divide-and-conquer scheme for 2D Delaunay triangulation was proposed by Chen, Chuang, and Wu (2006), who introduced an 'affected zone' to combine sub-Delaunay triangulations. The processing of 1–16 million points using eight processors resulted in speed-ups of 5.6 to 6.0. Recently, Wu, Guan, and Gong (2011) proposed a parallel Delaunay triangulation algorithm for LiDAR points, which distributes the triangulation procedure in a KD-tree based on a dynamic scheduling strategy thereby achieving good load balancing on different processors.

Existing parallel algorithms improve the wall time of the Delaunay triangulation, but the following three major problems still persist: (1) most of the existing algorithms are based on divide-and-conquer schemes and they use buckets or stripes division techniques. The best method for partitioning a point set into subsets with appropriate sizes has not been determined; thus, the efficiency of parallel computing is not fully exploited. (2) The time complexity of the algorithms is affected by the distribution of the data in space. If the data are distributed unevenly, the improvements achieved by using parallel technology are not significant. (3) Existing algorithms usually extract the affected area on the triangulation boundary before merging the sub-triangulations. This approach requires additional

Downloaded by [Nanjing University] at 04:10 29 September 2014

time when merging two Delaunay triangulations in subsets, and it affects the overall efficiency of parallel computing.

In this study, we propose a novel parallel algorithm for constructing the Delaunay triangulation of a planar point set using a twofold-divide-and-conquer approach. This algorithm employs a parallel adaptive subdivision method to automatically divide a planar point set into several non-overlapping subsets along the *x*-axis and *y*-axis directions alternately. All of the subsets with approximately equal number of points are distributed optimally to multicore processors. The Guibas–Stolfi divide-and-conquer algorithm is applied to each subset because it has a good time complexity. The topological relations of the Delaunay triangulations generated in the subsets facilitate further merging. After parallel computing, a binary tree is used to continue the merging of two horizontally adjacent sub-triangulations from the bottom to the top in the *x*-axis direction, or the merging of two vertically adjacent sub-triangulations from the right to the left in the *y*-axis direction, until the final overall Delaunay triangulation is generated.

Downloaded by [Nanjing University] at 04:10 29 September 2014

2. Parallel adaptive subdivision method

The strategy used for partitioning a planar point set is the most important factor that affects the efficiency when employing a parallel algorithm to construct a Delaunay triangulation. After the subsets have undergone several rounds of subdivision to obtain approximately the same number of points, a good level of load balancing can be achieved where tasks are distributed on different processors. Wang et al. (2014) proposed an adaptive subdivision method for a planar point set to resolve the problems of uneven subdivision in quad tree, rectangular and other polygonal partitions. First, this method establishes an auxiliary grid that covers the planar point set. The points are indexed by each cell in the auxiliary grid to facilitate subsequent rapid searches during data reading. Next, the point set is divided recursively into two groups until the recursion depth reaches a preset number of subdivisions. The two newly generated subsets must contain a similar number of points, and they are stored in a binary tree structure. Figure 2 shows the binary tree after three adaptive subdivisions of the grid illustrated in Figure 1.

During testing, we found that the efficiency of the subdivision scheme was a serious issue when the data volume reached 10 million points or when the cells in the auxiliary grid became too small. Therefore, we improved the method mentioned above by introducing a parallel adaptive subdivision scheme. As shown in Figure 2, each leaf node is assigned to one processor after one round of subdivision, and parallel subdivision is then performed for the leaf nodes in all processors. The process is recursively done until a designed number of subdivision level is reached. In this implementation, we use the binary tree level-order traversal to partition the point set. However, due to the constraints of the number of processors, the number of leaf nodes generally does not match the number of processors. If the number of leaf nodes is greater than the number of processors, we allocate one task to each processor and monitor all of the working processors. Next, we repeatedly assign one task to the processor that has just finished its task.

Our method utilizes coordinate arrays to store the *x*-, *y*-coordinates of planar points, and the auxiliary grid records the number of points within each cell as well as the corresponding serial numbers in the coordinate arrays. After the subdivision, the subsets are stored in a binary tree structure. Each subset records information that includes the

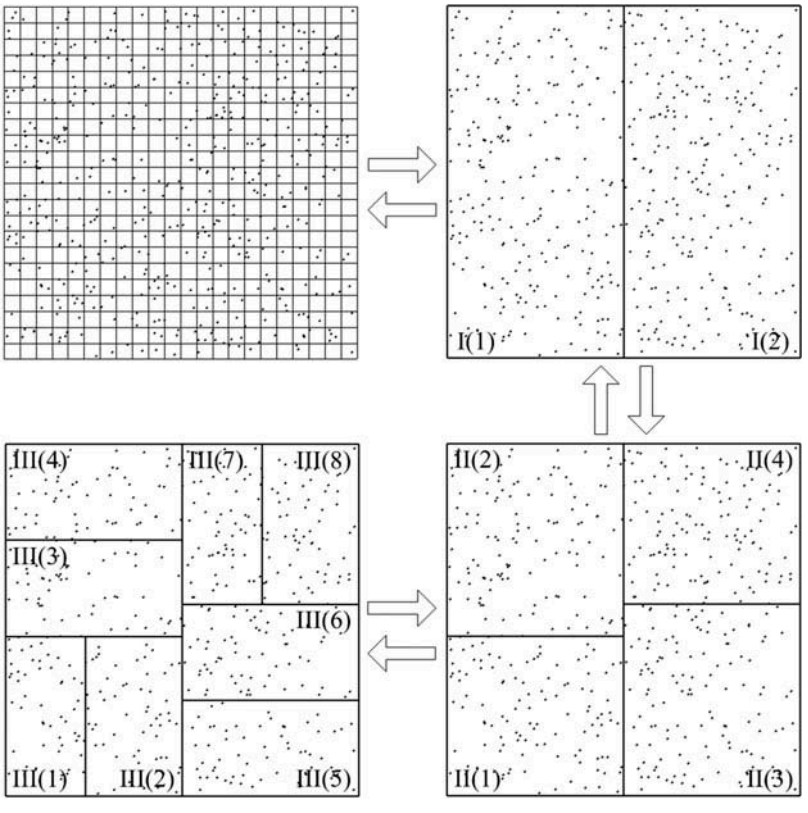


Figure 1. Adaptive subdivision for a planar point set.

