

A parallel method to accelerate spatial operations involving polygon intersections

Chen Zhou, Zhenjie Chen & Manchun Li

To cite this article: Chen Zhou, Zhenjie Chen & Manchun Li (2018) A parallel method to accelerate spatial operations involving polygon intersections, International Journal of Geographical Information Science, 32:12, 2402-2426, DOI: [10.1080/13658816.2018.1508689](https://doi.org/10.1080/13658816.2018.1508689)

To link to this article: <https://doi.org/10.1080/13658816.2018.1508689>



Published online: 29 Aug 2018.



Submit your article to this journal [↗](#)



Article views: 244



View Crossmark data [↗](#)



Citing articles: 1 View citing articles [↗](#)



RESEARCH ARTICLE



A parallel method to accelerate spatial operations involving polygon intersections

Chen Zhou , Zhenjie Chen and Manchun Li

Department of Geographic Information Science, Nanjing University, Nanjing, P. R. China

ABSTRACT

Polygon intersection is an important spatial data-handling process, on which many spatial operations are based. However, this process is computationally intensive because it involves the detection and calculation of polygon intersections. We addressed this computation issue based on two perspectives. First, we improved a method called boundary algebra filling to efficiently rasterize the input polygons. Polygon intersections were subsequently detected in the cells of the raster. Owing to the use of a raster data structure, this method offers advantages of reduced task dependence and improved performance. Based on this method, we developed parallel strategies for different procedures in terms of workload decomposition and task scheduling. Thus, the workload across different parallel processes can be balanced. The results suggest that our method can effectively accelerate the process of polygon intersection. When addressing datasets with 1,409,020 groups of overlapping polygons, our method could reduce the total execution time from 987.82 to 53.66 s, thereby obtaining an optimal speedup ratio of 18.41 while consistently balancing the workloads. We also tested the effect of task scheduling on the parallel efficiency, showing that reducing the total runtime is effective, especially for a lower number of processes. Finally, the good scalability of the method is demonstrated.

ARTICLE HISTORY

Received 28 June 2018

Accepted 1 August 2018

KEYWORDS

Polygon intersection;
boundary algebra filling;
parallel computing;
workload balance

1. Introduction

In the fields of computer graphics and geographical information science (GIS), the calculation of polygon intersection is an important process in many spatial operations such as vector union and overlay, buffer construction, and topology cleaning (Liu *et al.* 2007, Martínez *et al.* 2009, Wang *et al.* 2010, Fotheringham and Rogerson 2013, Longley *et al.* 2015). In these operations, the intersections between polygons must first be detected, and then the resulting polygons are configured (Wang *et al.* 2012). Polygon overlay operation, for example, is often based on a sweep line algorithm (Bentley and Ottmann 1979) for detecting line-segment intersections and a data structure called the doubly connected edge list to correctly reconstruct the topology for the resulting polygons (De Berg *et al.* 2000, Xiao 2016). A similar logic is also adopted in more specialized operations such as polygon clipping (Weiler and Atherton 1977, Vatti 1992, Greiner and Hormann 1998), polygon union (Fan *et al.* 2014), and spatial join (Brinkhoff *et al.* 1994). For

large datasets, researchers have improved methods to speedup the process by reducing the need to use spatial indexing methods to detect polygons for intersection, and considerable efforts have been devoted toward developing tree-based spatial indices to accelerate this process; these include the R-tree, R* tree, and R+ tree (Brinkhoff *et al.* 1994, Simion *et al.* 2012). In these methods, the polygons with nonintersecting minimum bounding rectangles (MBRs) are rapidly filtered out, and thus the computational efficiency can be improved (Fan *et al.* 2014).

The aforementioned computational approaches are sequential and generally efficient when dealing with spatial datasets. However, with the ever increasing volume of spatial data, these methods cannot meet the demands of rapid data processing (Hawick *et al.* 2003). Therefore, many researchers have directed their efforts toward parallel computing (Simion *et al.* 2012). A key strategy in developing a parallel algorithm is to rationally decompose workloads in a way that ensures workload balance (Shekhar *et al.* 1998). Some studies have decomposed vector polygons based on their amount, wherein each parallel process holds an approximately equal number of polygons (Agarwal *et al.* 2012, Shi 2012, Puri *et al.* 2013). Other studies proposed methods to decompose vector polygons based on grids, superimposing uniform grids with equal areas over the input layers and assigning each grid to a parallel process (Waugh and Hopkins 1992, Wang 1993, Healey *et al.* 1997, Zhou *et al.* 1998, Wang *et al.* 2015). In the two aforementioned strategies, each parallel process employs spatial indexing to detect polygon intersections that intersect with the corresponding polygons, achieving parallelization.

Three critical problems affect the efficiency of the above-mentioned parallel strategies for polygon intersection. First, the decomposition of polygons into independent subsets through these strategies is difficult, and different parallel processes are task dependent. Accordingly, frequent communication is needed, increasing the computational complexity. Moreover, although the spatial indexing method can rapidly detect polygons with intersecting MBRs, many redundant polygons that do not actually intersect may still be involved in the process of intersection calculations. Thus, additional time is required for determining whether these polygons intersect, especially for large datasets. Finally, vector polygons have inherently complex structures, and they vary significantly in size, shape, topology, and thus computational complexity. However, the existing methods only consider the load balance of polygons based on amount or area. They do not fully consider the variation in polygons and are not intended to balance workloads in a parallel computation setting, thus providing limited performance. Therefore, designing a rational parallel method that can achieve a proper degree of workload balance remains a great challenge.

The goal of this study is to develop an efficient parallel method to accelerate spatial operations involving polygon intersections. In our method, the vector data is first converted into a raster, which in turn is used for detecting polygon intersections. We improved an efficient algorithm called the boundary algebra filling (BAF) to use it to rapidly identify polygon intersections. The use of the raster data also helps reduce task dependence and improve the efficiency during parallel processing. In the remainder of this paper, we first discuss the rasterization algorithm and how to use the rasterized data for intersection detection (Section 2). Then, in Section 3, we describe the design and implementation of the parallel strategies. Section 4 demonstrates the use of our method in a set of experiments. We conclude the paper in Section 5 with a discussion on the future directions of this work.

2. Polygon intersection detection by using BAF

First, we introduce our ideas through two figures (Figure 1), and then provide their formal algorithms. The BAF algorithm rasterizes a polygon through the simple operations of addition and subtraction (Ren 1989), and is more efficient than other methods including the scan-line and ray-crossing algorithms (Zhou *et al.* 2015). Figure 1 illustrates the basic steps of the BAF algorithm. We trace the line segments on the boundary of a polygon in the counter-clockwise direction (Figure 1(a)). When the segment moves downward, we subtract the attribute from the cells to the left of the boundary (Figure 1(b)). In contrast, if the segment moves upward, the attribute value of the polygon is added to all the cells to the left of the boundary (Figure 1(c)). After the tracing is finished, all the pixels inside the polygon are assigned the value of the polygon (Figure 1(d)).

We then extended the original BAF so that it can be used to detect polygon intersections. This is illustrated using the detection of pixels with three overlapping polygons (Figure 2). Here, we assumed the attribute value of each polygon as 1. After running the BAF algorithm for the three polygons, the number in each pixel indicates the number of polygons that overlap at that pixel. For example, the two pixels marked by value 3 are where all three polygons overlap.

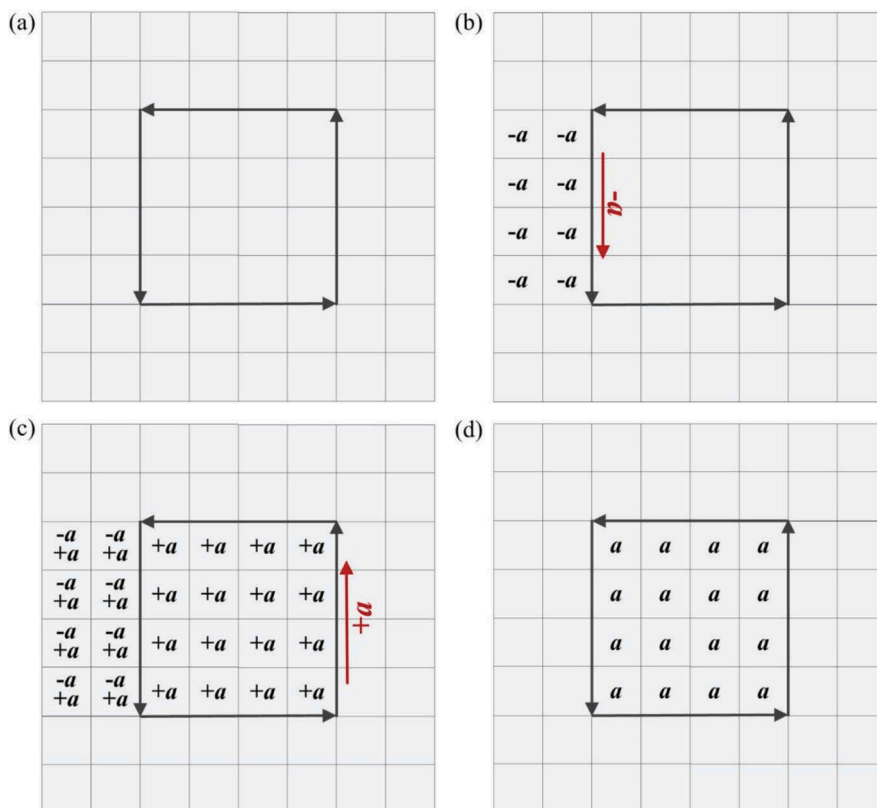


Figure 1. Basic steps of the BAF algorithm: (a) tracing all boundaries of a polygon counter-clockwise, (b) downward- and (c) upward-boundary processing, and (d) completed tracing.

