Application of concatenable queue for parallel computational geometry algorithms

Vasyl Tereshchenko
Taras Shevchenko
National University of
Kyiv
Kyiv, Ukraine
vtereshch@gmail.com

Semen Chudakov
Taras Shevchenko
National University of
Kyiv
Kyiv, Ukraine
semen.chudakov7@gmail.com

ABSTRACT

This paper is devoted to the development of an algorithmic model that solves a set of interrelated computational geometry problems efficiently. To do this, an algorithmic environment with a unified data structure is created, which allows to implement complex use cases efficiently with respect to required computational resources. We build the environment based on the "divide and conquer" strategy. Once a convex hull is a key to a set of computational geometry problems, we offer a concatenable queue data structure to maintain it. The data structure is in the form of a modified balanced binary tree. This allows us to perform operations needed in algorithms for a set of problems in $O(\log^2 n)$ time. Furthermore we offer a way to execute the algorithms both sequentially and in parallel. In the future the algorithmic environment can be improved to support other computational models with similar properties for solving problems. As an example, the Voronoi diagram or the Delaunay triangulation can be considered.

Keywords

unified data structure, simulation problem, interrelated problems set, unified algorithmic environment, concatenable queue

1 INTRODUCTION

Nowadays, advanced computer simulations and visualizations of complex scientific researches and large-scale technical projects require to solve simultaneously a set of problems. The core of this set are problems of computational geometry and computer graphics. To solve such problems it is needed to create suitable algorithmic frameworks that would yield accurate results in real-time. Existing methods (ImageJ [Ima19a], IMARIS [Ima19b], iLastic [SSKH11]), that are based on a set of algorithms implementations organized in a package do not result in desirable efficiency and accuracy. It is worth noting that there are a lot of parallel algorithms designed to solve specifically certain computational geometry problems such as in [ACG*88, AL93, AGR94, ACG89, BSV96, Che95, CG88, GJ97, JaJ97, Rei93, Lee90]. Every such algorithm requires its computational resources and is executed independently from others. In such cases identical steps, such as preprocessing and building data structures, are repeated several times.

Therefore, an important aim in developing algorithmic models is to create a universal tool that would have a means to solve efficiently a set of problems. This tool should also execute identical steps of the algorithms once and be able to represent the results of those steps in the form of unified data structures. In [TA10] the notion of a unified algorithmic environment is introduced, which is based on the "divide-and-conquer" principle and takes into account the aforementioned features of the algorithms. In particular, preprocessing and dividing the initial set of data to form a recursion tree is common for all problems and is executed only once. During the merge stage, intermediate results are maintained in a concatenable queue for the convex hull, Delaunay triangulation and Voronoi diagram problems. This model does not repeat identical computations, which yields good performance.

In this article we first describe how the convex hull algorithm for a static set of points is decomposed into separate stages and incorporated into our unified algorithmic environment model (UAEM). Then we detailedly explain how we implement the concatenable queue, which is used in the algorithmic environment. Finally, we make a complexity analysis for the algorithm and test its performance.

2 UNIFIED ALGORITHMIC ENVIRONMENT

In this section we described the principle of how we decompose each algorithm into distinct stages. We then use this partition to avoid repeating the computations in the algorithmic environment. The principle will be shown on a convex hull algorithm, which is similar to the one described in [OL81]. The idea there is to divide the hull into the left and right sub-hulls and represent them with two concatenable queues. This allows to achieve $O(\log^2 n)$ time for update operations such as adding or removing points from the hull. We use the approach to compute the convex hull for a static set of points. Additionally, unlike in [OL81], a convex hull here is divided into upper and lower sub-hulls.

2.1 Algorithms stages

The notion of a convex hull is simple. For a set of points S in a k-dimensional space it is the smallest convex set that comprises S. To solve such a problem means finding a subset in S that is "skeleton" for the convex hull. From now on, we will consider the case where $k = 2$.

To eliminate corner cases during the merging step we need to ensure that there are no 3 points that lie on a horizontal or vertical line. Points that violate this condition are removed from S in the preprocessing stage. Formally, the removal criterion is formulated as follows. For $a = (x_a, y_a)$ we denote $x(a) = x_a$, $y(a) = y_a$. Let the points $a_1, ..., a_k$ lie on one horizontal line and $x(a_1) < ... < x(a_k)$. Then, by the criterion, the points $a_2, a_3, ..., a_{k-1}$ must be removed. Analogously for the vertical case.