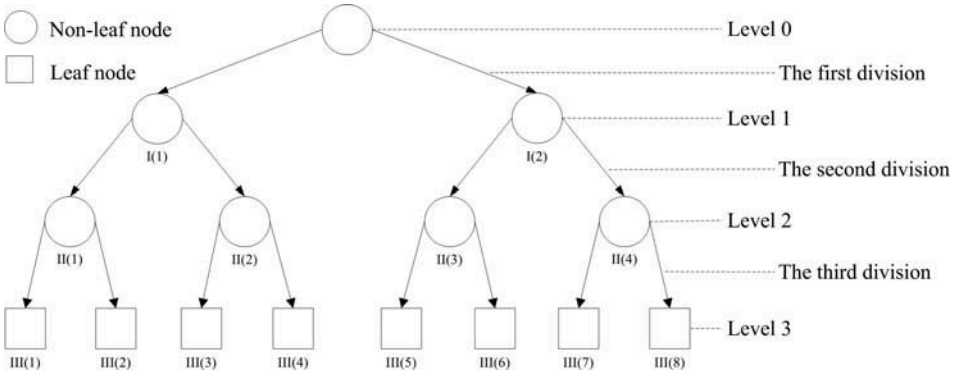


Figure 2. Binary tree structure for the adaptive subdivision.

numbers of the first and last row and column in the auxiliary grid, the number of points within the subset and the level number in the binary tree. The data structure in the C language is described as follows:

Downloaded by [Nanjing University] at 04:10 29 September 2014

```
struct Point2d{                                  //Point structure
   double x, y;};                                //Coordinates
struct RasterField {                             //Auxiliary grid structure
   int nNum;                                     //Record number of points within the raster cell
int *PointId; };                                 //Record serial number of points in coordinate arrays
struct BinaryTree{                               //Subset structure
   int nStartLine, nEndLine;                     //Record start and end row number
   int nStartCol, nEndCol;                       //Record start and end column number
   int nLevel;                                   //Record level number in the binary tree
   int nNum;                                     //Record the number of points within the subset
   Point2d *PointData;                           //Record the coordinates of points within the subset
   BinaryTree *LeftChild, *RightChild;           //Record left and right nodes after partition
   BinaryTree *Parent;                           //Record parent node of the subset
   ExtremeEdge extremeEdge; };                   //Record extreme directed edges of the Delaunay
                                                   triangulation generated by points within the subset
```

It is unnecessary to store the coordinates of the points in each subset after one subdivision, and only the number of points in each subset is recorded. After the parallel adaptive subdivision process is finished completely, the coordinates of the points in the leaf nodes are recorded, and each leaf node is employed as the basic unit to construct the Delaunay triangulation in the next step.

3. Twofold-divide-and-conquer Delaunay triangulation algorithm

First, our proposed algorithm utilizes a parallel adaptive subdivision method to automatically partition a planar points set alternately along the *x*-axis and *y*-axis directions, that is, the first subdivision. Next, the Guibas–Stolfi divide-and-conquer algorithm is used to partition the points in each subset in the *x*-axis direction, that is, the second subdivision. The Delaunay triangulations are then constructed in all of the subsets, and they are finally merged according to the inverse order of the first subdivision. The overall process is shown in Figure 3. The empty circumcircle criterion is tested when two subsets are being merged. After all the subsets have been merged, it is not necessary to test the empty circumcircle criterion for the overall triangulation. In addition, the use of parallel computing by our algorithm improves the efficiency. The main steps are described as follows:

(1) Use a parallel adaptive subdivision method to roughly divide a planar point set into several subsets.
(2) Assign each subset to one processor and use the divide-and-conquer algorithm to construct the Delaunay triangulation for each subset.
(3) Collect the Delaunay triangulations from all the processors and merge two Delaunay triangulations in adjacent subsets horizontally or vertically each time according to the binary tree structure.

3.1. Guibas–Stolfi divide-and-conquer algorithm

Many algorithms can be used to construct a Delaunay triangulation for each subset, but most do not establish the topological relations between triangles, which makes it more complex and difficult to merge the Delaunay triangulations. Guibas and Stolfi (1985) proposed a divide-and-conquer algorithm with a Quad-Edge data structure to support topological operations. Given that the Guibas–Stolfi divide-and-conquer algorithm is

Downloaded by [Nanjing University] at 04:10 29 September 2014

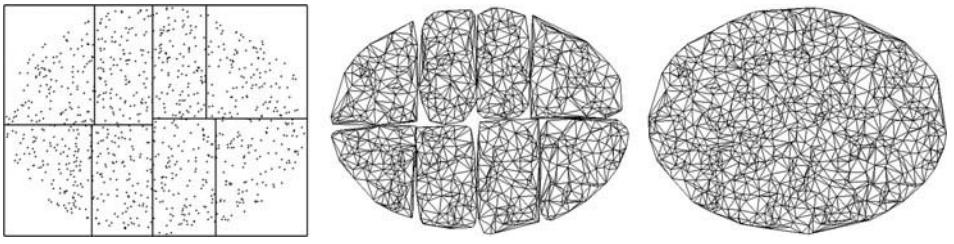


Figure 3. Overview of the Delaunay triangulation construction based on the twofold-divide-and-conquer scheme.

efficient (the time complexity in the worst case is $O(n\log n)$) and that the Quad-Edge data structure can readily maintain the topological relations, which are beneficial for merging the Delaunay triangulations in all the subsets at once, we selected the Guibas–Stolfi divide-and-conquer algorithm to construct the Delaunay triangulation. The basic steps are as follows:

(1) Set the point set in ascending lexicographic order according to the abscissa supplemented by the ordinate, that is, $(x_i, y_i) \leq (x_{i+1}, y_{i+1})$. Remove any duplicate points.
(2) Use a vertical line to recursively divide the point set into left and right subsets (L and R) which have approximately equal numbers of points, until the number of points in a subset is less than four.
(3) For two or three points in a subset, simply connect them as a segment or triangle.
(4) Merge two subsets L and R in a stepwise manner to form the final Delaunay triangulation.

In the Guibas–Stolfi divide-and-conquer algorithm, the most important step is to merge the generated left and right triangulations. As shown in Figure 4, the dividing line at the point where two Delaunay triangulations are merged is crossed by these newly added edges. Two adjacent newly added edges share the same vertex; thus, a series of successive 'cross-edges' is formed in the vertical direction, which makes the merging procedure progress in a bottom-up order. After testing the empty circumcircle criterion, an

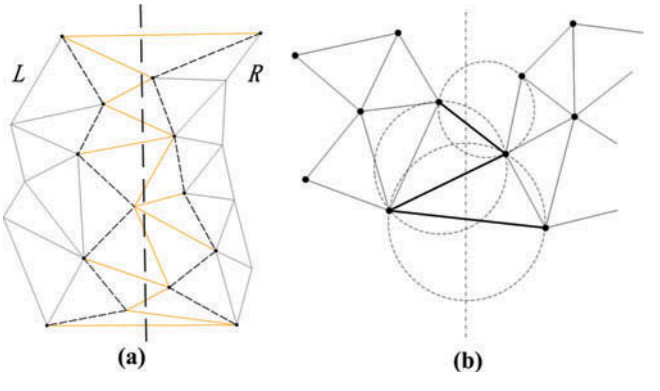


Figure 4. Cross-edges and rising circles.