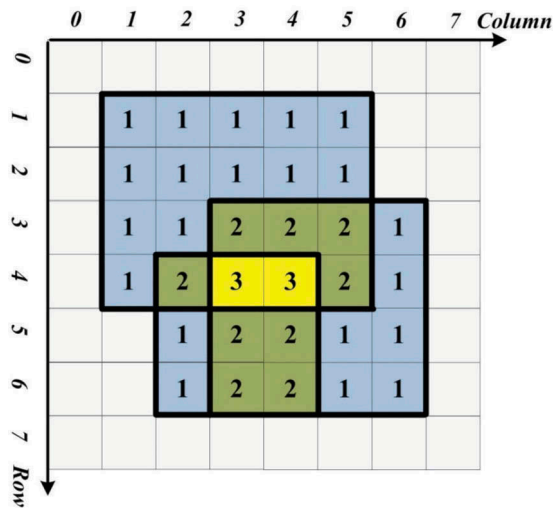


Figure 2. Result of three overlapping polygons after running the BAF algorithm.

2.1. Data structures

We used a few structures to store critical information regarding the result. In our implementation of the algorithm, the MBR of all input polygons was calculated first. Then, by using the MBR and a specified cell size, the number of rows (*RowSize*) and columns (*ColSize*) could be calculated. The cell values of the raster were stored in a one-dimensional array called *hDstDS* initialized at 0. Another array of the same size called *pIDArray* was also created to store polygon IDs and was initialized at -1. For a cell located at row *Row* and column *Col* in a raster, its index in *hDstDS* and *pIDArray* can be readily calculated as $locate = Row \times ColSize + Col$. All the polygons were assigned an attribute value of 1.

However, the challenge was to correctly extract the IDs of the polygons from the raster pixels overlapped by these polygons. The extracted polygon IDs can then be used to form different independent polygon groups. To achieve this goal, we used the run-length encoding (RLE) method (Pountain 1987) to represent the cells with the same value. In our case, the value of each pixel represents the number of overlapped polygons. In addition, we also explicitly stored the unique IDs of those overlapping polygons. The actual data structure is listed in Listing 1. The encoding of the pixels is denoted as $RLE\{Row, StartCol, EndCol, (pID_0, pID_1, \dots, pID_{pn-1}), pn\}$, where *Row* is the row number, *StartCol* and *EndCol* are column numbers of the starting and ending pixels of the RLE, respectively, and *pGroup* stores the IDs of *pn* overlapping polygons.

A polygon is usually composed of a single outer ring and several inner rings. The proposed algorithm rasterizes a polygon that may contain multiple rings. The input of the algorithm includes the row and column coordinates of the points of the polygon, number of points in each ring, number of rings, and attribute value of the polygon. The coordinates of the outer ring are stored first, followed by the sequential storage of the coordinates of the inner rings. The input also comprises an array called *hDstDS* that stores the values for all the pixels after the algorithm has been run.

Listing 1. Data structure of the RLE.

```

Struct RLE
{
    /* Location information of the RLE */
    Int Row;                /* The row number */
    Int StartCol;           /* The column number of the starting pixel */
    Int EndCol;             /* The column number of the ending pixel */

    /* Information of the overlapping polygons */
    Int *pGroup;            /* IDs of the overlapping polygons */
    Int pn;                 /* Number of overlapping polygons */
};

```

The details of the BAF algorithm are presented in the pseudo-code outlined in [Listing 2](#). We set up a counter initialized to 0 to help correctly find the coordinates of points of each ring (line 1). The algorithm loops through each of the rings in the polygon (line 2). For each ring, we obtained the number of points (line 3), calculated the MBR coordinates (line 4), and obtained the total number of rows and columns (line 5). We then traced the boundaries in the counter-clockwise direction by using a loop passing through all the points in the ring (lines 6–34). In the loop, we first determined the direction of each line segment by obtaining the coordinates of the line, and then calculated the pixels crossed by the line segment (lines 7–9). If the segment proceeded in the downward direction (line 10), we subtracted the attribute from the cells to the left of the boundary for an outer ring and added the attribute to the cells to the left of the boundary for an inner ring (lines 11–21). Otherwise, if the line proceeded in the upward direction (line 22), the attribute value of the polygon was added to all the cells to the left of the boundary for an outer ring and the attribute was subtracted from the cells to the left of the boundary for an inner ring (lines 23–34). Finally, we added the number of points of the ring to the counter (line 35). After this procedure is run on a polygon, the cells inside the polygon are assigned the correct attribute values of the polygon.

We used the BAF algorithm on multiple polygons by applying it individually. During the entire process, we maintained two arrays: *hDstDS*, which stores the total number of polygons overlaying on one pixel, and *pIDArray*, which stores the ID of the last polygon overlaying on each pixel. The BAF maintains the correctness of the first array (*hDstDS*), and we developed another algorithm that constructs the ID lists for the pixels. This algorithm uses RLE to expedite the process ([Listing 3](#)), which is illustrated in [Figure 3](#) by using three hypothetical polygons. Here, we assumed the attribute value (*pValue* in the BAF algorithm) for all polygons as 1. [Figure 3\(a\)](#) shows the process of using BAF on the first polygon, resulting in the values in array *hDstDS*.

The RLE algorithm takes the current array of *pIDArray* as input and updates its contents during the process. It also takes the MBR bounds of the given polygon as input. The algorithm loops through each pixel in the MBR (lines 1 and 2) and determines the index of the pixel in the *hDstDS* and *pIDArray* arrays (line 3). We first filtered out the pixels that fall within the polygon (line 4). Among these pixels, we consider those contained by two or more polygons. If a pixel is in two polygons (line 5), we set this pixel as the starting pixel of an RLE data structure and determined the subsequent pixels in the same row that are also in the same polygons (lines 6–11). Accordingly, we constructed an RLE data structure by using this information and stored it in the *RLEGroup* (line 12). We can directly add this RLE into *RLEGroup* because this set of pixels

Listing 2. Pseudo-code of the BAF algorithm.**Algorithm BAF is****Input:***PointRow* the row coordinates of the points of the polygon*PointCol* the column coordinates of the points of the polygon*pPointNum* the numbers of points in each ring*pRingNum* the number of rings*pValue* the attribute value of the polygon**Output:***hDstDS* the array to store the values of all the pixels after rasterization

```

1 Let sn be the starting number of points of each ring, sn = 0
2 for iRing = 0 to pRingNum - 1 do
3   Get the number of points ncount = pPointNum[iRing]
4   Calculate MBR coordinates of the current ring: minCol, maxCol, minRow, and maxRow
5   Get the total number of rows and columns, RowSize and ColSize
6   for in = 0 to ncount - 1 do
7     Get the current boundary ((PointCol[sn+in],PointRow[sn+in]),(PointCol[sn+in+1],PointRow[sn+in+1]))
8     Calculate the row and column numbers of the pixels crossed by the boundary, crossrow and crosscol
9     Get the number of the pixels crossed by the boundary crosscount
10    if (PointRow[sn+in] < PointRow[sn+in+1]) then
11      for icross = 0 to crosscount - 1 do
12        for ipixel = minCol to crosscol[icross] - 1 do
13          ilocate = crossrow[icross]× ColSize + ipixel
14          if (iRing == 0)
15            hDstDS[ilocate] = hDstDS[ilocate] - pValue
16          else
17            hDstDS[ilocate] = hDstDS[ilocate] + pValue
18          end if
19        end for
20      end for
21    end if
22    if (PointRow[sn+in] > PointRow[sn+in+1]) then
23      for icross = 0 to crosscount - 1 do
24        for ipixel = minCol to crosscol[icross] - 1 do
25          ilocate = crossrow[icross]× ColSize + ipixel
26          if (iRing == 0)
27            hDstDS[ilocate] = hDstDS[ilocate] + pValue
28          else
29            hDstDS[ilocate] = hDstDS[ilocate] - pValue
30          end if
31        end for
32      end for
33    end if
34  end for
35  sn = sn + ncount
36 end for

```

is contained by two polygons and is added into the group for the first time. So far, on the same row, we processed all the pixels starting from the current pixel until the last pixel with the same polygon ID. Therefore, we skipped all these pixels when updating the value of *iCol* (line 13). This part of the process is illustrated in Figure 3(b).

