

Unified data structures for solving optimally a set of interrelated computational geometry problems

Vasyl Tereshchenko¹ and Semen Chudakov²

- 1 Faculty of computer science and cybernetics, Taras Shevchenko National University of Kyiv, Ukraine
vtereshch@gmail.com
- 2 Faculty of computer science and cybernetics, Taras Shevchenko National University of Kyiv, Ukraine
semen.chudakov7@gmail.com

Abstract

This paper is devoted to the development of an efficient algorithmic model for solving a set of interrelated computational geometry problems. To do this, a unified algorithmic environment with unified data structures is created, which allows to implement complex use cases efficiently with respect to computational resources. We build the environment on the basis of the “divide and conquer” strategy. Once a convex hull is key to a set of computational geometry problems, we offer a concatenable queue data structure to maintain it. The data structure is implemented in a form of a binary tree. This allows to perform operations needed in algorithm for a set of problems in $O(\log n)$ 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.

Lines 329

1 Introduction

Nowadays, advanced computer simulations and visualizing of complex scientific researches as well as large scale technical projects requires to simultaneously solve 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 (iLastic [15], IMARIS [2], ImageJ [1]), 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 [3, 7, 10, 6, 9, 8, 11, 5, 12, 17, 14]. Every such algorithm requires its own computational resources and is executed independently from others. In such case some identical steps, such as preprocessing and building data structures, are performed several times.

Therefore, an important objective in developing the algorithmic models is to create a universal tool, which would have means to efficiently solve a set of problems. This tool should also execute identical steps of the algorithms once and be able to represent results of those steps in a form of the unified data structures. In [16] 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, the preprocessing and splitting the initial set of data to form the recursion tree is common for all problems and is executed only once. During the merge stage intermediate results are maintained in a

36th European Workshop on Computational Geometry, Würzburg, Germany, March 16–18, 2020.
This is an extended abstract of a presentation given at EuroCG’20. It has been made public for the benefit of the community and should be considered a preprint rather than a formally reviewed paper. Thus, this work is expected to appear eventually in more final form at a conference with formal proceedings and/or in a journal.

weighted concatenable queue. This model does not repeat computations and the intermediate results are highly reused during the algorithms, 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 provide detailed explanation on how the concatenable queue which is used in the algorithm is implemented. Finally we make complexity analysis for the algorithm and evaluate its performance.

2 Unified algorithmic environment

2.1 Algorithms stages

In this section we describe the principle of how we decompose each algorithm into distinct parts. This partition is then used to avoid repeating the computations in the algorithmic environment. The principle will be show on a convex hull algorithm which is similar to the one describe in [13] but operates on a static set of points.

The notion of a convex hull is simple. For a set of points S in a k -dimensional space it is a smallest convex set, that comprise S . In practice to solve such problem, means to find a subset in S , which can be a "skeleton" for the convex hull.

In the preprocessing stage, the "inner" points, which lie on a horizontal or vertical line, are removed from the set. 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 points a_1, a_2, \dots, a_k lie on one horizontal line and $x(a_1) < x(a_2) < \dots < x(a_k)$. Then, by the criterion, the points a_2, a_3, \dots, a_{k-1} must be removed. Analogously for the vertical case.

Consider an algorithm that, in a sorted array, for each group of identical elements, deletes all but the first and the last one (if there are more than two repetitions). The idea behind the algorithm is to use two pointers technique to delete repetitions by overwriting their place with other non-repeating element. This way, additional memory usage can be reduced to a constant value. The algorithm makes one pass through the array. At each step, a constant amount of work is performed to decide whether to delete the current item. Given this, the complexity of the above algorithm is $O(n)$.

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 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 repetitions by x coordinate using the described algorithm.

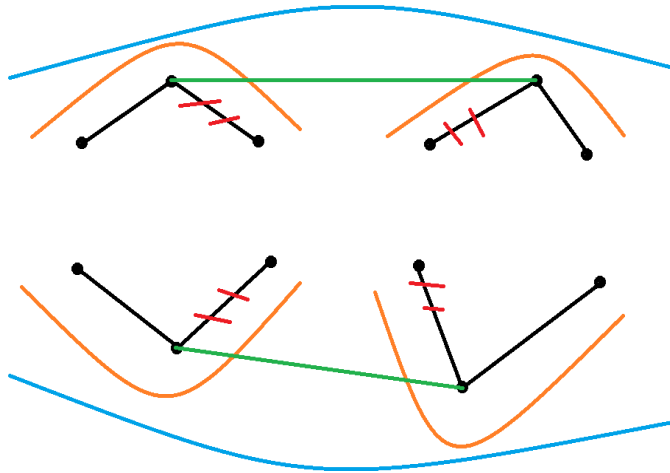
In the result we get a set of points for which we can apply the recursive convex hull algorithm.