The algorithm of removing "inner" repetitions in a sorted array is trivial. To perform the preprocessing described above, it is needed to:

1. Sort points by y (if y coordinates are equal, the x coordinates are compared).
2. Delete "inner" repetitions by y coordinate using the described algorithm.
3. Sort points by x (if x coordinates are equal, the y coordinates are compared).
4. Delete "inner" repetitions by x coordinate.

As a result we get a list of points for which we can apply the recursive convex hull algorithm.

At the stage of dividing the problem into sub-problems, the list of points is split into left and right parts of roughly equal size. This can be done in $O(1)$ time for an interval $[i, j]$ by computing the average of indices i and j.

The recursion stops when there are no more than 3 points in the list. For the base case the list of points can have 2 or 3 elements. In those two cases the convex hull can be trivially constructed. The result of the base case are two concatenable queues representing upper and lower sub-hulls with one or two points in each.

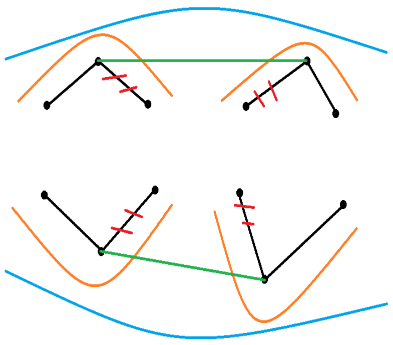
Figure 1: Merging two hulls.

To merge two convex hulls that are separable by a vertical line, we need to find upper and lower tangents that will serve as a basis for the resulting hull. Those tangents are found using the search algorithm described in [OL81]. It remains to split the sub-hulls at four found nodes that form upper and lower tangents and merge the remaining parts. An example of performing such a procedure is shown in Fig. 1.

We use the following markings on the convex hull schemes. Sub-hulls of left and right hulls are marked with orange color, sub-hulls of merged hull are marked with blue color, correct tangents are marked with green color, incorrect tangents are marked with red color and removed edges during merging are overscored with two red lines. Each convex hull is divided into the upper and lower parts due to its representation in the algorithm.

Now we will consider the corner cases that arise when performing the merging. The first of these cases is related to the ambiguity of the position of the utmost left and utmost right points in the described representation. They might be included both in the upper and lower sub-hull. Both points must belong to the upper sub-hulls of the left and right hulls before finding the tangent line, because otherwise such tangent may be found incorrectly. An example of such an incorrect search is given in Fig. 2.

To avoid such a situation, it is necessary to move the aforementioned points to the upper sub-hulls before merging them. For the rightmost point of the left hull and the leftmost point of the right hull we have the following cases. Similarly to the previous argument, they must be transferred to the upper parts of the hulls. And after merging, these points must be transferred to the lower parts of the hull, if they do not belong to the resulting upper part of the final hull. Otherwise, the formed hull may be incorrect. An example of such a case is shown in Fig. 3.

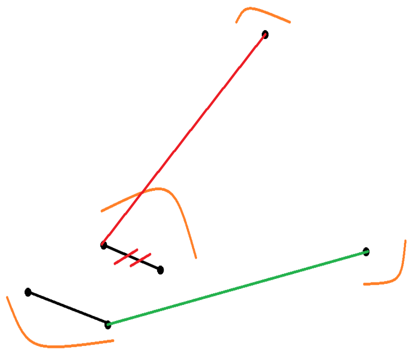
Figure 2: Example of an incorrect position of the utmost left point in the left hull.

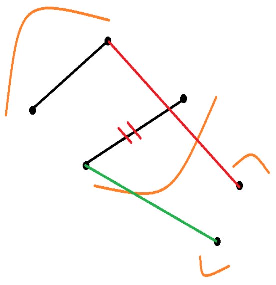
Figure 3: Example of a convex hull with a wrong position of the utmost left point of the left sub-hull.

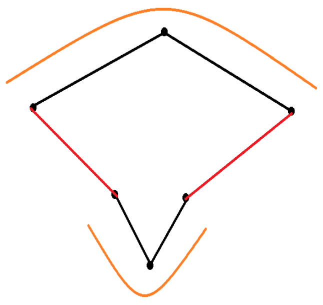
Figure 4: Example of the situation when upper and lower sub-hulls do not form a correct convex hull.

To transfer the utmost left point of the lower sub-hull to the upper sub-hull means to split the concatenable queue representing the lower sub-hull over its utmost left points and merge the obtained part with the upper sub-hull.