If the pixel lies in more than two polygons (line 15), it is considered as the starting pixel of a new RLE, and then we search for the subsequent pixels in the same row with the same number of overlaying polygons (lines 16–21). There is a possibility that the pixels in the new RLE are already included in some existing RLEs. Thus, we searched the existing RLEs in *RLEGroup* from previous runs (line 22) for pixels that are on the same row and have one less number of polygons (line 23) as well as those that overlay the new RLE (lines 24–26). Further, we tested for possible cases of split RLEs (lines 27–32) and stored the

Listing 3. Pseudo-code of the RLE algorithm.**Algorithm** RLE is**Input:***hDstDS* the array to store the values of all the pixels after rasterization*pIDArray* the array to store the polygon IDs*ColSize* the total number of columns in *hDstDS**RowSize* the total number of rows in *hDstDS**minCol*, *maxCol*, *minRow*, *maxRow* the column numbers of the four corners of the polygon's MBR*pCurrentID* the ID of the polygon**Output:***RLEGroup* the array to store RLEs

```

1 for iRow = minRow to maxRow do
2   for iCol = minCol to maxCol do
3     Calculate the location of the current raster pixel: locate = iRow × ColSize + iCol
4     if (the pixel falls within the current polygon) then
5       if (hDstDS[locate] == 2) then
6         for iPixel = iCol to maxCol do
7           Calculate the location of iPixel: ilocate = iRow × ColSize + iPixel
8           if (iPixel is not inside the polygon || hDstDS[ilocate] != 2 || pIDArray[ilocate] != pIDArray[locate]) then
9             break
10          end if
11        end for
12        Create an RLE {iRow, iCol, iPixel - 1, (pIDArray[locate], pCurrentID), 2} and add into RLEGroup
13        iCol = iPixel - 1
14      end if
15    if (hDstDS[locate] > 2) then
16      for iPixel = iCol to maxCol do
17        Calculate the location of iPixel: ilocate = iRow × ColSize + iPixel
18        if (iPixel is not inside the polygon || hDstDS[ilocate] != hDstDS[locate]) then
19          break
20        end if
21      end for
22      for iG = 0 to RLEGroup.size() - 1 do
23        if (RLEGroup[iG].Row == iRow && RLEGroup[iG].pn == hDstDS[locate] - 1) then
24          if (RLEGroup[iG].StartCol < iPixel - 1 && RLEGroup[iG].EndCol > iCol) then
25            olMinCol = max(RLEGroup[iG].StartCol, iCol)
26            olMaxCol = min(RLEGroup[iG].EndCol, iPixel - 1)
27            if (RLEGroup[iG].StartCol < olMinCol) then
28              Create an RLE{iRow, RLEGroup[iG].StartCol, olMinCol - 1, RLEGroup[iG].pGroup, hDstDS[ilocate] - 1}
29            end if
30            if (RLEGroup[iG].EndCol > olMaxCol) then
31              Create an RLE{iRow, olMaxCol + 1, RLEGroup[iG].EndCol, RLEGroup[iG].pGroup, hDstDS[ilocate] - 1}
32            end if
33            Put the new RLEs into RLEGroup and remove the old RLE RLEGroup[iG]
34            Create an RLE {iRow, olMinCol, olMaxCol, (RLEGroup[iG].pGroup, pCurrentID), hDstDS[ilocate]}
35            Add the new RLE into RLEGroup
36            iG = iG - 1
37          end if
38        end if
39      end for
40      iCol = iPixel - 1
41    end if
42  end if
43 end for
44 end for
45 Update pIDArray and assign pCurrentID to the raster pixels within the polygon.

```

results in *RLEGroup* (line 33). Finally, we placed the pixels with the current number of polygons into a new RLE and added it to *RLEGroup* (lines 34 and 35). This process is illustrated in Figure 3(c). Next, we updated the column of the search (line 40), updated *pIDArray*, and assigned *pCurrentID* to the raster pixels within the polygon (line 45).

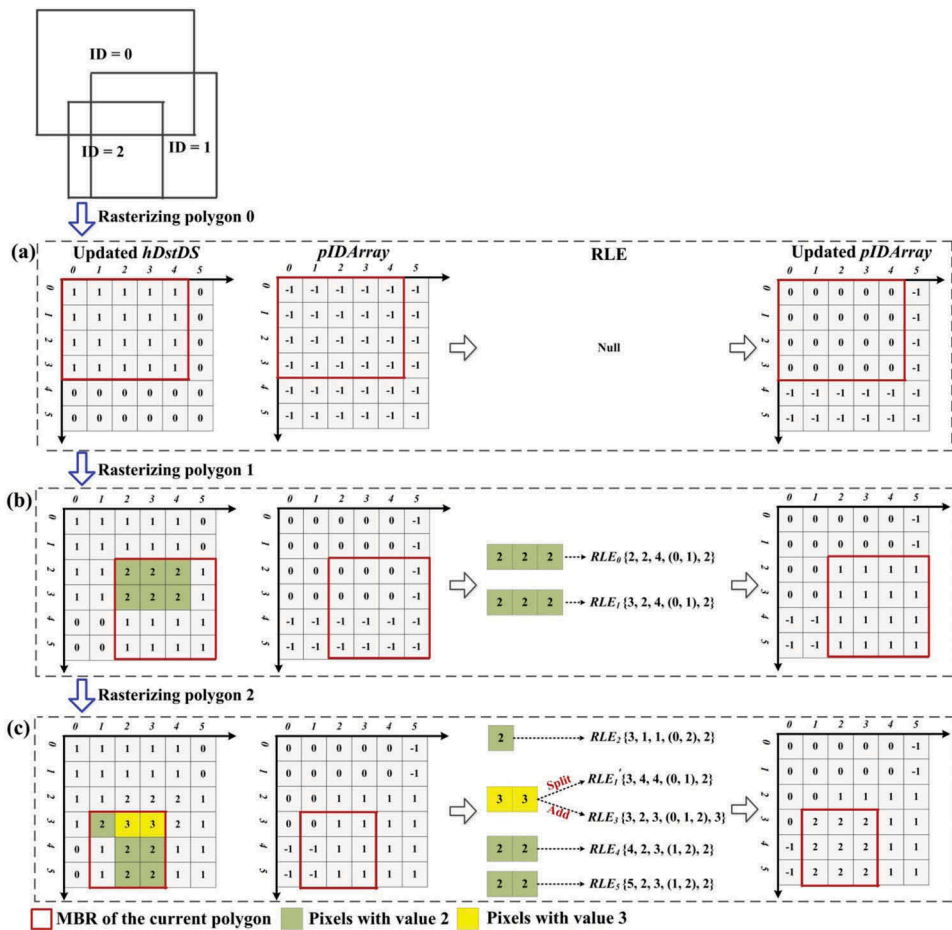


Figure 3. RLE-based extraction of polygon IDs: (a) pixels in one polygon are skipped and pixels in (b) two or (c) more polygons are processed.

2.2. Intersection calculations

The results of running BAF and RLE provide all information necessary for calculating the intersections of a set of polygons. The steps of this process are listed in Listing 4. The algorithm takes the number and IDs of the intersected polygons as input and outputs the resultant polygons. During this process, the Geometry Engine Open Source (GEOS) library is used to calculate the intersections and differences of the polygons (OSGeo 2014). When there are only two polygons in a polygon group (line 1), we can calculate their intersections immediately (lines 2 and 3). If there are more than two polygons (line 4), we must continuously test the intersections between each of the original polygons with all the polygons calculated during the process. We used *pQueue* to ensure that the next polygon to be tested is at the beginning of the queue, and we used another queue called *label* to ensure only those that must be outputted are marked as 1 while all others are marked 0 (line 5). Lines 6–26 show the continuous testing of intersections. After each original polygon is tested, the values that have been used are removed and the new calculated values are appended to the

Listing 4. Pseudo-code of intersection calculations.**Algorithm** INT is**Input:***pGroup* stores the IDs of intersected polygons*pn* the number of intersected polygons**Output:**

resultant polygons

```

1 if (pn = 2) then
2   pID0 = pGroup[0]; pID1 = pGroup[1]
3   Calculate the intersected polygon of pID0 and pID1
4 else if (pn > 2) then
5   Create a processing queue pQueue and a label queue label
6   pID0 = pGroup[0]; pID1 = pGroup[1]
7   Calculate the intersected polygon of pID0 and pID1, pIntersect
8   Calculate the difference polygon of pID0 and pID1, pDiff0, that is the part in pID0 but do not overlap with pID1
9   Calculate the difference polygon of pID1 and pID0, pDiff1, that is the part in pID1 but do not overlap with pID0
10  Put pIntersect, pDiff0, and pDiff1 into pQueue
11  Set the labels of pIntersect, pDiff0, and pDiff1 as 1, 0, and 0; and put into label
12  for ip = 2 to pn - 1 do
13    Get the number of polygons in pQueue, pQNum
14    for i = 0 to pQNum - 1 do
15      Calculate the intersected polygon of pQueue[i] and pQueue[ip]
16      Calculate the difference polygon of pQueue[i] and pQueue[ip], pDiff0
17      Calculate the difference polygon of pQueue[ip] and pQueue[i], pDiff1
18      Put pIntersect and pDiff0, into pQueue
19      Set the label of pIntersect as 1 and put the label into label
20      if (label[i] == 1)
21        Set the label of pDiff0 as 1 and put the label into label
22      else
23        Set the label of pDiff0 as 0 and put the label into label
24      end if
25      pGroup[ip] = pDiff1
26    end for
27    Remove the polygons in pQueue from pQueue[0] to pQueue[pQNum - 1]
28    Remove the labels in label from label[0] to label[pQNum - 1]
29    Put pGroup[ip] into pQueue, and set its label as 0
30  end for
31 end if
32 Output the polygons in pQueue with label 1

```

end of the corresponding queues (lines 27–29). In this way, after the process is completed, the polygons labeled 1 will indicate the result for polygon intersection (line 32).

3. Design and implementation of parallel strategies

Both the rasterization and intersection procedures can be parallelized. In this section, we first design workload decomposition strategies for these two aspects separately and then introduce a task scheduling strategy for the overall parallel process.

3.1. Workload decomposition

3.1.1. Workload decomposition for polygon rasterization

The rasterization procedure in this study consists of two steps. In the first step, each polygon is rasterized independently and the computational load is dependent on the number of points on each polygon. The second step involves the finding of the RLE groups, and the number of rows in the raster determines the computational granularity.

As a result, the original polygons are decomposed into subsets of rows so that the numbers of points in each subset are as equal as possible.