Downloaded by [Nanjing University] at 04:10 29 September 2014

arrangement of rising circles is formed along the vertical dividing line, as shown in Figure 4. The merging process is finished when all of the points in the left and right triangulations are below a newly added edge. The whole process is repeated until all of the subsets have been merged and the final Delaunay triangulation is obtained.

3.2. *Data structure*

The data structure of the Guibas–Stolfi divide-and-conquer algorithm is helpful for merging the left and right Delaunay triangulations, but it cannot be used to merge up and down sub-triangulations. Thus, we improved the data structure by adding the two extreme directed edges of the topmost vertex and bottommost vertex (topEdge and bottomEdge, respectively), as shown in Figure 5. In addition, there is no need to convert between the Delaunay triangulation and the Voronoi diagram; thus, we saved memory space by adopting a half-edge data structure (Brönnimann 2001) instead of the Quad-Edge structure. The half-edge data structure is also suited to topological operations that are supported by a Quad-Edge structure. Figure 6 shows the differences between the data structures of the directed edges and Quad-Edge. The modified data structure is described in the C language as follows.

```
struct Edge {                    //Directed edge structure
   Point2d* org;                 //Coordinates of start point
Edge* twin;                      //Twin directed edge
Edge* next;                      //Next directed edge
   Edge* prev; };                //Previous directed edge
structExtremeEdge {              //Edge structure of extreme points in the convex hull
   Edge* leftEdge;               //Leftmost directed edge
Edge* rightEdge;                 //Rightmost directed edge
Edge* bottomEdge;                //Bottommost directed edge
Edge* topEdge; };                //Topmost directed edge
```

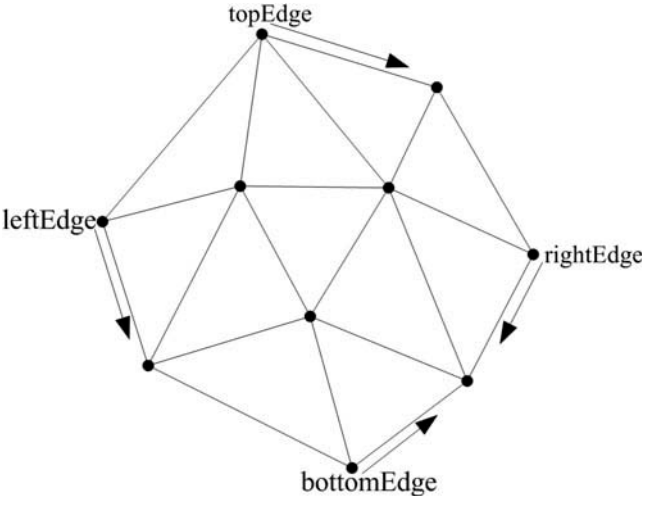


Figure 5. Directed edges of the extreme vertices in the convex hull of the Delaunay triangulation.

Downloaded by [Nanjing University] at 04:10 29 September 2014

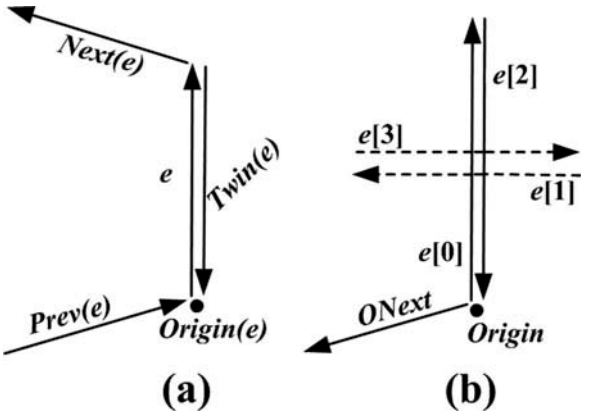


Figure 6. Data structures of directed edges (a) and Quad-Edge (b).

When a binary tree is used to store subsets after partitioning a planar point set, each node in the binary tree saves the information for the extreme directed edges in the convex hull of the Delaunay triangulation generated in the subsets, as shown in Section 2. After merging two sub-triangulations that represent the left and right nodes in a binary tree, the results obtained are extreme directed edges in the convex hull of the newly generated Delaunay triangulation, which point to the extreme directed edges in the convex hull of their parent node. This structure can be helpful for searching for any edge of the Delaunay triangulation in the subsets, but also for merging the Delaunay triangulations.

3.3. *Merge two adjacent Delaunay triangulations*

Before merging two adjacent Delaunay triangulations, we need to determine the positional relationship between them. If they have a left–right relationship, we search for their bottom common tangent. Otherwise, the right common tangent is required. In the following paragraph, we describe the merging of the left and right Delaunay triangulations as an example to illustrate our method in detail.

First, we find the bottom common tangent of the two convex hulls of the left and right Delaunay triangulations. As shown in Figure 7a, if we assume that the left and right convex hulls are *L* and *R, ldi* is the rightmost directed edge of the left convex hull, and *rdi* is the leftmost directed edge of the right convex hull. In Figure 7b, the start point of edge *rdi* is set as a reference point. If the reference point is on the left of the edge *ldi*, then set *ldi* as its subsequent chain. Repeat this process until the reference point is on the right of the edge *ldi*. Next, the start point of edge *ldi* is the left point of the bottom common tangent (red circle in Figure 7b). Similarly, the right point of the bottom common tangent is shown in Figure 7c. We set the start point of edge *ldi* as a reference point. Assess the position of the reference point *ldi* repeatedly until it is on the left of the directed edge *rdi*. If *ldi* is on the right of *rdi*, set *rdi* as the next edge in the counterclockwise direction of the convex hull by introducing the precursor chain of its twin counterpart (dotted arrows in Figure 7c). Finally, connect the start points of the current *ldi* and *rdi* as the bottom common tangent of the left and right convex hulls (Figure 7d).

Start with the bottom common tangent as the baseline (*basel*) and select a 'third point' in the convex hulls of the two sub-triangulations to form a Delaunay triangle by testing the empty circumcircle criterion. Add the triangle to the Delaunay triangulation and reset the baseline as the new link between the two sub-triangulations (Figure 8b). When the baseline reaches the top