At the stage of splitting the initial problem, the set of points is divided into left and right parts of equal size. Since a array-like data structure is used to store the points, this operations can be completed in $O(1)$ using the formula below:

$$M_{i,j} = \frac{i+j}{2} \tag{1}$$

The recursion stops when there are no more than 3 points in the set.

For the base case, the set of points can have 2 or 3 elements. In the first case the highest of two points forms the upper part of the hull, and the lower - the lower part. To consider the base case of 3 points, we introduce an additional notion of the tangent slope given by



87 ■ **Figure 1** Merging step of two hulls

76 two points on the plane. For arbitrary points a_1, a_2 such that $x(a_1) < x(a_2)$ slope is denoted
77 as λ :

$$78 \quad \lambda(a_1, a_2) = \frac{y(a_2) - y(a_1)}{x(a_2) - x(a_1)} \quad (2)$$

79 According to this definition, the possible cases for points a_1, a_2, a_3 are given in Table 1:

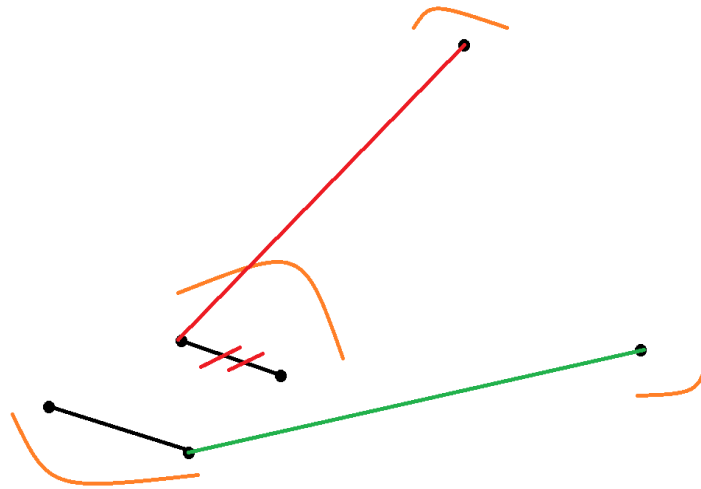
80 ■ **Table 1** 3-points base cases

$\lambda(a_1, a_2) > \lambda(a_2, a_3)$	$y(a_1) < y(a_2)$	upper part	lower part
false	false	a_1, a_3	a_2
false	true	a_1, a_3	a_2
true	false	a_2, a_3	a_1
true	true	a_1, a_2	a_3

81 In [13] an algorithm for merging two convex hulls is described. The idea of this algorithm
82 is to maintain the hull in two concatenated queues. The hull is divided into two parts. In
83 this article it is divided into the upper and the lower sub-hulls. Then, using the two-pointer
84 technique and the cases described in [13], the proper tangent line is searched. It remains to
85 split the two queues at the found vertices that form the tangent and to merge the remaining
86 parts. An example of performing such a procedure is shown in Fig. 1.

88 It remains to consider the corner cases that arise when performing the merging. The
89 first of these cases is related to the ambiguity of the position of the utmost points in the
90 described representation. The leftmost point of the left hull and the rightmost point of the
91 right hull must belong to the upper parts of the view before finding the tangent line, because
92 otherwise such tangent may be found incorrectly. An example of such incorrect search is
93 given in Fig. 2.

95 To avoid such a situation, it is necessary to move the indicated points to the upper
96 sub-hulls before merging them. For the rightmost point of the left hull and the leftmost
97 point of the right hull we have the following cases. Similarly to the previous argument, they



94 ■ **Figure 2** Example of an incorrect position of the utmost left in the left sub-hull

98 must be transferred to the upper parts of the hulls. And after merging these points must be
 99 transferred to the lower parts of the hull, if they do not belong to the resulting upper part
 100 of the final hull. Otherwise, the formed hull may be incorrect. An example of such case is
 101 shown in Fig. 3.