When decomposing the original problem into n subsets, the following steps are executed. (1) The original polygons are collected and their enveloping MBR is calculated. (2) The total number of points tn , which is the sum of the number of points of each polygon, is calculated. Then, each subset should hold stn points, where $stn = tn/n$. (3) The enveloping MBR is decomposed by rows such that the subsets have an identical number of rows, and thus identical areas. (4) Next, each subset's corresponding polygons are extracted through a spatial range query, and then its actual number of points atn is calculated. (5) The spatial location of each subset is adjusted such that atn is approximately equal to stn . (6) Steps (4) and (5) are repeated until the spatial locations of all the subsets are determined. By using this approach, the areas of the resultant subsets are not necessarily equal but their numbers of points are close. Consequently, the workload is approximately balanced for the parallel rasterization procedure (Figure 4).

To deal with polygons across multiple subsets (the red polygons in Figure 4(a)), the part of the polygon that falls inside a subset is processed, whereas the part outside is ignored. With this approach, polygons across different subsets can be processed in parallel (Figure 4(b)). The subsets are independent with respect to each other; consequently, the task dependence is reduced.

3.1.2. Workload decomposition for intersection calculations

In the INT algorithm, the workload depends on each polygon group. In this study, we developed an empirical model for estimating the workload of a polygon group.

Factors that influence the processing efficiency of a polygon group include the total number of points and the number of polygons in the group. We created a simulated dataset by using a regularly shaped polygon to test the impact of each factor on the execution time. More specifically, the base polygon contains four points and covers an area of 294.12 m². We stacked instances of the base polygon to form polygon groups

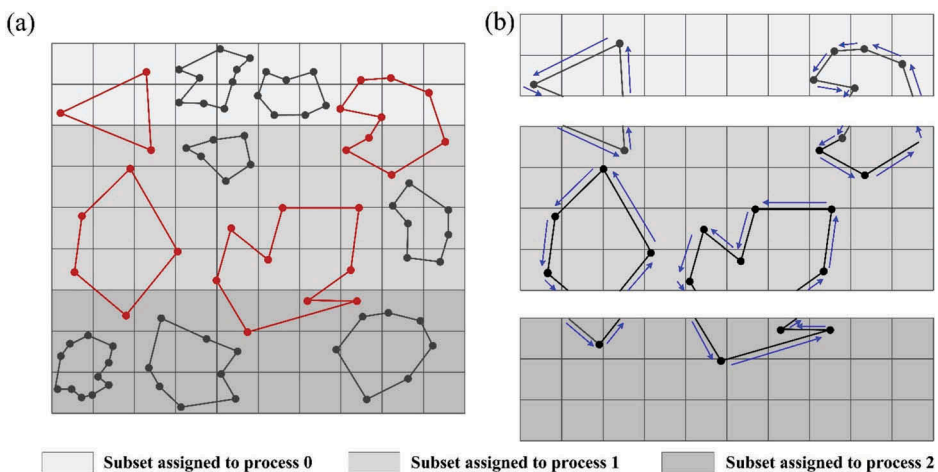


Figure 4. Workload decomposition for polygon rasterization: (a) decomposing the original polygons into three subsets and (b) addressing polygons across multiple subsets.

with different numbers of polygons (Figure 5(a)). We also added more points to the boundaries of each polygon in a group to form different configurations with respect to the total number of points (Figure 5(b)).

Overall, we performed two experiments. (1) We created 100 polygon groups with two base polygons but with different total number of points ranging from 16 to 1600. (2) We stacked the varying numbers of the base polygon to create five more sets of polygon groups. Each set contained 11 polygon groups with 2–12 polygons, respectively. Polygon groups in the same set had an identical total number of points and those in different sets had a varying number of points ranging from 100 to 500.

For the results of the first experiment, we calculated the execution time of each polygon group, and then determined the relationship between the execution time and number of points (Figure 6(a)). The result of fitting a quadratic polynomial model to the results yields $R^2 = 0.984$. Therefore, the estimated execution time of a group with two polygons (T_2) can be described as

$$T_2 = 5.3 \times 10^{-9}tn^2 - 7.9 \times 10^{-7}tn + 0.0001, \quad pn = 2 \quad (1)$$

where tn denotes the total number of points in the group and pn is the number of overlapping polygons.

In the second experiment, for each of the sets in the dataset, we first calculated the time ratio (tr) for each polygon group with two more polygons, which is stated as

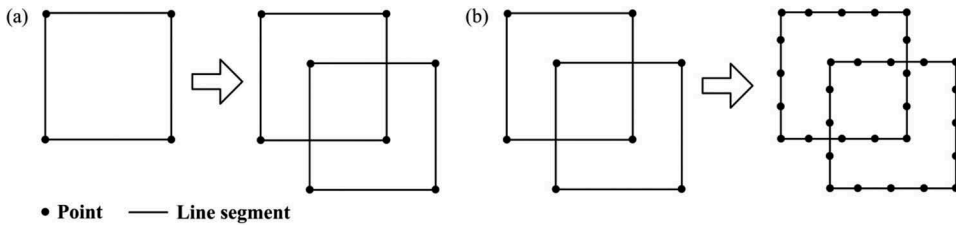


Figure 5. Process for creating simulated datasets. (a) A polygon group created by stacking two base polygons. (b) A polygon group with 32 points created by adding 12 points to each polygon in a group with 8 points.

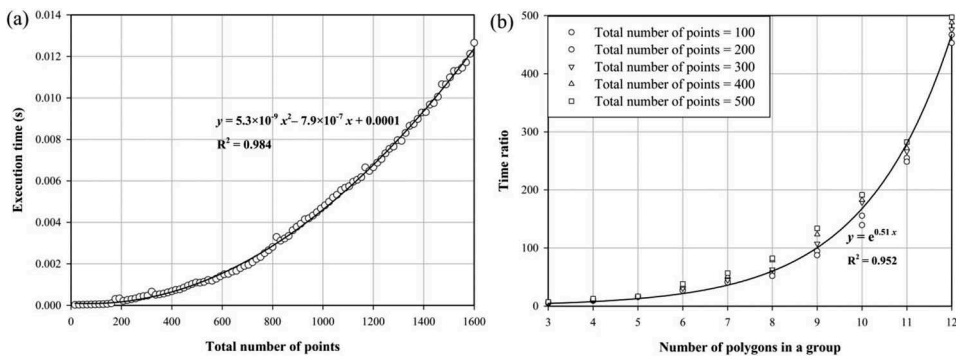


Figure 6. Influence of different factors on execution time: (a) effect of the total number of points on execution time and (b) effect of the number of polygons on execution time.

$$tr = \frac{T_{pn}}{T_2}, \quad pn > 2 \quad (2)$$

where T_{pn} and T_2 are the execution times of a group with pn ($pn > 2$) and two polygons, respectively. We then fit the relationship between the time ratio and number of polygons (Figure 6(b)). The results show that the polygon groups with an identical number of polygons, but belonging to different sets, have approximately equal tr values. This indicates that the execution time of a group with pn ($pn > 2$) polygons is approximately tr times that of a group with two polygons and the same number of points. When using a natural exponential model to fit the relationship, $R^2 = 0.952$. Therefore, the relationship between the time ratio and number of polygons can be described as

$$tr = e^{0.51pn}, \quad pn > 2 \quad (3)$$

According to Equations (1–3), the estimated execution time of a group with more than two polygons (T_{pn}) can be described as

$$T_{pn} = e^{0.51pn} T_2, \quad pn > 2 \quad (4)$$

Finally, the workload of the i -th polygon group (w_i) can be calculated as

$$w_i = \frac{T^i - \min_{j=1-n} \{T^j\}}{\max_{j=1-n} \{T^j\} - \min_{j=1-n} \{T^j\}} \quad (5)$$

where T^i is the estimated execution time of a polygon group and is calculated using Equation (1) or (4), and $\max_{j=1-n} \{T^j\}$ and $\min_{j=1-n} \{T^j\}$ are the maximum and minimum times among n polygon groups, respectively. The value of workload ranges from 0 to 1, and a higher value represents a higher workload.

By using Equation (5), the workload of a polygon group can be calculated using the empirical model, and the total workload is the sum of the workload of each group. Then, the total workload can be evenly decomposed into different subsets such that these subsets have different numbers of polygon groups but approximately the same workload.

3.2. Task scheduling

The tasks in the rasterization and intersection procedures are called *PR* and *INT*, respectively, and we used a master–slave paradigm (Beaumont *et al.* 2003) to deal with these tasks. One of the invoked parallel processes was used as the master process, while the others were used as slave processes. In addition, we employed a task queue and a process queue to manage the tasks and parallel processes, respectively (Figure 7). The task queue stores the information necessary for the tasks to be processed, including the ID, type of task, and required data. In particular, the data information of a *PR* task represents the starting and ending row numbers in the raster dataset; while that of a *INT* task refers to the IDs of intersecting polygons in different polygon groups. The process queue records the IDs of the idle slave processes.