After combining the parts of the convex hulls, another corner case might take place. The search for the tangent for the upper parts of the hulls does not take into account the position of the lower parts and vice versa. As a result, the upper and lower parts of the final hull may not form a coherent structure. An example of such a situation is shown in Fig. 4.

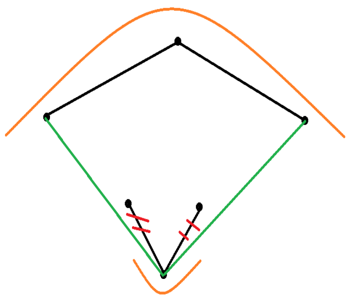
Figure 5: Correctly constructed convex hull. Here the *pivoting left* and *pivoting right* nodes are represented by the lowers points in the convex hull.

To avoid such a situation, it is necessary to perform the step of cutting off the redundant left and right parts of the formed lower sub-hull. To do so, we perform two more binary searches on the lower sub-hull to find *pivoting left* and *pivoting right* nodes. The part of the lower sub-hull in between those pivoting nodes forms the correct hull with the upper sub-hull. Fig. 5 shows example of such procedure.

2.2 "Divide-and-conquer" algorithm interface

The next goal of this work is to build a unified algorithmic environment. The construction of such an object requires the combination of an algorithmic database together with the necessary data structures. It is needed to create an interface for generic algorithms based on the "divide-and-conquer" strategy.

We first list the components of such an interface:

- Preprocessing.
- Dividing task into sub-tasks.
- Merging results of solved sub-tasks.
- Checking if a given input represents a base case.
- Solving the base case.

To construct the final interface it remains to determine the input and output types of its functions. A large number of computational geometry algorithms, such as the minimum spanning tree, the Delaunay triangulation, the Voronoi diagram, and the convex hull accept the list of points. A list can also be easily split into two parts of roughly the same size. The output type of the interface should store result data computed by the algorithmic environment. In our implementation we use a special class *Result* with *convexHull* field. The field stores convex hull computed on a specific step of the recursion.

Listing 1 shows the constructed algorithm model. Here every aforementioned component is represented as a function.

```
interface DaCAlgorithm:
  Points preprocess(Points input)
  Pair[Points, Points] divide(Points input)
  Result merge(Result first, Result second)

  boolean isBaseCase(Points input)
  Result solveBaseCase(Points input)
```

Listing 1: Algorithm model based on the "divide-and-conquer" principle. Here a list of points is denoted as Points

2.3 Sequential and parallel execution

Although this model very accurately describes the class of algorithms, it does not make it possible to solve the problem directly by having input data. This allows us to separate the implementation of the algorithm from how it is executed. Next the principles of sequential and parallel execution are discussed.

When executing sequentially an algorithm, the sub-problems are computed one by one. We first check if the current input is a base case and if so we can directly compute it by calling *solveBaseCase* procedure. Otherwise the input is split with *divide* and the obtained sub-problems are solved sequentially. Finally the obtained results are merged with *merge* procedure.

In parallel execution, we take into account that the individual sub-problems can be calculated independently, which significantly speeds up the execution of the algorithm. To construct the concurrent execution algorithm, we use the following parallel computation abstraction $computeInParallel(function1, function2)$ which runs the functions $function1$ and $function2$ simultaneously. We use it to solve sub-problems obtained after dividing a given input. Other than that parallel version is identical to the sequential one.

From the implementation standpoint, the performance of parallel execution was improved by introducing a limit on the size of sub-tasks that can be calculated in parallel. This allowed us to distribute work between threads more evenly.

3 CONCATENABLE QUEUE IMPLEMENTATION

As shown in [OL81], the concatenable queue is the key data structure for the algorithm described above and is therefore the basis for the UAEM. Now we will focus on how to efficiently implement it for our algorithmic environment.

Concatenable queue is an Abstract Data Type, that supports the following operations:

- insert();
- remove();
- getMinimum();
- contains();
- split();
- concatenate().

By default the elements in a concatenable queue are kept in a certain predefined order [AH74, pp.. 155-157]. In this article the concatenable queue is implemented as a modified balanced binary tree. Its nodes are divided into non-leaf and leaf ones. The leaf nodes contain all points kept in a tree. A *ConcatenableQueue* object maintains pointers to the root of the tree as well as leaf nodes that contain minimum and maximum values in the tree. Every node has $left$ and *right* pointers which point to its left and right child respectively. For the leaf nodes those pointers point to the left and right neighboring leaf nodes or *nil* if the node is utmost in the tree. Additionally every node keeps a pointer $leftMax$ to a node with the largest element in its left sub-tree, which allows us to perform binary search. The node constructor accepts values $leftMax$, $left$ and *right*. The *height* value each node is kept for the balancing during the split and merge operations. Now we will go into details on how this data structure is implemented.