Downloaded by [Nanjing University] at 04:10 29 September 2014

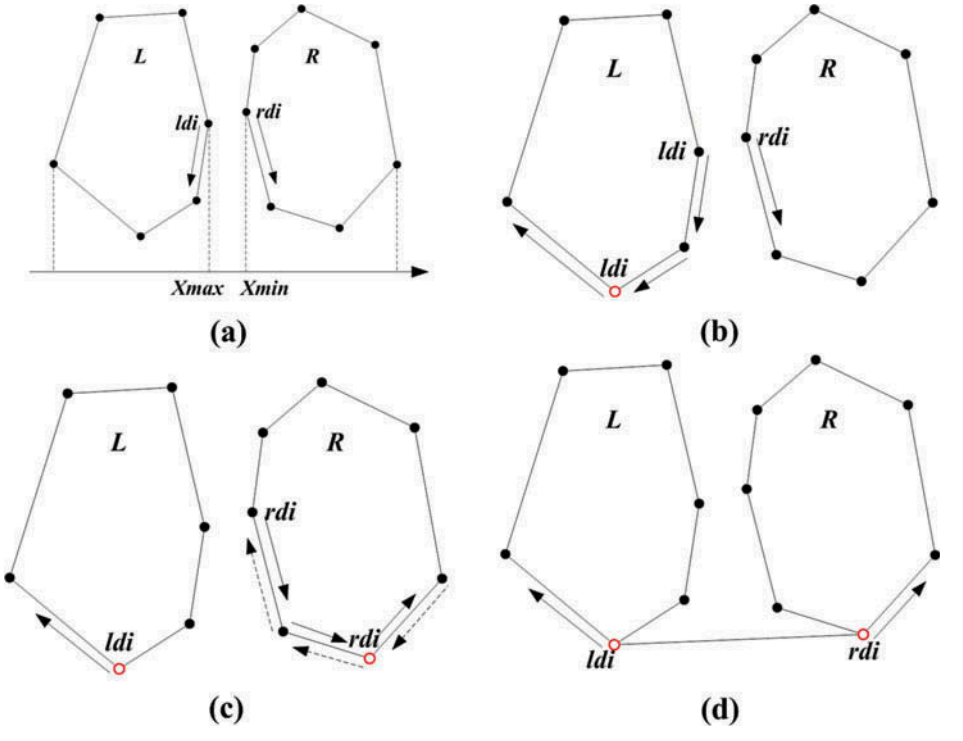


Figure 7. Search for the bottom common tangent in the left and right convex hulls. (a) The rightmost and leftmost directed edges. Search for the left point (b) and the right point (c) of the final bottom common tangent (d).

common tangent, the merging process is complete. During the search for the 'third point', *lcand* is selected from the edges in the left triangulation that interact with left endpoint of *basel*, meanwhile *rcand* is selected from the edges in the right triangulation that interact with right endpoint of *basel*. The other endpoints of *lcand* and *rcand* become the candidates for the 'third point' (Figure 8c). For each left candidate in *lcand*, if the triangle formed by *lcand* and *basel* cannot satisfy the empty circumcircle criterion in the left triangulation, then delete this *lcand* (Figure 8d). A similar process is performed for the right candidates in Figure 8e. Finally, select one point from the remaining left and right candidates that satisfies the empty circumcircle criterion in the overall triangulation and connect it with *basel* to form the new baseline *basel*, as shown in Figure 8f.

It is easy to complete the merging process using the topological operation from the Guibas–Stolfi Quad-Edge data structure. After merging two Delaunay triangulations, it is necessary to update the extreme directed edges in the convex hull of the newly generated Delaunay triangulation to facilitate the search for common tangents when performing the merging process in the upper level.

The first process used to search for the 'third point' in the convex hulls of two adjacent sub-triangulations can be completed in $O(n)$ time. As the baseline moves forward, edges will be deleted from the sub-triangulations or added to the new triangulation. In the worst case, all the existing edges are deleted and new edges are added. The relationship between the number of edges and the number of points in a triangulation is

Downloaded by [Nanjing University] at 04:10 29 September 2014

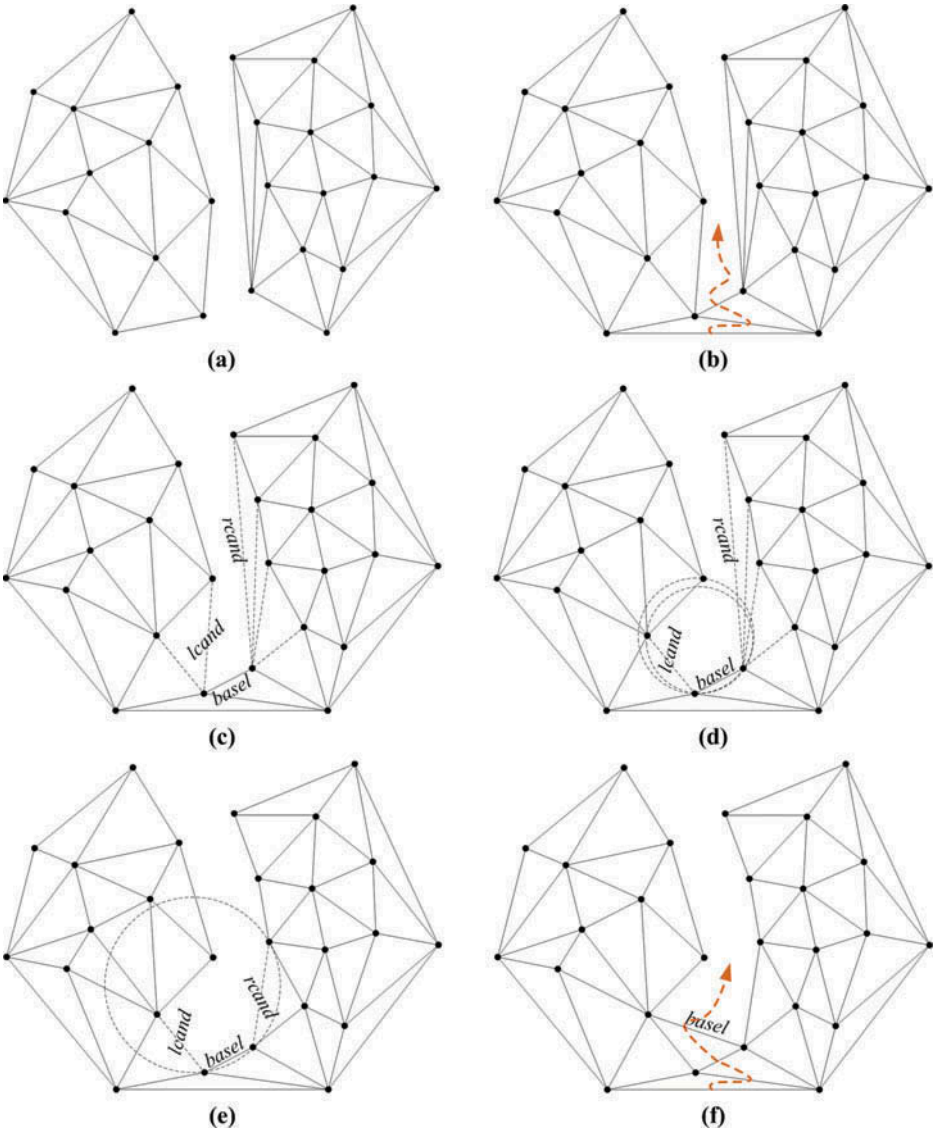


Figure 8. (a) Merging two Delaunay triangulations. (b) The baseline moves upwards. (c) Candidates for the 'third point'. Test the empty circumcircle criterion for the *lcand* (d) and *rcand* (e). (f) Find the 'third point' and generate a new baseline.

linear; thus, the complexity of deleting and adding edges is $O(n)$. The total time required for the overall merging process is $O(n)$.