When invoking p parallel processes, the master process executes the following steps. (1) It decomposes the total workload in the rasterization procedure into $sg(p - 1)$ subsets (where sg

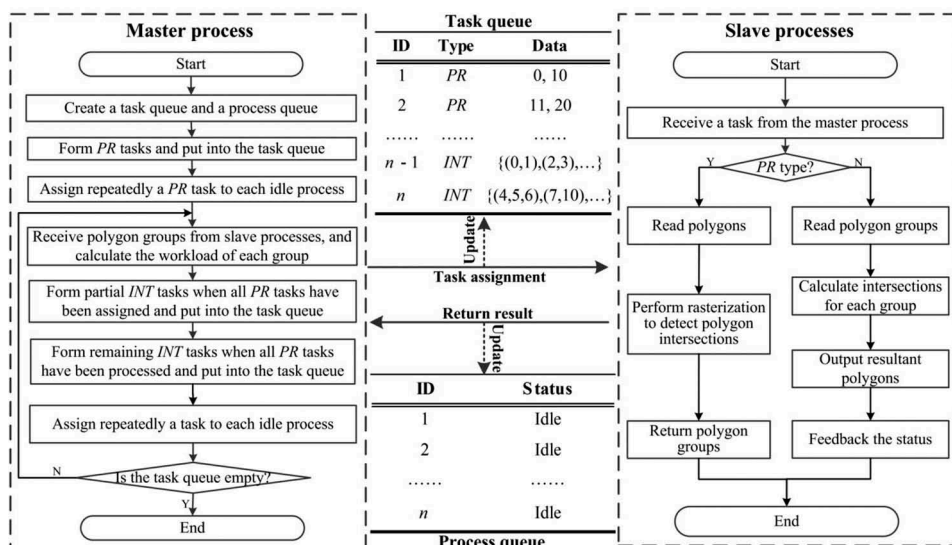


Figure 7. Task scheduling process.

is an integer greater than 1) such that each subset denotes an independent *PR* task. It situates the formed *PR* tasks and all the slave processes into the task and process queues, respectively. (2) It repeatedly acquires a *PR* task from the task queue, and then assigns it to an idle parallel process. (3) During parallel processing, the master process receives the IDs of the calculated polygon groups from the slave processes that have completed their *PR* tasks, and adds these slave processes to the process queue. Then, it calculates the workloads of received polygon groups. (4) When a parallel process is idle, the master process acquires another task from the top of the task queue and assigns it to the idle process for handling. (5) When all *PR* tasks have been allocated, the master process decomposes the polygon groups received into $sg(p-1)$ subsets such that each subset denotes an independent *INT* task, and adds them to the end of the task queue. When *PR* tasks are completely processed, it decomposes the remaining received polygon groups into $sg(p-1)$ *INT* tasks, and adds them to the task queue. (6) The master process continues to allocate tasks to idle processes until the task queue becomes empty. A slave process executes the following steps. (1) It receives the information of a task from the master process and reads the corresponding data. (2) When processing a *PR* task, the slave process forms the polygon groups and then returns the results to the master process. Otherwise, it calculates and outputs the resultant polygons.

By dynamically allocating and appending the two types of tasks, the slave processes are continually kept busy with no idle time. Moreover, the master process measures the workloads of the polygon groups while the slave processes are working in parallel processing, reducing the total time. The value of *sg* represents the scheduling granularity in our strategy, and Section 4.3 shows the testing of the effect of different *sg* values on parallel efficiency.

3.3. Implementation

The parallel algorithm was implemented using C++ under the Message Passing Interface (MPI) environment using OpenMPI version 1.10.2 (organization of Software in the Public

Interest, Inc. (SPI), New York, USA). The Geospatial Data Abstraction Library 2.0.2 was employed to read and write geographical data, and GEOS 3.5.0 (Open Source Geospatial Foundation (OSGeo), Chicago, USA) was used to calculate the intersection of two polygons.

The algorithm takes the number of parallel processes, raster cell size, and layers of polygons as input and outputs the resultant polygons (Listing 5). It initializes parallel settings and creates parallel processes (line 1). The master process (with rank 0) collects all polygons and assigns them ascending IDs (lines 2 and 3). It then creates a task queue and a process queue and performs task scheduling during parallel processing (lines 4–20). Multiple slave processes handle different types of tasks in parallel (line 21). A slave process first receives a task from the master process (line 22). For a *PR* task, the slave process invokes the BAF and RLE algorithms to form polygon groups and returns control to the master process (lines 23–40). Otherwise, it invokes the INT algorithm to calculate polygon intersections and outputs the resultant polygons (lines 41–50). Finally, all processes exit the parallel execution (line 51).

4. Experiments and results

Experiments were performed on an IBM parallel cluster consisting of nine computing nodes. The configuration of each node was as follows: two Intel(R) Xeon(R) CPUs (E5-2620; clock at 2.00 GHz; six-core model of twelve threads), 16 GB of memory, and a 2 TB hard drive with a dual-port Gigabit Ethernet network.

Three datasets for Shanghai, China, were employed in the experiments (Table 1). Dataset 1 comprises the land-use data for 2009, with data volume of 1.06 GB and 1,371,765 polygons. Dataset 2 contains 392 polygons, representing the districts of Shanghai. We practically calculated the intersections of these datasets to analyze the land-use cover and change for each district. Dataset 3 is a simulated dataset created from dataset 1. Specifically, a number of polygons were shifted manually in dataset 1 to intersect with others such that there were 163,934 polygon groups with two polygons. We practically calculated the intersections of this type of dataset for topology cleaning.

In our evaluation, a typical R-tree based algorithm was implemented for comparison. The possible overlapping polygons were first detected using the R-tree, and then the intersections were calculated using the GEOS library. Further, two parallel versions of this algorithm were implemented using a grid – or amount-based strategy. In the experiments, we first evaluated the accuracy of the proposed parallel algorithm. We then compared our parallel algorithm with the parallel R-tree based algorithms in terms of the execution time, speedup, and load balance. We also examined the effect of task scheduling on parallel efficiency. Finally, we tested the scalability with different datasets and spatial operations.

4.1. Evaluation of accuracy

In this experiment, we used the proposed parallel algorithm to calculate the intersections between datasets 1 and 2. There were 1,409,020 polygon groups found in these datasets. Nine parallel processes were used and the raster cell size for the rasterization procedure was set at 10 m. The data format of the result was Esri Shapefile (*.shp) with a data volume of 1.11 GB.

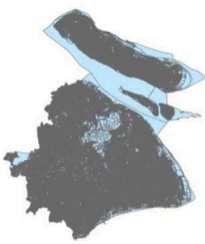
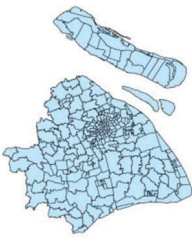

Listing 5. Pseudo-code for parallel processing.**Algorithm** PP is**Input:** p the number of parallel processes RCS the specified raster cell size for polygon rasterization $InputLayer$ the input layers of original polygons**Output:** $ResultLayer$ the layer to store the resultant polygons

```

1 Initialize the parallel settings, and create  $p$  parallel processes
2 If ( $rank == 0$ ) then
3   Collect all polygons from  $InputLayer$  and give sequential ascending IDs
4   Create a task queue  $tq$  and a process queue  $pq$ 
5   Form  $sg(p - 1)$   $PR$  tasks for parallel polygon rasterization and put into  $tq$ 
6   Assign a  $PR$  task to each slave process
7   while ( $tq$  is not empty)
8     Receive polygon groups from a slave process, and calculate workload of each group
9     Put the slave process into  $pq$ 
10    while ( $pq$  is not empty)
11      Assign a task from  $tq$  to an idle process and remove the process from  $pq$ 
12    end while
13    If ( $PR$  tasks in  $tq$  have been assigned) then
14      Form partial  $INT$  tasks and put into  $tq$ 
15    end if
16    If ( $PR$  tasks have been processed) then
17      Form remaining  $INT$  tasks and put into  $tq$ 
18    end if
19    Update  $tq$  and  $pq$ 
20  end while
21 else if ( $rank > 0$ ) then
22   Receive the info of a task from the master process
23   If ( $PR$  task) then
24     Read the corresponding polygons  $poly$ , and get the number of polygons  $polynum$ 
25     Create an array  $hDstDS$  to store rasterization result according to  $RCS$ 
26     Create an array  $pIDArray$  to store the polygon IDs, and create an array to store the RLEs  $RLEGroup$ 
27     for  $ipoly = 0$  to  $polynum - 1$  do
28       Get the row and column coordinates of the points  $PointRow$ ,  $PointCol$ 
29       Get the number of points in each ring  $pPointNum$ , and the number of rings  $pRingNum$ 
30        $BAF(PointRow, PointCol, pPointNum, pRingNum, 1, hDstDS)$ 
31       Get the column and row numbers of  $hDstDS$ ,  $ColSize$  and  $RowSize$ 
32       Get the coordinates of the four corners of the polygon's MBR  $minCol$ ,  $maxCol$ ,  $minRow$ ,  $maxRow$ 
33        $RLE(hDstDS, pIDArray, ColSize, RowSize, minCol, maxCol, minRow, maxRow, poly[ipoly], RLEGroup)$ 
34     end for
35     Create an array to store the formed polygon groups  $pGroup$ 
36     for  $iG = 0$  to  $sizeof(RLEGroup) - 1$  do
37       Put  $RLEGroup[iG].pGroup$  into  $pGroup$ 
38     end for
39     Remove the repetitive groups in  $pGroup$ , and return to the master process
40   end if
41   If ( $INT$  task) then
42     Read the corresponding polygon groups  $pGroup$ , and get the number  $pgnum$ 
43     for  $ipg = 0$  to  $pgnum - 1$  do
44       Get the number of polygons within the current group  $pn$ 
45       Create an array to store the results of the current group  $respoly$ 
46        $INT(pGroup[ipg], pn, respoly)$ 
47       Output the polygons into  $ResultLayer$ 
48     end for
49   end if
50 end if
51 Exit the parallel execution

```


Table 1. Datasets employed in the experiments.