The *contains* operations is pretty straightforward and uses binary search over the tree so its complexity is $O(\log n)$, where n hereafter denotes the numbers of nodes in the queue.

Algorithm of inserting a new element in a queue looks like as follows. First, the position for a new node is searched. Then a new node is inserted between two adjacent leaves. Going back a new non-leaf node is created - the parent for the new node and one of its neighbors. The algorithm is formally described in Listing 2. Here the *updateHeight* subroutine updates the height value for a given node, the $insertLeaf$ insets a new leaf node between two adjacent leaf nodes.

```
Node insert(int v, Node node):
  Node result = nil
  if v > node.leftMax.value:
    if !node.isLeaf:
      node.right = insert(v, node.right)
    else:
      Node newNode = insertLeaf(node, node.
          right, v)
      result = Node(node, node, newNode)
  else:
        if !node.isLeaf:
            node.left = insert(v, node.left)
        else:
            Node newNode = insertLeaf(node.left,
                    node, v)
            result = Node(newNode, newNode, node)

    if result == nil:
        result = node

    updateHeight(result)

    return result
```

Listing 2: Queue insertion algorithm

```
split(int v, Node node, ConcatenableQueue lq,
        ConcatenableQueue rq):
    if !node.isLeaf
        if v < node.leftMax.value:
            split(v, node.left, lq, rq)
            rq.root = concatenate(rq.root, node.right,
                    node.leftMax)
        else if v > node.leftMax.value:
            split(v, node.right, left, rq)
            lq.root = concatenate(node.left, lq.root,
                    node.leftMax)
        else:
            lq.root = node.left
            lq.maxNode = node.leftMax
            rq.root = node.right
            rq.minNode = node.leftMax.right
            cut(node.leftMax)
    else:
        lq.root = node
        lq.maxNode = node
        rq.minNode = node.right
        cut(node)
```

Listing 3: Queue split algorithm

In the first step we find out if $leftMax$ point to a node with smaller value than v. If so, then, if the current node is not a leaf, the search proceeds on the right sub-tree of the current node. Otherwise, a new leaf is created between the current node and its right neighbor. The case, when $leftMax$ has a greater value than v is analogous. The procedure ends with updating the height on a newly created node, which is returned as a result value.

Since in every step we perform a constant amount of work, the complexity of the procedure is $O(h) = O(\log n)$, where h hereafter denotes the height of the tree. The *remove* operation is analogous.

Now we will discuss the *split* procedure. As an input the procedure takes a value based on which the split is performed, a current node as well as left and right queues, which are constructed as a result of the procedure. The value, by which the split has been performed, belongs to the left queue. The procedure is formally described in Listing 3.

Here the *concatenate* procedure is used. It performs concatenation of two arbitrary nodes and uses their heights to balance the resulting queue. The *cut* procedure breaks connections between two adjacent leaf nodes in a queue and therefore is trivial. In the first step of the *split* operation we check if the current node is not a leaf. Is so, then the procedure continues on either left or right sub-tree. Here a special corner case is considered, where $leftMax$ contains the dividing value. If that is the case, then $leftMax$ becomes the maximum node for the left queue and its right neighboring leaf node becomes the minimum node for the right queue. Finally, if *node* is a leaf, its connections are broken and the value of *maxNode* is updated for the left queue as well as the value of *minNode* for the right queue.

```
Node concatenate(Node ln, Node rn, Node
        leftMax) {
    if ln == nil:
        return rn
    else if rn == nil:
        return ln
    else if ln.height < rn.height:
        rn.left = concatenate(ln, rn.left, leftMax)
        updateHeight(rn)
        return rn
    else if ln.height > rn.height:
        ln.right = concatenate(ln.right, rn, leftMax)
        updateHeight(ln)
        return ln
    else:
        Node result = Node(leftMax, ln, rn)
        updateHeight(result)
        return result
```

Listing 4: Algorithm of merging two queues

The algorithm of the *concatenate* procedure is described in Listing 4. First, we consider corner cases where one of the nodes is nil. This is needed to ensure the correctness of the recursion. Then, if the left node is lower than the right, one step down is taken for the right node. If the right node is lower - we take a step down for the left node. If the heights are equal, the joining point is found and a new node must be created. In each step, it is necessary to update the height of the current node because it changes.