3.4. Merging the Delaunay triangulations based on a binary tree

After using the Guibas–Stolfi divide-and-conquer algorithm to construct the Delaunay triangulation in each subset, the best method for merging these sub-triangulations is a critical issue. Two subsets are stored by the left and right child nodes of the same parent

Downloaded by [Nanjing University] at 04:10 29 September 2014

node in a binary tree and they must be adjacent. The merging algorithm described in the previous section can be used for this process.

We merge the Delaunay triangulations in the subsets according to the reverse order of the parallel adaptive subdivision process for the planar point set. The algorithm starts with the leaf nodes of the binary tree. The Delaunay triangulations of two leaf nodes with the same parent node are merged. The process continues from the bottom to the top until it reaches the root node of the binary tree to form the final Delaunay triangulation.

```
Process GapMerge(pNode)
Input: root node of a binary tree
Output: leaf nodes of the binary tree
  If pNode is a leaf node then
    return pNode;
  Else
    pLeftNode←GapMerge (left child node of pNode);
    pRightNode←GapMerge (right child node of pNode);
    pParent←parent node of pLeftNode;
    If pLeftNode and pRightNode showing a left-right relationship then
      extremeEdge of pParent← Merge(pLeftNode, pRightNode, true);
    Else
      extremeEdge of pParent← Merge(pLeftNode, pRightNode, false);
    End if
  return pParent;
  End if
```

Downloaded by [Nanjing University] at 04:10 29 September 2014

4. Parallel computing

The proposed parallel algorithm used to construct the Delaunay triangulation of a planar point set involves three sequential processes: parallel adaptive subdivision, Delaunay triangulation construction in each subset and merging the Delaunay triangulations in subsets. The construction of the Delaunay triangulation in each subset is the most time-consuming process. However, it is the easiest component for parallel processing. Therefore, we use a parallel computing method to construct the Delaunay sub-triangulations in different subsets. A master–slave model is applied, that is, one processor is selected as the master node, whereas the other processors have the roles of slaves. The master node is responsible for adaptive subdivision of the point set, data distribution to the slave nodes and merging the sub-triangulations. The slave nodes only need to construct the Delaunay triangulations of the point sets they received. However, if the number of parallel tasks is greater than the number of slave nodes, one slave node may receive several tasks. The overall process is described as follows:

(1) The master node partitions a planar point set according to a given number of subdivisions. Assume that the number of subsets is N.
(2) If there are M slave nodes and $N \leq M$, each planar point subset is assigned to one slave node. Otherwise, each slave node receives one point subset and the master

node monitors all the working processors. Next, repeatedly assign a task to the slave node that finishes its task first.

(3) The master node monitors the implementation of each slave node until all of the subsets are processed.

(4) The master node merges the N Delaunay sub-triangulations based on a binary tree structure and generates the final Delaunay triangulation.

5. Experimental test and analysis

To investigate the efficiency and stability of the proposed algorithm, we tested the algorithm using non-uniformly distributed data. The test environment was as follows: Intel (R) Xeon (R), CPU 2.0 GHz, RAM 32.0 GB and Windows Server 2008. The algorithm was programmed using Microsoft Visual C++ 6.0. The maximum number of usable processors was 64. The experimental discrete data set comprised 35,602 contours in the United States of America (including 2,335,537 points). The initial numbers of rows and columns in the auxiliary grid were 1000 × 1000 and the number of levels was 6. Figure 9 shows the construction of the Delaunay triangulation using the American contour data. We then resampled the original data and set different numbers of points to test, that is, 0.1, 0.5, 1, 2.5, 5, 7.5 and 10 million.

The numbers of rows and columns in the auxiliary grid affect the adaptive subdivision results, where the subdivision granularity is finer with larger numbers of rows and columns. Alternatively, the subdivision granularity is rougher with smaller numbers. To determine the effects of the numbers of rows and columns on every process in our algorithm, we compared two experiments. Table 1 shows the elapsed time during the implementation with the following preset values: initial numbers of rows and columns in the auxiliary grid were 1000 × 1000, and the number of subsets was 64. Note that all values presented in the tables were averaged over 10 independent runs. Figure 10 shows the corresponding graph. Table 2 shows the results obtained with 500 × 500 auxiliary grid and 64 subsets.

As shown in Figure 10, subdivision using a single-core processor improved the construction efficiency slightly compared with the Guibas–Stolfi divide-and-conquer algorithm, but parallel computing with multicore processors reduced the construction time significantly. Tables 1 and 2 show clearly that the parallel adaptive subdivision and parallel merging of Delaunay sub-triangulations were less time consuming, whereas the Delaunay triangulation construction in the subsets required the most time. Let n be the number of points. After d adaptive subdivisions, $k = 2^d$ subsets are obtained and each subset contains approximately $m = n/k$ points. In the first step of adaptive subdivision, the process of traversing the points in each subset is related to the size of the auxiliary grid (s rows and s columns); thus, it requires in $O(s^2)$ time for each subdivision and $O(d \times s^2)$ time after d subdivisions. To assign the points to each subset, we need $O(n)$ time. Then, the sequential time complexity of the first step is $O(d \times s^2) + O(n)$. In the second step when constructing the Delaunay triangulations in k subsets, the algorithm requires $O(k \times (m \log m)) = O(n \times \log(n/k))$. The third step of merging the sub-triangulations requires $O(nn)$ time to merge two adjacent sub-triangulations at one level, where nn indicates the number of 'third points' searched in Figure 9, and there are d levels in a binary tree, it takes $O(d \times nn)$ time to finish the third step. When performing all three steps in a single-core processor computation, the algorithm has a time complexity of $O(d \times s^2) + O(n) + O(n \times \log(n/k)) + O(d \times nn)$. Because nn is far smaller than n, the time complexity is approximately $O(d \times s^2 + n \times \log(n/k))$. If we use parallel technology to partition the original point