Name	Real datasets		Simulated dataset
	Dataset 1	Dataset 2	Dataset 3
Overview	  		
Projection	Gauss–Kruger projection		
Data volume	1.06 GB	2.58 MB	1.06 GB
Area (km ²)	8,132.79	6769.93	8,132.79
Number of polygons	1,371,765	392	1,371,765

We also obtained a result by using the professional GIS software called ArcGIS 10.2 (Environmental Systems Research Institute, Inc. (Esri), California, USA). To compare the results calculated using our algorithm and ArcGIS, we selected four regions from the experimental datasets to show the consistency in detail (Table 2). The results show that, for each region, the polygons derived using the proposed method were the same in shape and location as those derived using ArcGIS. Further, the comparison of the two results showed no difference between them. This suggests that the results obtained using the proposed algorithm and ArcGIS are the same at 10 m.

Moreover, the accuracy was measured quantitatively as the ratio of the total area of polygons obtained using our result ($Area_p$) to those obtained using the ArcGIS result ($Area_A$) (Dong *et al.* 2009); this ratio can be described as

$$Accuracy = \frac{Area_p}{Area_A} \times 100\% \quad (6)$$

We changed the raster cell size from 1 to 30 m and calculated the corresponding accuracy (Table 3). The ArcGIS results yield an area of 6764.70 km². As the cell size increased, the accuracy decreased gradually from 100% to 83.06%. This indicates that the cell size obviously affected the accuracy. When the cell size was 10 m or less, the accuracy was 100%, indicating that all polygon groups were correctly identified and calculated. A smaller cell size usually leads to more intensive computations. Considering the tradeoff between accuracy and performance, we selected 10 m as the appropriate cell size.

4.2. Evaluation of parallel efficiency

4.2.1. Execution time and speedup

Execution time and speedup are two basic indices used to evaluate the efficiency of a parallel algorithm (Guan *et al.* 2016). Execution time refers to the time between invoking the algorithm and completing the last parallel process. Speedup is the ratio between the time consumed by parallel and sequential versions of the algorithm.

Table 2. Comparison of the computational results obtained using the proposed parallel algorithm and ArcGIS.

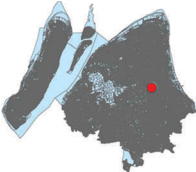
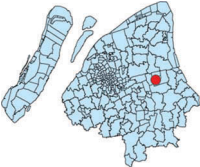


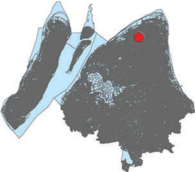
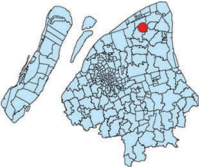



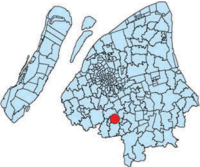



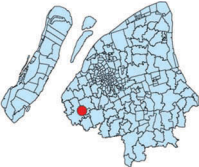


Region	Location in original datasets		Result obtained using ArcGIS	Result obtained using the proposed algorithm
	Dataset 1	Dataset 2		
1				
2				
3				
4				

Table 3. Accuracy of our proposed approach for different raster cell sizes with respect to the area obtained through ArcGIS.

Raster cell size (m)	Area obtained using our result (km ²)	Area obtained using the ArcGIS result (km ²)	Accuracy (%)
1	6764.70	6764.70	100.00
5	6764.70		100.00
10	6764.70		100.00
15	6577.99		97.24
20	6390.61		94.47
25	6038.17		89.26
30	5618.51		83.06

We first used the proposed sequential algorithm and the R-tree based algorithm to calculate the intersections between datasets 1 and 2, respectively, and recorded the number of detected polygon groups and different time components during the running of the algorithms (Table 4). In the results, the time of data reads/writes did not vary much between the two algorithms. The time required for detecting intersections by using our algorithm was more than that required for detecting intersections by using the R-tree based algorithm; however, the time required for calculating the intersection by using our algorithm was much shorter. Thus, our proposed algorithm requires much less execution time (987.82 s) than the R-tree based algorithm (1268.74 s). Although the R-tree based algorithm could rapidly detect polygons with intersecting MBRs, the number of formed polygon groups was much larger than that obtained through our algorithm. Therefore, it needed more time to determine if the polygons actually intersect. In contrast, the number of polygon groups detected through our algorithm equaled the actual number, and the groups can be immediately calculated to form the result, reducing the total execution time. This result indicates that the proposed algorithm performs better when identifying polygon intersections.

We then used the proposed parallel algorithm and two versions of the parallel R-tree based algorithms to calculate the intersections. We increased the number of processes from 2 to 120 and calculated the corresponding total execution time and speedup. The results in Figures 8(a,b) show that the behaviors of the execution time and speedup for different parallel algorithms were similar. When the number of processes increased from 2 to 108, the execution time first decreased sharply and then gradually before reaching a plateau. The speedup increased steadily and reached its peak value after 108 processes, beyond which the speedup began to gradually decrease. In addition, the minimum times were 53.66, 83.86, and 87.26 s for different parallel algorithms, and the optimal speedups were 18.41, 15.13, and 14.54, respectively. Between the two versions of the parallel R-tree based algorithms, the algorithm using the amount-based parallel strategy performed better. In contrast, the proposed parallel algorithm was more effective than

Table 4. Number of polygon groups and different time components of the proposed sequential algorithm and R-tree based algorithm.

	Number of polygon groups		Execution time (s)			
	Detected number	Actual number	Data reads/writes	Intersection detection	Intersection calculation	Total
R-tree based algorithm	2,787,743	1,409,020	23.14	51.21	1194.39	1268.74
Proposed algorithm	1,409,020		32.01	218.36	737.45	987.82

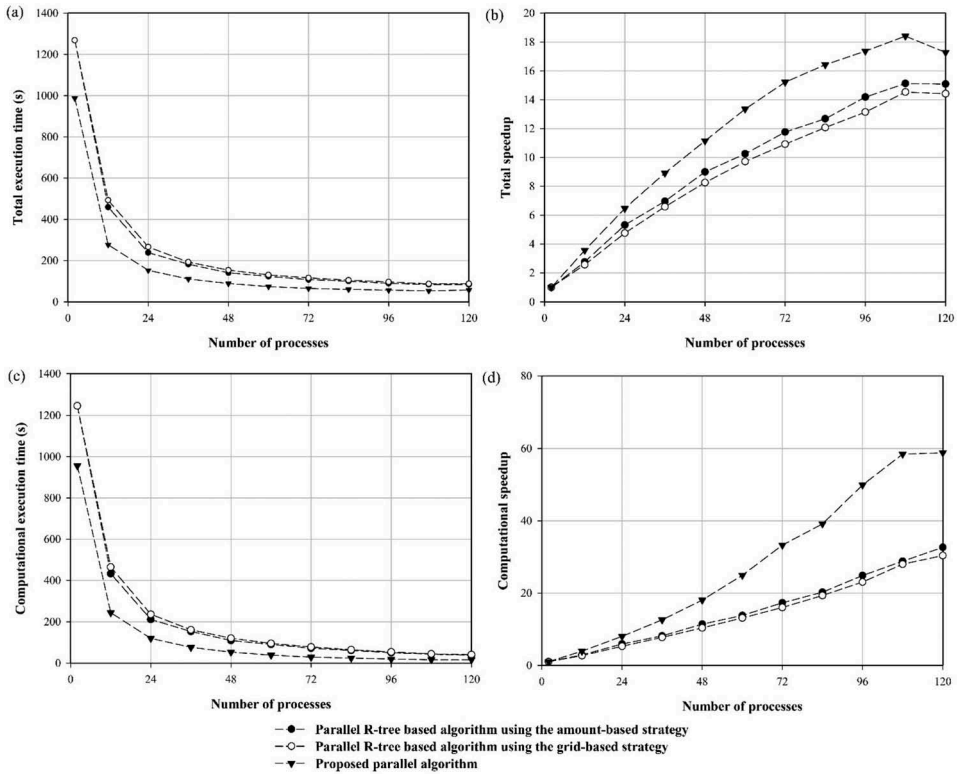


Figure 8. Comparison of the parallel efficiency of different algorithms: (a) total execution time, (b) total speedup, (c) computational execution time, and (d) computational speedup.

both the R-tree based algorithms, achieving much shorter execution time and higher speedup. This suggests that the proposed parallel method can efficiently decrease the total execution time because of its advantages of reduced task dependency and improved degree of workload balance during parallel processing.

In a parallel algorithm, the efficiency of the components that can be totally parallelized intuitively reflects the effectiveness of the applied parallel strategies. Therefore, we further calculated the computational execution time, only including the time of the components that can be totally parallelized (i.e. parallel intersection detection and calculations), and then calculated the corresponding computational speedups (Figures 8(c,d)). In the results, our parallel algorithm achieved much higher computational speedup (58.84) than the other two parallel algorithms (32.64 and 30.37). These results further demonstrate the higher efficiency of the proposed parallel strategies.

4.2.2. Load balance

We measured the load balance of the parallel algorithm by using the ratio of the time consumed by the slowest slave parallel process ($T_{slowest}$) to that consumed by the fastest slave process ($T_{fastest}$) (Zhou et al. 2015), which is defined as

$$\text{Load balance} = \frac{T_{\text{slowest}}}{T_{\text{fastest}}} \quad (7)$$

Here, the load balance is greater than 1, and a lower value indicates a higher load balance. In this experiment, the load balance values for different parallel algorithms were calculated separately (Figure 9).