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

109 To avoid such a situation, it is necessary to perform the step of cutting off the redundant
 110 parts of the formed lower sub-hull. Then searching for the left and right pivoting vertices in
 111 the concatenable queue is performed. After that, the queue is split over the found vertices.
 112 Fig. 5 shows correct convex hull.

114 2.2 “Divide-and-conquer” algorithm interface

115 The next goal of this work is to build a unified algorithmic environment. The construction
 116 of such an object requires the combination of an algorithmic database together with the
 117 necessary data structures. In fact, it is needed to create an interface of generic algorithm
 118 based on “divide-and-conquer” strategy, which will then be used on a specific input data.

119 We start creating the generic interface by listing its components:

- 120 ■ Preprocessing.
- 121 ■ Splitting task into sub-tasks.
- 122 ■ Solving sub-tasks.
- 123 ■ Merging obtained results.
- 124 ■ Checking, if a given input data is a base case for the algorithm.
- 125 ■ Solving the base case.

126 Each of these components will represent a function in the future interface. Here there
 127 is a clear separation between the input type for the algorithm and the type of the result it
 128 returns. These two types should be parameters of the algorithm model. One should also pay
 129 attention to the input and output types of functions in the interface.

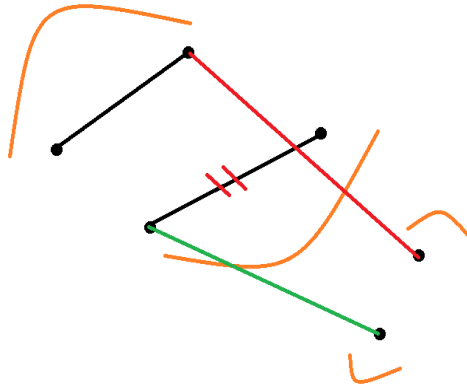


Figure 3 Example of a convex hull for a wrong position of the utmost left points of the left sub-hull

Listing 1 Algorithm model based on the “divide-and-conquer” principle

```

interface DaCAlgorithm[IT, OT]:
  boolean isBaseCase(IT input)
  int inputSize(IT input)

  OT solveBaseCase(IT input)
  IT preprocess(IT input)

  OT merge(OT first, OT second)
  Pair[IT, IT] divide(IT input)

```

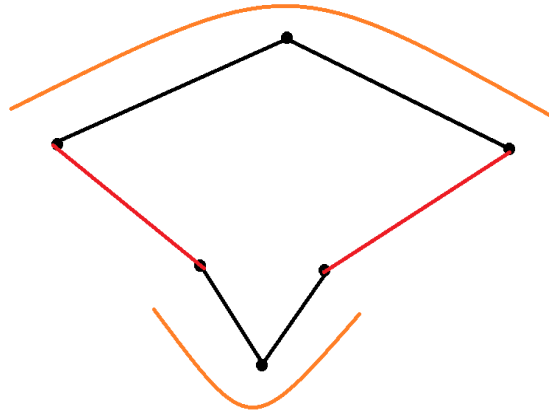
A large number of computational geometry algorithms, such as computing minimal spanning tree, the Delaunay triangulation, the Voronoi diagram and the convex hull accept the list of points. However, it would be wrong to limit the input type in the algorithm interface to a list of points. The reason for this is the fact that some algorithms can use the results of other algorithms as input data. A well known example is the construction of a Delaunay triangulation based on the Voronoi diagram maintained in a special data structure.

Listing 1 shows the constructed algorithm model. *IT* denotes the input type and *OT* - output type.

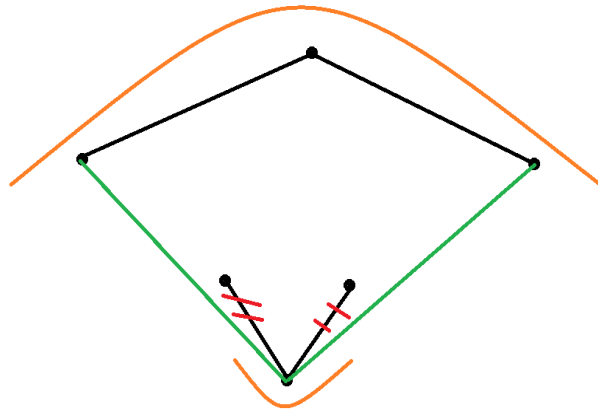
2.3 Sequential and parallel execution

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 in fact allows to divide implementation of the algorithm from how it is executed. Next the principles of sequential and parallel execution are discussed.