Downloaded by [Nanjing University] at 04:10 29 September 2014

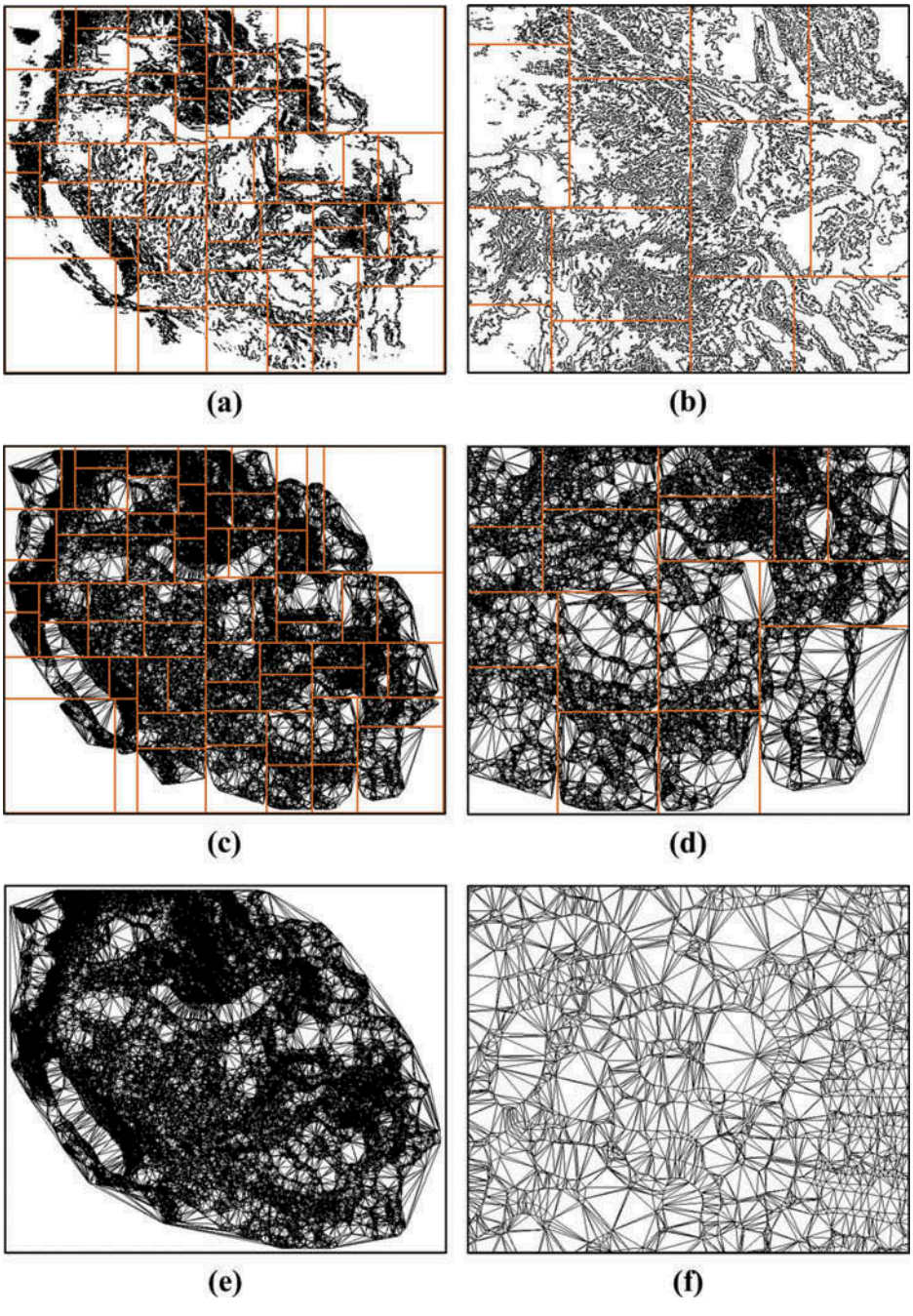


Figure 9. Implementation of the algorithm.

set, construct the Delaunay triangulations in subsets and merge triangulations, where the number of processors is p ($k > p$), the time complexity of each step decreased to $O((s^2)\times(2-1/2^{(d-1)}) + n/k)$, $O((n/p)\times\log(n/k))$ and $O(nn\times(2-1/2^{(d-1)}))$, respectively, that is, our parallel algorithm approximately requires $O((s^2) + (n/p)\times\log(n/k))$ time.

Downloaded by [Nanjing University] at 04:10 29 September 2014

Table 1. Test results obtained using the algorithm (1000 × 1000 auxiliary grid).

Number of points (million)	Elapsed time (seconds)					
	Parallel adaptive subdivision (64 subsets)	Delaunay triangulation construction in subsets	Merging Delaunay triangulations	*T*1	*T*2	*T*3
10	0.520	1.742	0.156	2.418	56.410	57.658
7.5	0.489	1.388	0.130	2.007	42.994	44.366
5	0.421	0.951	0.098	1.471	29.630	31.372
2.5	0.218	0.546	0.078	0.842	15.148	16.364
1	0.182	0.255	0.052	0.489	6.162	7.472
0.5	0.135	0.156	0.031	0.322	3.136	4.056
0.1	0.078	0.078	0.016	0.172	0.749	0.952

Notes: *T*1 = Elapsed time with the twofold-divide-and-conquer parallel algorithm using 64 processors.
*T*2 = Elapsed time with the twofold-divide-and-conquer algorithm.
*T*3 = Elapsed time with the Guibas–Stolfi divide-and-conquer algorithm.

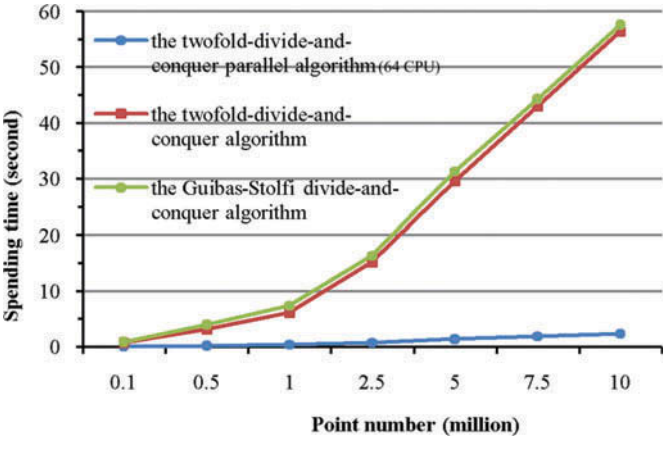


Figure 10. Elapsed time versus different numbers of points.

Table 2. Test results obtained using the algorithm (500 × 500 auxiliary grid).

Number of points (million)	Elapsed time (seconds)					
	Parallel adaptive subdivision (64 subsets)	Delaunay triangulation construction in subsets	Merging Delaunay triangulations	*T*1	*T*2	*T*3
10	0.473	1.768	0.171	2.412	55.646	57.658
7.5	0.370	1.377	0.125	1.872	42.264	44.366
5	0.275	0.936	0.093	1.305	28.933	31.372
2.5	0.166	0.551	0.083	0.801	14.747	16.364
1	0.099	0.260	0.047	0.406	6.142	7.472
0.5	0.078	0.161	0.041	0.280	3.089	4.056
0.1	0.042	0.078	0.015	0.135	0.645	0.952

Notes: *T*1 = Elapsed time with the twofold-divide-and-conquer parallel algorithm using 64 processors.
*T*2 = Elapsed time with the twofold-divide-and-conquer algorithm.
*T*3 = Elapsed time with the Guibas–Stolfi divide-and-conquer algorithm.