We begin analyzing the complexity of the *split* procedure by determining the complexity of the $concatenate$ procedure. At each iteration, a step is performed either to the left son of the current node or to the right one. The execution of the recursive procedure finishes by merging two nodes.

Since each step moves us down one level and a constant amount of work is performed for each level, the total complexity of the $concatenate$ procedure is $O(h) = O(\log n)$. The $split$ procedure uses the $concatenate$ procedure as a subroutine. The complexity of a $split$ call is equal to the complexity of $concatenate$. The number of recursive $split$ calls for one such operation is $O(\log n)$, so the total complexity of the procedure $O(\log^2 n)$. The merge operation of two queues is reduced to the clamping of their root nodes by the $concatenate$ procedure, so its complexity is $O(\log n)$.

4 COMPLEXITY ANALYSIS

Theorem 1. *The complexity of the described convex hull construction algorithm for a static set of points is $O(n \log n)$ with sequential execution.*

Proof. We will argue the complexity of the algorithm by listing the complexities of the main steps.

1. Preprocessing: sorting and removal of "inner" repetitions by x and y coordinates $O(n \log n)$.
2. Divide step: splitting list of points in half $O(1)$.
3. Merge step: merging convex hulls obtained from solved sub-tasks $O(\log^2 n)$:
   (a) transfer of the utmost points to upper parts of convex hulls with at most 4 calls to $split$ and $merge$ operations $O(\log^2 n)$;
   (b) finding the tangent for the upper parts of the hulls with a binary search $O(\log n)$;
   (c) splitting and merging the upper parts with 2 calls to $split$ and 1 call to $merge$ operations $O(\log^2 n)$;
   (d) moving the utmost points to the bottom of the hulls with at most 2 calls to $split$ and $merge$ operations $O(\log^2 n)$;
   (e) finding the tangent for the lower parts of the hulls with a binary search $O(\log n)$;
   (f) splitting and merging the lower parts with 2 calls to $split$ and 1 call to $merge$ operation $O(\log^2 n)$;
   (g) normalization of the obtained lower part with 2 binary searches and 2 calls to $split$ operation $O(\log n)$.

Using known algorithms, we can perform sorting in $O(n \log n)$. To estimate the complexity of the recursive procedure for constructing a convex hull, we make the following equation:

$$T(n) = 2T(\frac{n}{2}) + O(\log^2 n) \quad (1)$$

According to result from the theory of algorithmic complexity, we have that the solution of this equation is:

$$T(n) = O(n) \quad (2)$$

Thus, taking into account the preprocessing, we get the total complexity of the algorithm $O(n \log n)$. $\square$

Theorem 2. *The complexity of the recursive convex hull construction is $O(\log^3 n)$ when executed concurrently on $\frac{n}{2}$ processors.*

Proof. The recursion tree has a height of $O(\log n)$ levels. At the lowest level, the number of sub-tasks created is $\frac{n}{2}$. Thus, each sub-task takes no more than $\frac{n}{2}$ time. Next, $O(\log^2 n)$ work is performed at each level. Having the height of the recursion tree, we get the total complexity of the algorithm. $\square$

5 PERFORMANCE EVALUATION

We implement the UAEM with Java programming language using its standard library. We used a machine running Ubuntu 18.04 LTS equipped with 16GB of DDR4/2 RAM and Intel Core i7-8750H CPU which has 6 cores and supports up to 12 threads.

To efficiently solve "divide-and-conquer" subproblems we delegate parallelization to the ForkJoinPool which is available as part of the Oracle's JDK 8. Furthermore, we limit the average number of recursive subproblems per thread. This allows us to better control load balancing for large inputs. We conducted the performance comparison of sequential execution and parallel execution with different numbers of average sub-tasks per thread. Parallel execution was done with 12 thread. All performance data is reported in Table 1.

Parallel computation allows us to achieve up to 37% performance improvement in the best case. Fine-tuning through the average number of sub-problems per thread achieves up to 8% speedup compared to the unbalanced case. Base on collected data we can conclude that the optimal number of subproblems per thread for the convex hull computation is 30.

6 CONCLUSION

We've considered in detail the process of designing and implementing the UAEM as well as the unified data structure for it. In this model a generic interface of a "divide-and-conquer" algorithm was created. This allows to execute the algorithms which are implemented according to this model both sequentially and in parallel. Apart from that a concatenable queue was implemented and served as the basis for the model described above.