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



108 ■ **Figure 4** An example of a non-integral hull after merging along the reference lines



113 ■ **Figure 5** Correctly constructed convex hull

148 In parallel execution, we take into account that the individual sub-problems can be
149 calculated independently, which significantly speeds up the execution of the algorithm.

150 To construct the concurrent execution algorithm, we use the following parallel computation
151 abstraction *computeInParallel*(*function1*, *function2*). which runs the functions *function1*
152 and *function2* simultaneously. We use it to solve sub-problem obtained after splitting a
153 given input. Other than that parallel version is identical to the sequential one.

154 Practically, from the implementation standpoint performance of parallel execution was
155 improved by introducing a limit on the size of sub-tasks that can be calculated in parallel.
156 This allowed to put a threshold on the amount of work for one thread.

3 Implementation details

3.1 Concatenable queue

As shown in [13], 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 in our algorithmic environment.

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

```

■ ADD_ELEMENT();
■ REMOVE_ELEMENT();
■ GET_MINIMUM();
■ CONTAINS();
■ SPLIT();
■ MERGE().

```

By default the elements in a concatenable queue are kept in a certain predefined order [4, pp. 155-157].

In this article the concatenable queue is implemented as a binary $B+$ tree. Its vertices are divided into non-leaf and leaf ones. The leaf vertices contain all data kept in a tree. Every vertex has a pointer to its left and right child. For the leaf vertices those pointers point to the left and right neighboring leaf vertices or *null* if the vertex is utmost in the tree. Additionally every vertex keeps a pointer to a vertex with the largest element in its left sub-tree, which allows to perform binary search [4, pp. 155-157]. The height of each vertex is kept for the balancing during the split and merge operations. It is measured as a maximum amount of steps it is possible to do in order to reach a leaf vertex.

From 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 vertices in the queue.

Algorithm of inserting an element in a queue looks like as follows. First, the position of a new vertex is searched. Then the connections between adjacent leaf vertices are broken to insert a new one. Going back a new non-leaf vertex is created - the parent for the new element and one of its neighbor. The algorithm is formally described on the Listing 2.

Here the *updateHeight* subroutine updates the height value for a given vertex, the *createLeafBetween* subroutine breaks the connection between adjacent vertices to insert a new one.

On the first step we find out if *leftSubtreeMax* point to an element with greater value than the value to be inserted e . If so, then, if current element is a leaf, a new element is created between the current element and its left neighbor. Otherwise the search proceeds on the left sub-tree of the current element. The case, when *leftSubtreeMax* is smaller than the value e is analogous. The procedure ends with updating the height on a newly created element. The element is returned as its result value.

Since on 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 operation of removing an element from the queue is performed analogously.

The split operation is a bit more complex. As an input the procedure takes current element and value based on which the split is performed. As a result two independent queues are formed. The value, by which the split has been performed, belongs to the left sub-queue. The procedure is formally described on the Listing 3.

```

Node insert(Node node, int e):
    Node result = nil
    if e <= node.leftSubtreeMax.data:
        if node.isLeaf:
            if e == node.data:
                node.data = e
            else:
                Node createdLeaf = createLeafBetween(e, node.left,
                                                    node)
                result = Node(createdLeaf, createdLeaf, node)
        else:
            node.left = insert(node.left, e)
    else:
        if node.isLeaf:
            Node createdLeaf = createLeafBetween(e, node, node.right)
            result = Node(node, node, createdLeaf)
        else:
            node.right = insert(node.right, e)

    if result == nil:
        result = node

    updateHeight(result)
    return result