Downloaded by [Nanjing University] at 04:10 29 September 2014

Table 3. Elapsed time required for parallel Delaunay triangulation construction with different point set sizes and numbers of processors.

Number of points (million)	Elapsed time for parallel Delaunay triangulation construction with different numbers of processors (seconds)						
	1	2	4	8	16	32	64
10	57.658	30.930	14.810	8.124	5.913	3.120	2.418
7.5	44.366	23.712	11.903	6.214	4.571	2.485	2.007
5	31.372	16.547	8.260	4.560	3.291	1.934	1.471
2.5	16.364	9.001	4.259	2.688	1.731	1.123	0.842
1	7.472	3.962	1.825	1.357	0.910	0.582	0.489
0.5	4.056	2.559	1.149	0.661	0.510	0.374	0.322
0.1	0.952	0.785	0.286	0.182	0.156	0.130	0.172

Downloaded by [Nanjing University] at 04:10 29 September 2014

The time complexity analysis shows that the adaptive subdivision and merging processes both are less time consuming than the parallel construction of the Delaunay triangulations in the subsets. Furthermore, the effect of the size of the auxiliary grid (s^2) on the overall time complexity is trivial compared with the parallel construction in the second step. Thus, the efficiency of the algorithm improves slightly when varying the numbers of rows and columns in the auxiliary grid. The test results shown in Tables 1 and 2 are consistent with our analysis.

During multiprocessor parallel computing, the numbers of subsets and processors were the same in our experiments. Table 3 shows the elapsed time for the parallel construction of the Delaunay triangulations with different sized point sets and numbers of processors. Figure 11 is the corresponding graph, which shows that the efficiency of the algorithm decreased significantly as the number of processors increased.

To assess the parallel efficiency of the proposed algorithm, we fixed the size of the point set and calculated the relative speed-up and efficiency with different numbers of processors. In general, the relative speed-up, *Sp*, is the main indicator that reflects the parallel efficiency: $Sp = Ts/Tp$, where *Ts* is the sequential execution time in parallel computing, and *Tp* is a parallel execution time with *p* processors. The efficiency *Ep*

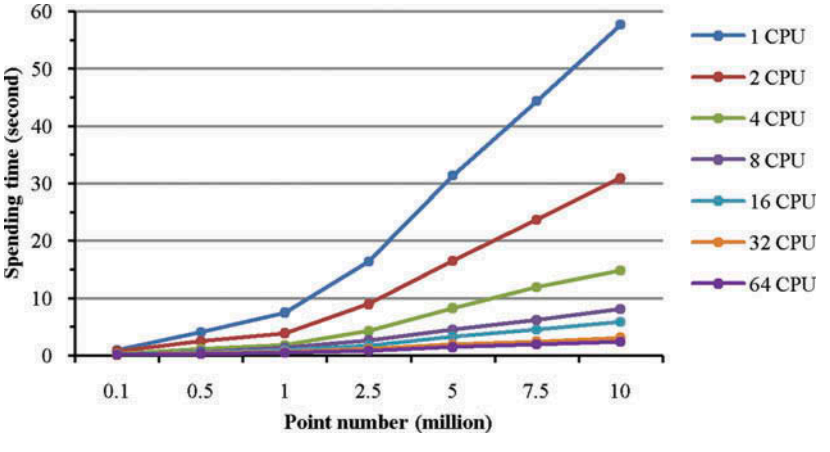


Figure 11. Elapsed time required for the construction with different numbers of processors.

Table 4. Relative speed-up and efficiency of the proposed algorithm.

Number of points (million)	Relative speed-up (S) and efficiency (E) with different number of processors													
	1		2		4		8		16		32		64	
	S_1	E_1	S_2	E_2	S_4	E_4	S_8	E_8	S_{16}	E_{16}	S_{32}	E_{32}	S_{64}	E_{64}
10	1.00	100	1.978	99	3.962	99	7.534	94	9.952	62	18.737	59	23.329	36
7.5	1.00	100	1.941	97	3.892	97	7.371	92	9.665	60	18.377	57	21.422	33
5	1.00	100	1.925	96	3.836	96	6.781	85	9.286	58	15.718	49	20.143	31
2.5	1.00	100	1.913	96	3.707	93	6.353	79	9.285	58	13.896	43	17.990	28
1	1.00	100	1.876	94	3.598	90	5.725	72	7.474	47	11.411	36	12.601	20
0.5	1.00	100	1.573	79	3.104	78	5.554	69	6.618	41	8.468	26	9.739	15
0.1	1.00	100	1.252	63	2.892	72	4.830	60	4.897	31	5.877	18	4.355	7

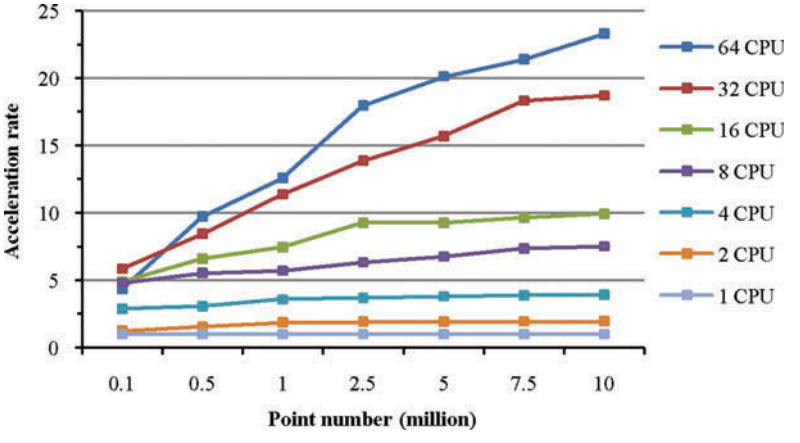


Figure 12. Relative speed-up versus number of points.

measures whether the CPU is used completely. $Ep = Sp/p$. Table 4 shows the relative speed-up and efficiency of the proposed algorithm with a fixed point set. Figure 12 shows the corresponding graph for the relative speed-up versus the size of the point set when the numbers of processors were fixed.

As shown in Table 4, when the number of CPUs was fixed, both the relative speed-up and efficiency indicators improved gradually as the number of points increased. When the number of CPUs was low, the relative speed-up, *Sp,* increased slowly as shown in Figure 12. This was because the corresponding efficiency was high and the CPU was therefore fully utilized. When the number of points was fixed, *Sp* increased with the number of CPUs, whereas the efficiency *Ep* exhibited the opposite trend. In particular, when a 16-core CPU was used, *Ep* decreased dramatically with less than 60% CPU utilization. This was because processing progresses were inconsistent with different CPUs when the number of subsets was high, that is, some CPUs were waiting after computing their tasks until the processing of all the subsets was completed. Therefore, selecting an appropriate number of processors can improve the efficiency of the algorithm, but it also avoids wasting the resources.