Number of points	**Time (μs)**				
	sequential	20 tpth	30 tpth	40 tpth	50 tpth
$1 \cdot 10^5$	62	43	**41**	44	44
$5 \cdot 10^5$	457	309	**288**	311	308
$1 \cdot 10^6$	861	**643**	644	686	672
$5 \cdot 10^6$	3926	2876	**2860**	3005	2973
$1 \cdot 10^7$	6002	5207	**5112**	5465	5120

Table 1: Performance of the convex hull computation in the UAEM. Since our model uses fork-join parallelism we measure how the average number of recursive tasks per thread (tpth) affects the computation time. Bold numbers indicate the best time in each row.

Using the data structure allowed to significantly reduce the time and computational resources for solving the convex hull problem. The main advantages of the developed algorithm are an optimized preprocessing stage and the efficiently implemented merge step, due to the usage of concatenable queue.

The performance comparison for both types of execution shows that the algorithm has a high level of parallelism. We've achieved a speedup of 37% in the best case. It is easy to extend the functionality of the created environment either by adding new or modifying existing algorithms.

7 REFERENCES

[Ima19a] Imagej: An open platform for scientific image analysis. https://imagej.net/Welcome. Accessed: 15.04.2019.

[Ima19b] Imaris software. https://imaris.oxinst.com. Accessed: 15.04.2019.

[ACG*88] Aggarwal, A., Chazelle, B., Guibas, L., O'dunlaing, C. and Yap., C. Parallel computational ge- ometry. Algorithmica, 3(1-4):293-327, November 1988. DOI: 10.1007/BF01762120.

[AH74] Aho, Alfred V. and Hopcroft, John E. The Design and Analysis of Computer Algorithms. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1974.

[AL93] Akl, Selim G. and Lyons, Kelly A. Parallel Computational Geometry. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[AGR94] Amato, N.M., Goodrich, M.T., and Ramos, E.A. Parallel algorithms for higher-dimensional convex hulls. In Proceedings 35th Annual Symposium on Foundations of Computer Science, pages 683-694, Nov 1994. DOI: 10.1109/S-FCS.1994.365724.

[ACG89] Atallah, M.J., Cole, R. and Goodrich, M.T. Cascading divide-and-conquer: A technique for designing parallel algorithms. SIAM J. Comput., 18(3):499-532, June 1989. DOI: 10.1137/0218035.

[BSV96] Berkman, O., Schieber, B. and Vishkin, U. A fast parallel algorithm for finding the convex hull of a sorted point set. Int. J. Comput. Geometry Appl., 6:231-242, 1996.

[Che95] Chen, D.Z. Efficient geometric algorithms on the erew pram. IEEE Transactions on Parallel and Distributed Systems, 6(1):41-47, Jan 1995. DOI: 10.1109/71.363412.

[CG88] Cole, R. and Goodrich, M.T. Optimal parallel algorithms for polygon and point-set prob- lems. In Proceedings of the Fourth Annual Symposium on Computational Geometry, SCG '88, pages 201-210, New York, NY, USA, 1988. ACM. DOI: 10.1145/73393.73414.

[GJ97] Goodman, Jacob E. and O'Rourke, J., (eds.). Handbook of Discrete and Computational Geometry. CRC Press, Inc., Boca Raton, FL, USA, 1997.

[JaJ97] JaJa, J. An Introduction to Parallel Algorithms. Addison Wesley, 1997.

[OL81] Overmars, Mark H. and Leeuwen, Jan V.. Maintenance of configurations in the plane. Journal of Computer and System Sciences, 23(2):166 - 204, 1981.

[Rei93] Reif, John H. Synthesis of Parallel Algorithms. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1993.

[SSKH11] Sommer, C., Straehle, C., Koethe, U. and Hamprecht, F.A. Ilastik: Interactive learning and segmentation toolkit. In 2011 IEEE International Symposium on Biomedical Imaging: From Nano to Macro, pages 230-233, March 2011. DOI: 10.1109/ISBI.2011.5872394.

[TA10] Tereshchenko V.N. and Anisimov A.V. Recursion and parallel algorithms in geometric modeling problems. Cybernetics and Systems Analysis, 46(2):173-184, 2010.

[Lee90] Leeuwen, Jan V., (eds.). Handbook of Theoretical Computer Science (Vol. A): Algorithms and Complexity. MIT Press, Cambridge, MA, USA, 1990.