```

186 ■ Listing 2 Queue insertion algorithm

```

203 split(Node node, int e, ConcatenableQueue leftQueue,
204       ConcatenableQueue rightQueue):
205     if node.isLeaf
206         leftQueue.root = node
207         leftQueue.maxNode = node
208         rightQueue.minNode = node.right
209         cut(node)
210     else:
211         if e == node.leftSubtreeMax.data:
212             leftQueue.root = node.left
213             leftQueue.maxNode = node.leftSubtreeMax
214
215             rightQueue.root = node.right
216             rightQueue.minNode = node.leftSubtreeMax.right
217
218             cut(node.leftSubtreeMax)
219         else if e < node.leftSubtreeMax.data:
220             split(node.left, e, leftQueue, rightQueue)
221             rightQueue.root = concatenateNodes(rightQueue.root, node.
222                                                right, node.leftSubtreeMax)
223         else:
224             split(node.right, e, leftQueue, rightQueue)
225             leftQueue.root = concatenateNodes(node.left, leftQueue.
226                                                root, node.leftSubtreeMax)
227
228

```

229 ■ Listing 3 Queue split algorithm

Here the *concatenateNodes* procedure is used. It performs concatenation of two arbitrary elements and uses their heights to balance the resulting queue. The *cut* procedure breaks connection between two adjacent leaf elements in a queue and therefore is trivial. On the first step in the *split* operation we check, is the current element is a leaf. If so, its connection are broken and the value of *maxNode* is updated for the left queue as well as the value of *minNode* for the right queue. If element is not a leaf, then the procedure continues of either left or right sub-tree. Here a special corner cases is considered, where *leftSubtreeMax* contain the dividing value. Then the analogous action to the usual search are performed.

The algorithm of the *concatenateNodes* procedure is described on the Listing 4.

```

Node concatenateNodes(Node leftNode, Node rightNode, Node
leftSubtreeMax) {
    if leftNode == nil:
        return rightNode
    else if rightNode == nil:
        return leftNode
    else if leftNode.height == rightNode.height:
        Node result = Node(leftSubtreeMax, leftNode, rightNode)
        updateHeight(result)
        return result
    else if leftNode.height < rightNode.height:
        rightNode.left = concatenateNodes(leftNode, rightNode.left,
leftSubtreeMax)
        updateHeight(rightNode)
        return rightNode
    else:
        leftNode.right = concatenateNodes(leftNode.right, rightNode,
leftSubtreeMax)
        updateHeight(leftNode)
        return leftNode

```

Listing 4 Merging two queues

First, we consider corner cases where one of the elements is *null*. This is needed to ensure correctness of the recursion. Then, if the left element is higher than the right, one step down is taken for the left element. If the right element is higher - we take a step down for it. If the heights are equal, the joining point is found and a new element must be created. At each step, it is necessary to update the height of current element because it changes.

We begin analyzing the complexity of the *split* procedure by determining the complexity of the *concatenateNodes* procedure. At each iteration, a step is performed either to the left son of the current element or to the right one. The execution of the recursive procedure finishes by merging two elements. Since each step moves us down one level and a constant amount of work is performed for each level, the total complexity of the *concatenateNodes* procedure is $O(h) = O(\log n)$.

The *split* procedure uses the *concatenateNodes* function as a subroutine. The complexity of a *split* call is equal to the complexity of *concatenateNodes*. The number of recursive *split* calls for one such operation is $\log n$, so the total complexity of the procedure $\log^2 n$.

The merge operation of two queues is reduced to the clamping of their root vertices by the *concatenateNodes* procedure, so its complexity is $O(\log n)$.

278 4 Algorithm analysis and performance evaluation

279 4.1 Complexity

280 ► **Theorem 4.1.** *The complexity of the described convex hull construction algorithm for a*
 281 *static set of points is $O(n \log n)$ with sequential execution.*

282 **Proof.** We will argue the complexity of the algorithm by listing the complexities of the main
 283 stages it consists of.

- 284 1. Preprocessing $O(n \log n)$.
- 285 2. Recursive descent and splitting the set into 2 parts $O(1)$.
- 286 3. Recursive ascent and merging parts of the convex hull $O(\log n)$.
 - 287 a. Transfer of the utmost points to upper parts of convex hulls $O(\log n)$.
 - 288 b. Finding the tangent for the upper parts of the hulls $O(\log n)$.
 - 289 c. Splitting and merging the upper parts $O(\log^2 n)$.
 - 290 d. Moving the utmost points to the bottom of the hulls $O(\log n)$.
 - 291 e. Finding the tangent for the upper parts of the hulls $O(\log n)$.
 - 292 f. Splitting and merging the upper parts $O(\log^2 n)$.
 - 293 g. Merging the lower parts of the hull $O(\log n)$.
 - 294 h. Normalization of the obtained lower part $O(\log n)$.