For the two versions of the parallel R-tree based algorithms, the load balance values decreased sharply with the increase in the number of processes because a larger number of processes can gradually compensate for the unbalanced workloads between different processes, thereby improving the degree of load balance. For the algorithms using the grid-based and amount-based parallel strategies, the maximum load balance values were 6.27 and 5.75, respectively. This indicates that significant workload imbalance occurred in these parallel algorithms, and thus the conventional strategies cannot ensure a sufficient degree of load balance during parallel processing. By contrast, the values of the proposed parallel algorithm were relatively steady, ranging from 1.98 to 1.37. This suggests that our parallel strategies can achieve consistently balanced workloads, thereby mitigating data skewing and reducing the total execution time.

4.3. Effect of task scheduling on parallel efficiency

To test the effect of task scheduling on parallel efficiency, we changed the *sg* value from 2 to 7, ran our parallel algorithm to calculate the intersections of datasets 1 and 2, and recorded the corresponding total execution time. We also recorded the runtime of the parallel algorithm without task scheduling, that is, the intersection calculation procedure was not executed until the rasterization procedure was completely finished and the number of tasks in each procedure equaled the number of slave processes.

The results in Figure 10 show that the total execution time first reduced and then increased as the *sg* value increased. The minimum runtime of the algorithm using task scheduling was less than that of the algorithm without task scheduling. For

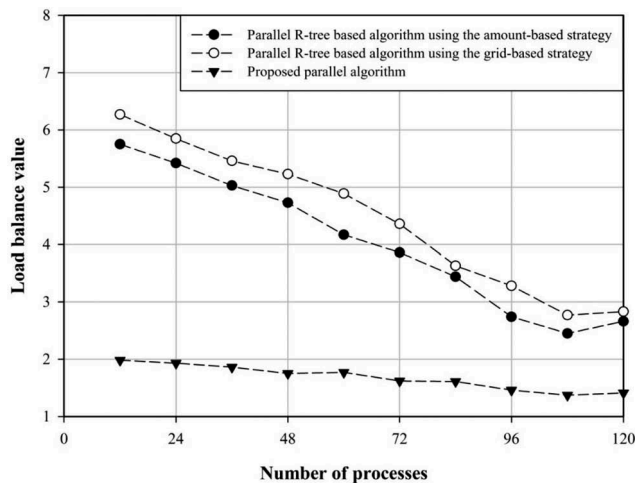


Figure 9. Load balance values of different parallel algorithms.

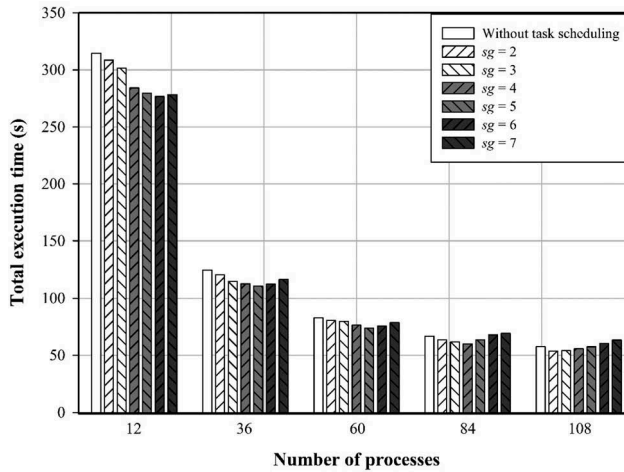


Figure 10. Total execution time of the parallel algorithm without and with task scheduling for various sg values.

different numbers of processes, the total execution time decreased from 314.56 to 276.70 s, from 124.61 to 110.74 s, from 82.81 to 73.94 s, from 66.78 to 60.12 s, and from 57.67 to 53.66 s; accordingly, the reduced runtime was 37.86, 13.87, 8.87, 6.66, and 4.01 s, respectively. This verifies that the proposed strategy can effectively conceal a part of the execution time and avoid idle slave processes, therefore clearly reducing the total time.

The optimized sg values at which the minimum times were obtained were 6, 5, 5, 4, and 2 for different numbers of processes. For each number of processes, a balance was achieved between the use of the computing resource and overheads of scheduling at the optimized sg value. Moreover, the optimized sg values as well as the reduced times decreased gradually with the increasing number of processes. When using a higher number of processes, an increase in sg value rapidly increases the scheduling overhead, which in turn decreases the efficiency. In addition, the performance improvement ratio, which equals the reduced time divided by the original total time, was 12.04%, 11.13%, 10.71%, 9.97%, and 6.95% for different numbers of processes. Therefore, the process of task scheduling is more efficient for a lower number of processes.

4.4. Evaluation of scalability

In this experiment, we tested the scalability of the proposed parallel method using three test cases. Specifically, case A represents the situation in which polygon intersections only occurred in a single layer; here, dataset 3 was used with 163,934 polygon groups. Case B represents the situation in which intersections occur between two layers, and datasets 1 and 2 were used with 1,409,020 polygon groups. Case C represents the situation in which intersections occur between more than two polygons; here, datasets 2 and 3 were used consisting of 1,215,483 polygon groups with two polygons and 171,586 groups with three polygons. Case C is the most complicated of the three cases. We used the proposed parallel algorithm and the two versions of the parallel R-tree

Table 5. Optimal execution time and speedup of different parallel algorithms for three test cases.

		Case A	Case B	Case C
Optimal execution time (s)	Parallel R-tree based algorithm using the amount-based strategy	15.87	83.86	109.06
	Parallel R-tree based algorithm using the grid-based strategy	16.71	87.26	121.52
Optimal speedup	Proposed parallel algorithm	10.25	53.66	62.82
	Parallel R-tree based algorithm using the amount-based strategy	14.87	15.13	13.85
	Parallel R-tree based algorithm using the grid-based strategy	14.12	14.54	12.43
	Proposed parallel algorithm	17.27	18.41	18.73

based algorithms to calculate the test cases, recording the corresponding optimal execution time and speedup (Table 5). The results show that for each case, our parallel algorithm can clearly reduce execution time and increase the speedup. In particular, for case C, the maximum speedup by using our algorithm (18.73) was much higher than that by using other algorithms (13.85 and 12.43). This suggests that our parallel algorithm scaled well with different types of datasets, especially for complicated datasets.

We also tested the scalability of the parallel method for other spatial operations, namely polygon union and buffer construction. We used the BAF and RLE algorithms to detect polygon intersections, and the workload decomposition and task scheduling methods to achieve parallelization. For each operation, the two versions of the parallel R-tree based algorithms were also implemented for comparison. We used these parallel algorithms to calculate the intersections of datasets 1 and 2, and then recorded the total execution time and speedup (Figure 11). For parallel polygon union, the optimal speedups achieved using our parallel algorithm and the parallel R-tree based algorithms were 18.46, 15.22, and 14.63, respectively; and for parallel buffer construction, the optimal speedups were 18.26, 15.21, and 14.02, respectively. These results show that for each spatial operation, the algorithm using the proposed parallel method can perform much better, further verifying proper scalability of our method with other spatial operations.

5. Conclusions and future work

This paper presented a parallel method for speeding up spatial operations involving polygon intersections. The following three conclusions can be drawn from the experiments.

- (1) The accuracy of the proposed parallel algorithm was 100% when the cell size for the rasterization procedure was 10 m or smaller.
- (2) The proposed sequential algorithm outperformed the R-tree based algorithm when identifying polygon intersections. Moreover, the parallel algorithm can significantly improve the efficiency of polygon intersection. For 1,409,020 polygon groups, the proposed algorithm reduced the total execution time from 987.82 to 53.66 s and obtained an optimal speedup of 18.41. In addition, the proposed algorithm can consistently balance the workloads among different parallel processes.
- (3) The task scheduling can efficiently reduce the idle time of parallel processes. Moreover, the proposed parallel method scales well for different types of datasets and spatial operations.

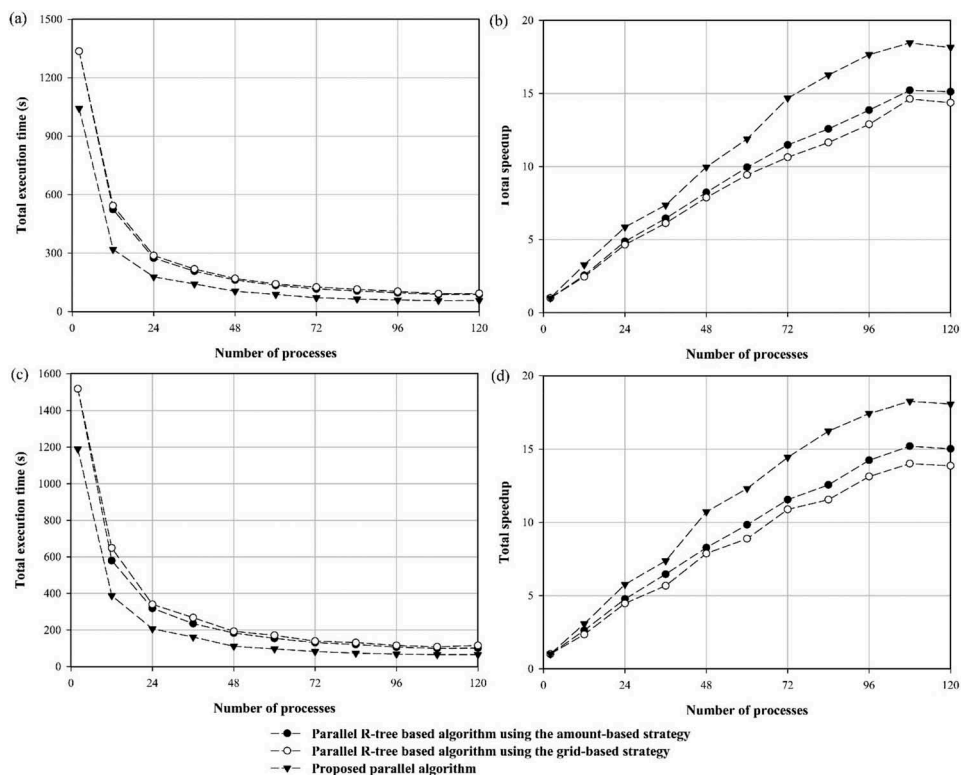


Figure 11. Execution times and speedup for two spatial operations obtained using different parallel methods: (a) execution time and (b) speedup for the parallel polygon union operation and (c) execution time and (d) speedup of for the parallel buffer construction operation.