Downloaded by [Nanjing University] at 04:10 29 September 2014

6. Conclusions

Although existing sequential algorithms for constructing the Delaunay triangulation satisfy current demands in terms of execution efficiency, the wall time for constructing the Delaunay triangulation continues to grow as the size of the data sets increases. In this study, we proposed a parallel algorithm for constructing the Delaunay triangulation based on a twofold-divide-and-conquer scheme, which has the following advantages: (1) After the subdivision, each subset contains a similar number of points, thereby avoiding skinny triangles, which may be generated by strip splits and cost more time to process because more edges need to be switched during optimization. (2) Partitioning a point set and merging the Delaunay sub-triangulations are more flexible than the normal divide-and-conquer algorithm. The Delaunay triangulations in subsets are merged with Delaunay triangles based on the topological relationships between two adjacent sub-triangulations. (3) After the subdivision, the point set in each subset becomes data independent; thus, the algorithm is more likely to obtain a desired relative speed-up and it has a high portability.

Funding

This work is supported by the National Natural Science Foundation of China [grant number 41371017] and the Program for New Century Excellent Talents in University (NCET-13-0280).

References

An, P. T., and L. H. Trang. 2012. "A Parallel Algorithm Based on Convexity for the Computing of Delaunay Tessellation." *Numerical Algorithms* 59 (3): 347–357. doi:10.1007/s11075-011-9493-2.

Batista, V. H. F., D. L. Millman, S. Pion, and J. Singler. 2010. "Parallel Geometric Algorithms for Multi-core Computers." *Computational Geometry* 43 (8): 663–677. doi:10.1016/j.comgeo.2010.04.008.

Blelloch, G. E., G. L. Miller, J. C. Hardwick, and D. Talmor. 1999. "Design and Implementation of a Practical Parallel Delaunay Algorithm." *Algorithmica* 24 (3–4): 243–269. doi:10.1007/PL00008262.

Brönnimann, H. 2001. "Designing and Implementing a General Purpose Halfedge Data Structure." In *Proceedings of the International Workshop on Algorithm Engineering*, vol. 2141, 51–66. Berlin: Springer.

Chen, M. B., T. R. Chuang, and J. J. Wu. 2006. "Parallel Divide-and-Conquer Scheme for 2D Delaunay Triangulation." *Concurrency and Computation: Practice and Experience* 18 (12): 1595–1612. doi:10.1002/cpe.1007.

Chen, Y., and X. Zhu. 2013. "An Integrated GIS Tool for Automatic Forest Inventory Estimates of *Pinus Radiata* from LiDAR Data." *GIScience & Remote Sensing* 50 (6): 667–689.

Cignoni, P., D. Laforenza, R. Perego, R. Scopigno, and C. Montani. 1995. "Evaluation of Parallelization Strategies for an Incremental Delaunay Triangulator in E3." *Concurrency: Practice and Experience* 7: 61–80. doi:10.1002/cpe.4330070106.

Dwyer, R. A. 1987. "A Faster Divide-and-Conquer Algorithm for Constructing Delaunay Triangulations." *Algorithmica* 2: 137–151. doi:10.1007/BF01840356.

Fortune, S. 1987. "A Sweepline Algorithm for Voronoi Diagrams." *Algorithmica* 2: 153–174. doi:10.1007/BF01840357.

Guibas, L. J., D. E. Knuth, and M. Sharir. 1992. "Randomized Incremental Construction of Delaunay and Voronoi Diagrams." *Algorithmica* 7: 381–413. doi:10.1007/BF01758770.

Guibas, L. J., and J. Stolfi. 1985. "Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams." *ACM Transactions on Graphics* 4 (2): 74–123. doi:10.1145/282918.282923.

Kolingerová, I., and J. Kohout. 2002. "Optimistic Parallel Delaunay Triangulation." *The Visual Computer* 18 (8): 511–529. doi:10.1007/s00371-002-0173-z.

Downloaded by [Nanjing University] at 04:10 29 September 2014

Lawson, C. L. 1977. "Software for C' Surface Interpolation." In *Mathematical Software III*, edited by J. R. Rice, 161–194. New York: Academic Press.

Lee, D. T., and B. J. Schachter. 1980. "Two Algorithms for Constructing a Delaunay Triangulation." *International Journal of Computer and Information Sciences* 9 (3): 219–242. doi:10.1007/BF00977785.

Lee, S., C. I. Park, and C. M. Park. 2001. "An Improved Parallel Algorithm for Delaunay Triangulation on Distributed Memory Parallel Computers." *Parallel Processing Letters* 11 (2/3): 341–352. doi:10.1142/S0129626401000634.

Petroselli, A. 2012. "LIDAR Data and Hydrological Applications at the Basin Scale." *GIScience & Remote Sensing* 49 (1): 139–162. doi:10.2747/1548-1603.49.1.139.

Puppo, E., L. S. Davis, D. De Menthon, and Y. A. Teng. 1994. "Parallel Terrain Triangulation." *International Journal of Geographical Information Systems* 8 (2): 105–128. doi:10.1080/02693799408901989.

Shamos, M. I., and D. Hoey. 1975. "Closest-point Problems." In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, 151–162. Washington, DC: IEEE Computer Society.

Su, P., and R. L. S. Drysdale. 1997. "A Comparison of Sequential Delaunay Triangulation Algorithms." *Computational Geometry: Theory and Applications* 7 (5–6): 361–385. doi:10.1016/S0925-7721(96)00025-9.

Teng, Y. A., F. Sullivan, I. Beichl, and E. Puppo. 1993. "A Data-Parallel Algorithm for Three-dimensional Delaunay Triangulation and its Implementation." In *Proceedings of the ACM/IEEE Conference on Supercomputing*, 112–121. New York: ACM.

Wang, J., C. Cui, Y. Rui, L. Cheng, Y. Pu, W. Wu, and Z. Yuan. 2014. "A Parallel Algorithm for Constructing Voronoi Diagrams Based on Point-set Adaptive Grouping." *Concurrency and Computation: Practice and Experience* 26 (2): 434–446. doi:10.1002/cpe.3005.

Watson, D. F. 1981. "Computing the N-dimensional Delaunay Tessellation with Application to Voronoi Polytopes." *The Computer Journal* 24 (2): 167–172. doi:10.1093/comjnl/24.2.167.

Wu, H., X. Guan, and J. Gong. 2011. "Parastream: A Parallel Streaming Delaunay Triangulation Algorithm for LiDAR Points on Multicore Architectures." *Computers & Geosciences* 37 (9): 1355–1363. doi:10.1016/j.cageo.2011.01.008.

Downloaded by [Nanjing University] at 04:10 29 September 2014