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

$$297 \quad T(n) = 2T\left(\frac{n}{2}\right) + O(\log^2 n) \quad (3)$$

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

$$300 \quad T(n) = O(n) \quad (4)$$

301 Thus, taking into account the preprocessing, we get the total complexity of the algorithm
 302 $O(n \log n)$. ◀

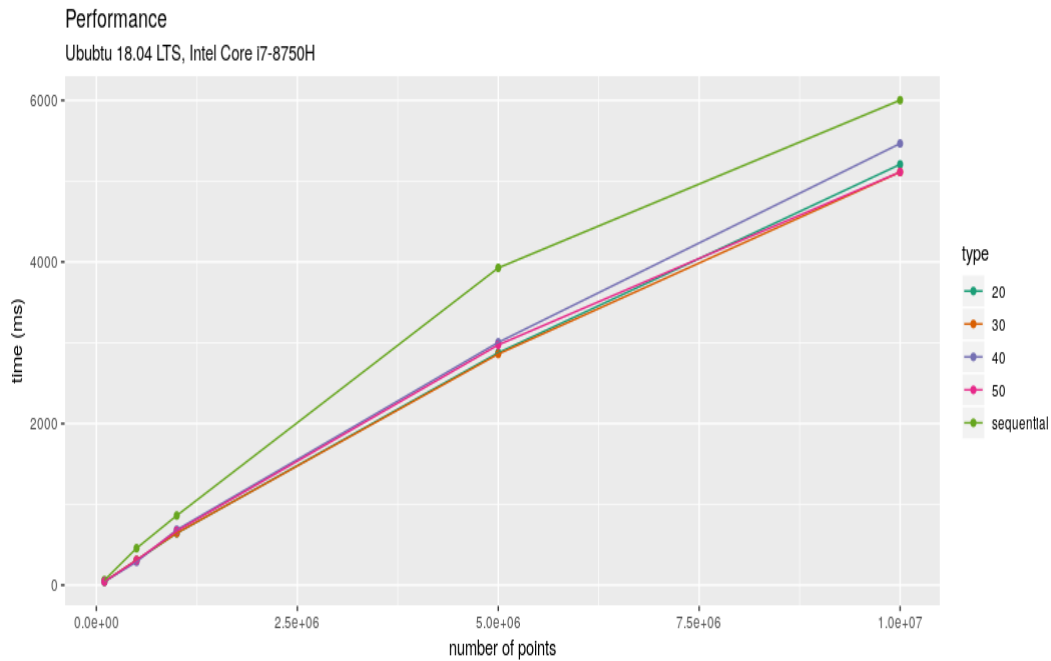
303 ► **Theorem 4.2.** *The complexity of the recursive convex hull construction is $O(\log^3 n)$ when*
 304 *executed concurrently on $\frac{n}{2}$ processors.*

305 **Proof.** The recursion tree has a height of no more than $\log n$ levels. At the lowest level, the
 306 number of sub-tasks created is $\frac{n}{2}$. Thus, each sub-task takes no more than $\frac{n}{2}$ time.

307 Next, $O(\log^2 n)$ work is performed at each level. Having the height of the recursion tree,
 308 we get the total complexity of the algorithm. ◀

309 4.2 Performance

310 A number of algorithm performance measurements were performed for different input sizes
 311 and the average number of recursive subproblems per thread. The results are shown on the
 312 Fig. 6.



313 ■ **Figure 6** Performance data

314 5 Conclusion

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

320 Using the data structure allowed to significantly reduce the time and computational
 321 resources for solving set of problems, such as constructing the convex hull. The algorithmic
 322 environment was implemented in Java programming language using its standard library.
 323 The main advantages of the developed algorithm are optimized preprocessing stage and the
 324 efficiently implemented merge step, due to the usage of concatenable queue.

325 The performance comparison for both types of execution allows to conclude that the
 326 algorithm has high level of parallelism. We've achieved speedup of 28% in the best case. It
 327 is easy to extend functionality of the created environment either by adding new or modifying
 328 existing algorithms. This flexibility is achieved by using the modular principle in its design
 329 and choosing optimal abstractions to represent algorithms.

