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

Vasyl Tereshchenko[1] and Semen Chudakov[2]

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 and 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.

Lines 171

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. 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 weighted concatenable queue. This model does not repeat computations and the intermediate results are highly reused during the algorithms, which yields good performance.

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.

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 $a_1, a_2, ..., a_k$ lie on one horizontal line and $x(a_1) < x(a_2) < ... < x(a_k)$. Then, by the criterion, the points $a_2, a_3, ..., a_{k-1}$ 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. The idea behind the algorithm is to use two pointers technique to delete repetitions by overwriting their place with other non-repeating element. At each step, constant work is performed to decide whether to overwrite the current item. Given this, the complexity of the above algorithm is $O(n)$.

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 done in $O(1)$ by computing middle positiong in the array.

The recursion stops when there are 2 or 3 elements in the array. First case is triial. To consider the base case of 3 points, we introduce a notion of the tangent slope given by two points on the plane. For arbitrary points a_1, a_2 such that $x(a_1) < x(a_2)$ slope is denoted as λ:

$$\lambda(a_1, a_2) = \frac{y(a_2) - y(a_1)}{x(a_2) - x(a_1)} \tag{1}$$

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

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

In [13] an algorithm for merging two convex hulls is described. It remains to 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 points in the described representation. The leftmost point of the left hull and the rightmost point of the right hull must belong to the upper parts of the view before finding the tangent line, because otherwise such tangent may be found incorrectly. An example of such incorrect search is given in Fig. 1.

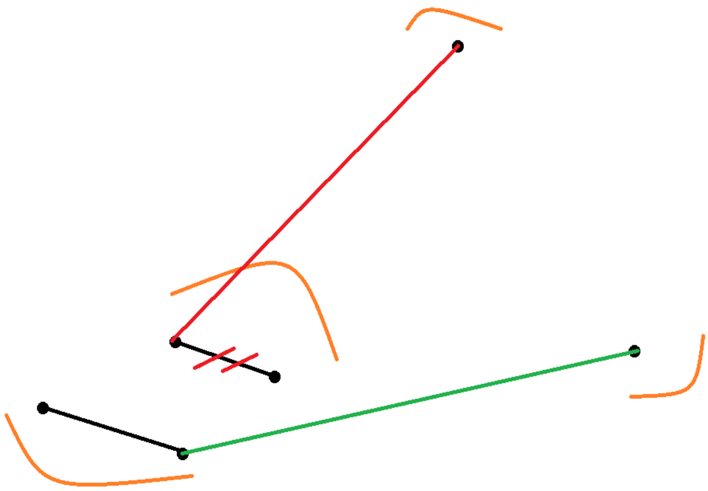

Figure 1 Example of an incorrect position of the utmost left in the left sub-hull

To avoid such a situation, it is necessary to move the indicated 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 case is shown in Fig. 2.

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. 3.

To avoid such a situation, it is necessary to perform the step of cutting off the redundant parts of the formed lower sub-hull. Then searching for the left and right pivoting vertices in the concatenable queue is performed. After that, the queue is split over the found vertices. Fig. 4 shows correct convex hull.

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. In fact, it is needed to create an interface of generic algorithm based on "divide-and-conquer" strategy, which will then be used on a specific input data.

The proposed generic interface for the "divide-and-conquer" algorithm is described on the Listing 1.

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.

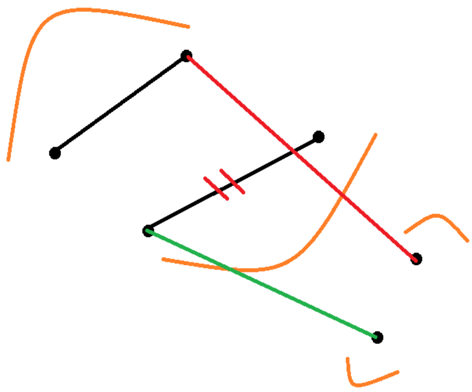

Figure 2 Example of a convex hull for a wrong position of the utmost left points of the left sub-hull