Given the rapid advancements in graphics processing units (GPUs), we plan to further improve the parallel efficiency using collaborative CPUs and GPUs in the future. In particular, we will focus on the design and implementation of rational task scheduling among different CPUs and GPUs when addressing large polygon datasets.

Disclosure statement

No potential conflict of interest was reported by the authors.

Funding

This work was supported by the National Key R&D Program of China [Grant number 2017YFB0504205], National Natural Science Foundation of China [Grant number 41571378], and the Program B for Outstanding PhD Candidate of Nanjing University [Grant number 201702B067].

Notes on contributors

Chen Zhou is an assistant research fellow at Department of Geographic Information Science, Nanjing University. His research topics include high-performance geocomputation, parallel computing, and geospatial analysis.

Zhenjie Chen is an associate professor at Department of Geographic Information Science, Nanjing University. His research interests include web-based GIS, land use and land cover change, and spatial decision support systems.

Manchun Li is a professor at Department of Geographic Information Science, Nanjing University. His research interests include spatial data mining, cartography, environmental and ecological modeling.

ORCID

Chen Zhou  <http://orcid.org/0000-0002-2707-2624>

Zhenjie Chen  <http://orcid.org/0000-0002-3033-8470>

Manchun Li  <http://orcid.org/0000-0002-8689-9007>

References

- Agarwal, D., et al., 2012. A system for GIS polygonal overlay computation on Linux cluster—an experience and performance report. In: *Proceedings of IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, 21–25 May 2012 Shanghai, New York: IEEE, 1433–1439. doi: [10.1109/IPDPSW.2012.180](https://doi.org/10.1109/IPDPSW.2012.180)
- Beaumont, O., Legrand, A., and Robert, Y., 2003. The master-slave paradigm with heterogeneous processors. *IEEE Transactions on Parallel and Distributed Systems*, 14 (9), 897–908. doi:[10.1109/TPDS.2003.1233712](https://doi.org/10.1109/TPDS.2003.1233712)
- Bentley, J.L. and Ottmann, T.A., 1979. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, 28 (9), 643–647. doi:[10.1109/TC.1979.1675432](https://doi.org/10.1109/TC.1979.1675432)
- Brinkhoff, T., et al., 1994. Multi-step processing of spatial joins. *Acm Sigmod Record*, 23 (2), 25–34. doi:[10.1145/191839.191880](https://doi.org/10.1145/191839.191880)
- De Berg, M., et al., 2000. *Computational geometry*. Heidelberg: Springer-Verlag Berlin Heidelberg.
- Dong, H., Cheng, Z., and Fang, J., 2009. One rasterization approach algorithm for high performance map overlay. In: *Proceedings of 17th international conference on geoinformatics*, 12–14 August 2009 Fairfax, New York: IEEE, 1–6. doi: [10.1109/GEOINFORMATICS.2009.5293561](https://doi.org/10.1109/GEOINFORMATICS.2009.5293561)
- Fan, J.F., et al., 2014. DWSI: an approach to solving the polygon intersection-spreading problem with a parallel union algorithm at the feature layer level. *Boletim De Ciencias Geodesicas*, 20 (1), 159–182. doi:[10.1590/s1982-21702014000100011](https://doi.org/10.1590/s1982-21702014000100011)
- Fotheringham, S. and Rogerson, P., 2013. *Spatial analysis and GIS*. Boca Raton, FL: CRC Press.
- Greiner, G. and Hormann, K., 1998. Efficient clipping of arbitrary polygons. *ACM Transactions on Graphics*, 17 (2), 71–83. doi:[10.1145/274363.274364](https://doi.org/10.1145/274363.274364)
- Guan, Q.F., et al., 2016. A hybrid parallel cellular automata model for urban growth simulation over GPU/CPU heterogeneous architectures. *International Journal of Geographical Information Science*, 30 (3), 494–514. doi:[10.1080/13658816.2015.1039538](https://doi.org/10.1080/13658816.2015.1039538)
- Hawick, K.A., Coddington, P.D., and James, H.A., 2003. Distributed frameworks and parallel algorithms for processing large-scale geographic data. *Parallel Computing*, 29 (10), 1297–1333. doi:[10.1016/j.parco.2003.04.001](https://doi.org/10.1016/j.parco.2003.04.001)
- Healey, R., et al., 1997. *Parallel processing algorithms for GIS*. Boca Raton, FL: CRC Press.
- Liu, Y.K., et al., 2007. An algorithm for polygon clipping, and for determining polygon intersections and unions. *Computers & Geosciences*, 33 (5), 589–598. doi:[10.1016/j.cageo.2006.08.008](https://doi.org/10.1016/j.cageo.2006.08.008)

- Longley, P.A., et al., 2015. *Geographic information science and systems*. New Jersey: John Wiley & Sons.
- Martínez, F., Rueda, A.J., and Feito, F.R., 2009. A new algorithm for computing Boolean operations on polygons. *Computers & Geosciences*, 35 (6), 1177–1185. doi:[10.1016/j.cageo.2008.08.009](https://doi.org/10.1016/j.cageo.2008.08.009)
- Open Source Geospatial Foundation (OSGeo), 2014. *GEOS-Geometry Engine Open Source*. Available from: <http://trac.osgeo.org/geos> [Accessed 10 July 2017].
- Pountain, D., 1987. Run-length encoding. *Byte*, 12 (6), 317–319.
- Puri, S., et al., 2013. MapReduce algorithms for GIS polygonal overlay processing. In: *Proceedings of IEEE 27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, 20–24 May 2013 Cambridge, MA. New York: IEEE, 1009–1016. doi: [10.1109/IPDPSW.2013.254](https://doi.org/10.1109/IPDPSW.2013.254)
- Ren, F.H., 1989. *Theory, method and application of geographical information system*. Thesis (PhD). Peking University.
- Shekhar, S., Chubb, D., and Turner, G., 1998. Declustering and load-balancing methods for parallelizing geographic information systems. *IEEE Transactions on Knowledge and Data Engineering*, 10 (4), 632–655. doi:[10.1109/69.706061](https://doi.org/10.1109/69.706061)
- Shi, X., 2012. *System and methods for parallelizing polygon overlay computation in multiprocessing environment*. United States Patent Application 13/523,196, 20 December.
- Simion, B., Ray, S., and Brown, A.D., 2012. Speeding up spatial database query execution using GPUs. *Procedia Computer Science*, 9, 1870–1979. doi:[10.1016/j.procs.2012.04.205](https://doi.org/10.1016/j.procs.2012.04.205)
- Vatti, B.R., 1992. A generic solution to polygon clipping. *Communications of the ACM*, 35 (7), 56–63. doi:[10.1145/129902.129906](https://doi.org/10.1145/129902.129906)
- Wang, F., 1993. A parallel intersection algorithm for vector polygon overlay. *IEEE Computer Graphics and Applications*, 2, 74–81. doi:[10.1109/38.204970](https://doi.org/10.1109/38.204970)
- Wang, J.C., et al., 2010. A novel algorithm of buffer construction based on run-length encoding. *The Cartographic Journal*, 47 (3), 198–210. doi:[10.1179/000870410X12786821061413](https://doi.org/10.1179/000870410X12786821061413)
- Wang, J.C., et al., 2012. An efficient algorithm for clipping operation based on trapezoidal meshes and sweep-line technique. *Advances in Engineering Software*, 47 (1), 72–79. doi:[10.1016/j.advengsoft.2011.12.003](https://doi.org/10.1016/j.advengsoft.2011.12.003)
- Wang, Y., et al., 2015. Improving the performance of GIS polygon overlay computation with MapReduce for spatial big data processing. *Cluster Computing*, 18 (2), 507–516. doi:[10.1007/s10586-015-0428-x](https://doi.org/10.1007/s10586-015-0428-x)
- Waugh, T.C. and Hopkins, S., 1992. An algorithm for polygon overlay using cooperative parallel processing. *International Journal of Geographical Information Science*, 6 (6), 457–467. doi:[10.1080/02693799208901928](https://doi.org/10.1080/02693799208901928)
- Weiler, K. and Atherton, P., 1977. Hidden surface removal using polygon area sorting. In: *Proceedings of the 4th annual conference on computer, graphics and interactive techniques*. California: ACM, 214–222.
- Xiao, N.C., 2016. *GIS algorithms*. London: Sage Publications.
- Zhou, C., et al., 2015. Data decomposition method for parallel polygon rasterization considering load balancing. *Computers & Geosciences*, 85PA, 196–209. doi:[10.1016/j.cageo.2015.09.003](https://doi.org/10.1016/j.cageo.2015.09.003)
- Zhou, X., Abel, D.J., and Truffet, D., 1998. Data partitioning for parallel spatial join processing. *Geoinformatica*, 2 (2), 175–204. doi:[10.1023/A:100975593105](https://doi.org/10.1023/A:100975593105)