330 References

- 331 1 Imagej: An open platform for scientific image analysis. <https://imagej.net/Welcome>.
 332 Accessed: 15.04.2019.
- 333 2 Imaris software. <https://imaris.oxinst.com>. Accessed: 15.04.2019.
- 334 3 A. Aggarwal, B. Chazelle, L. Guibas, C. Ó'dúnlaing, and C. Yap. Parallel computational ge-
 335 ometry. *Algorithmica*, 3(1-4):293–327, November 1988. URL: [https://doi.org/10.1007/](https://doi.org/10.1007/BF01762120)
 336 BF01762120, doi:10.1007/BF01762120.

- 337 **4** Alfred V. Aho and John E. Hopcroft. *The Design and Analysis of Computer Algorithms*.
338 Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1974.
- 339 **5** Selim G. Akl and Kelly A. Lyons. *Parallel Computational Geometry*. Prentice-Hall, Inc.,
340 Upper Saddle River, NJ, USA, 1993.
- 341 **6** N. M. Amato, M. T. Goodrich, and E. A. Ramos. Parallel algorithms for higher-dimensional
342 convex hulls. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*,
343 pages 683–694, Nov 1994. doi:10.1109/SFCS.1994.365724.
- 344 **7** M. J. Atallah, R. Cole, and M. T. Goodrich. Cascading divide-and-conquer: A technique
345 for designing parallel algorithms. *SIAM J. Comput.*, 18(3):499–532, June 1989. URL:
346 <http://dx.doi.org/10.1137/0218035>, doi:10.1137/0218035.
- 347 **8** Omer Berkman, Baruch Schieber, and Uzi Vishkin. A fast parallel algorithm for finding
348 the convex hull of a sorted point set. *Int. J. Comput. Geometry Appl.*, 6:231–242, 1996.
- 349 **9** D. Z. Chen. Efficient geometric algorithms on the erew pram. *IEEE Transactions on*
350 *Parallel and Distributed Systems*, 6(1):41–47, Jan 1995. doi:10.1109/71.363412.
- 351 **10** R. Cole and M. T. Goodrich. Optimal parallel algorithms for polygon and point-set prob-
352 lems. In *Proceedings of the Fourth Annual Symposium on Computational Geometry*, SCG
353 ’88, pages 201–210, New York, NY, USA, 1988. ACM. URL: [http://doi.acm.org/10.](http://doi.acm.org/10.1145/73393.73414)
354 1145/73393.73414, doi:10.1145/73393.73414.
- 355 **11** Jacob E. Goodman and Joseph O’Rourke, editors. *Handbook of Discrete and Computational*
356 *Geometry*. CRC Press, Inc., Boca Raton, FL, USA, 1997.
- 357 **12** J. JaJa. *An Introduction to Parallel Algorithms*. Addison Wesley, 1997. URL: [https:](https://books.google.com.ua/books?id=9BpYtwAACAAJ)
358 [//books.google.com.ua/books?id=9BpYtwAACAAJ](https://books.google.com.ua/books?id=9BpYtwAACAAJ).
- 359 **13** Mark H. Overmars and Jan van Leeuwen. Maintenance of configurations in the plane.
360 *Journal of Computer and System Sciences*, 23(2):166 – 204, 1981. URL: [http://www.](http://www.sciencedirect.com/science/article/pii/002200008190012X)
361 [sciencedirect.com/science/article/pii/002200008190012X](http://www.sciencedirect.com/science/article/pii/002200008190012X), doi:[https://doi.org/](https://doi.org/10.1016/0022-0000(81)90012-X)
362 10.1016/0022-0000(81)90012-X.
- 363 **14** John H. Reif. *Synthesis of Parallel Algorithms*. Morgan Kaufmann Publishers Inc., San
364 Francisco, CA, USA, 1st edition, 1993.
- 365 **15** C. Sommer, C. Straehle, U. Köthe, and F. A. Hamprecht. Ilastik: Interactive learning
366 and segmentation toolkit. In *2011 IEEE International Symposium on Biomedical Imaging:*
367 *From Nano to Macro*, pages 230–233, March 2011. doi:10.1109/ISBI.2011.5872394.
- 368 **16** V. N. Tereshchenko and A. V. Anisimov. Recursion and parallel algorithms in geometric
369 modeling problems. *Cybernetics and Systems Analysis*, 46(2):173–184, 2010.
- 370 **17** Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science (Vol. A): Algorithms*
371 *and Complexity*. MIT Press, Cambridge, MA, USA, 1990.