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)
```

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.

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-problem obtained after splitting a given input. Other than that parallel version is identical to the sequential one.

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

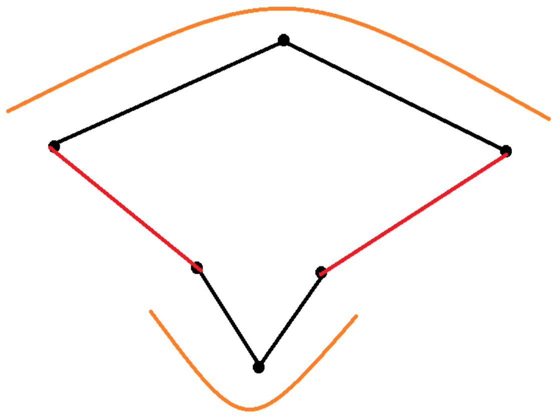

Figure 3 An example of a non-integral hull after merging along the reference lines

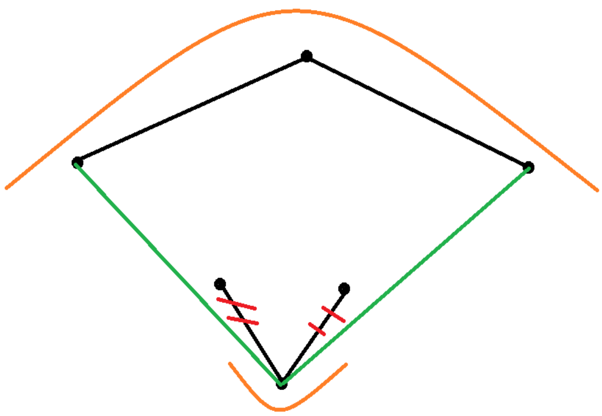

Figure 4 Correctly constructed convex hull

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.

Concatenable queue is an Abstract Data Type. Its operations are shown on Table 2 together with theirs time complexities in our implementation:

Table 2 Concatebable queue operations

Operation	Complexity
ADD_ELEMENT()	$O(\log n)$
REMOVE_ELEMENT()	$O(\log n)$
GET_MINIMUM()	$O(\log n)$
CONTAINS()	$O(\log n)$
SPLIT()	$O(\log^2 n)$
MERGE()	$O(\log n)$

4 Algorithm analysis and performance evaluation

4.1 Complexity

▶ **Theorem 4.1.** *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 stages it consists of.

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

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:

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

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

$$T(n) = O(n) \tag{3}$$

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

▶ **Theorem 4.2.** *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 no more than $\log n$ 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. ◀

4.2 Performance

A number of algorithm performance measurements were performed for different input sizes and the average number of recursive subproblems per thread. The results are shown on the Fig. 5.

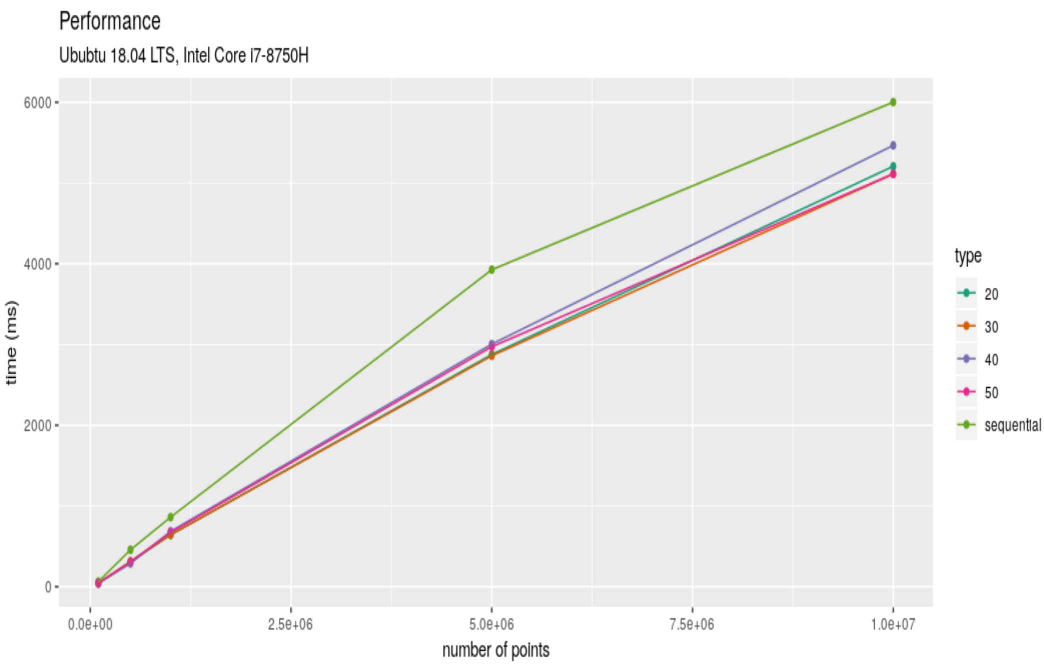

Figure 5 Performance data

5 Conclusion

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

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

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

References

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

2 Imaris software. https://imaris.oxinst.com. Accessed: 15.04.2019.

3 A. Aggarwal, B. Chazelle, L. Guibas, C. Ó'dúnlaing, and C. Yap. Parallel computational geometry. *Algorithmica*, 3(1-4):293–327, November 1988. URL: https://doi.org/10.1007/BF01762120, doi:10.1007/BF01762120.

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

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

6 N. M. Amato, M. T. Goodrich, and E. A. Ramos. 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/SFCS.1994.365724.

7 M. J. Atallah, R. Cole, and M. T. Goodrich. Cascading divide-and-conquer: A technique for designing parallel algorithms. *SIAM J. Comput.*, 18(3):499–532, June 1989. URL: http://dx.doi.org/10.1137/0218035, doi:10.1137/0218035.

8 Omer Berkman, Baruch Schieber, and Uzi Vishkin. A fast parallel algorithm for finding the convex hull of a sorted point set. *Int. J. Comput. Geometry Appl.*, 6:231–242, 1996.

9 D. Z. Chen. 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.

10 R. Cole and M. T. Goodrich. Optimal parallel algorithms for polygon and point-set problems. In *Proceedings of the Fourth Annual Symposium on Computational Geometry*, SCG '88, pages 201–210, New York, NY, USA, 1988. ACM. URL: http://doi.acm.org/10.1145/73393.73414, doi:10.1145/73393.73414.

11 Jacob E. Goodman and Joseph O'Rourke, editors. *Handbook of Discrete and Computational Geometry.* CRC Press, Inc., Boca Raton, FL, USA, 1997.

12 J. JaJa. *An Introduction to Parallel Algorithms.* Addison Wesley, 1997. URL: https://books.google.com.ua/books?id=9BpYtwAACAAJ.

13 Mark H. Overmars and Jan van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23(2):166 – 204, 1981. URL: http://www.sciencedirect.com/science/article/pii/002200008190012X, doi:https://doi.org/10.1016/0022-0000(81)90012-X.

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

15 C. Sommer, C. Straehle, U. Köthe, and F. A. Hamprecht. 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.

16 V. N. Tereshchenko and A. V. Anisimov. Recursion and parallel algorithms in geometric modeling problems. *Cybernetics and Systems Analysis*, 46(2):173–184, 2010.

17 Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science (Vol. A): Algorithms and Complexity.* MIT Press, Cambridge, MA, USA, 1990.