An approach to solving complex problems of computational geometry

Vasyl Tereshchenko, Igor Bujak, Andrey Fisunenko

Taras Shevchenko National University of Kyiv, Ukraine

vtereshch@gmail.com, zerg28@gmail.com, alf.via@gmail.com

Abstract. *The problem of complex tasks related to set of points on a plane with help of a generalized and optimized algorithm is considered. The algorithm is easily parallelized. Theoretical analysis has been carried out and practical results have been obtained.*

Keywords

Unified algorithmic platform, complex tasks, parallel-and-recursive algorithm, computational geometry, weighted concatenable queue.

1 Introduction

At the current moment there exists plethora of algorithms for solving basic tasks of computational geometry. Moreover for solving the same task there may be several algorithms having optimal execution time. At the same time for solving a complex of tasks there is an opportunity to build a generalized algorithm which uses a common solving conception for all tasks.

Visual modeling and other adjacent domains often require solving a complex of tasks for the same set of input data (e.g. for a set of points in Euclidian space). Typically the set of input data is processed by each algorithm of the complex sequentially. Surely while using the most optimal algorithms for each of the tasks separately we can gain effective enough execution performance. But from other side we can select such algorithms for different tasks of the complex which uses unified approach and common data structures. In that case when algorithms have something in common then intermediate results of one algorithm can be reused in others skipping repeating computation. Furthermore there are instances of algorithms which allow performing fast transformation of their outputs to final result of another algorithm achieving additional overall performance increase. In such case a task of constructing specific data structure enabling fast and convenient work of a definite set of algorithms appears.

Computational power of computers is increasing and one of factors of this growth is multi-core and multi-processor architectures. Many of sequential algorithms have reached their theoretical performance limits. This motivates applying of parallel algorithms for tasks solving. In other words, there are conditions which do not only promote applying parallelization but make parallel algorithms quite necessary to justify this increase of computational power as sensible and meaningful.

By now many effective parallel algorithms for solving tasks of computational geometry have already been developed. In particular many of them are described in details in the capacious work [1] containing many references to other authors' results. One can find a description of known methods of constructing convex hull and Voronoi diagram in [2-7]. The task of building a unified parallel algorithm is presented in works [8-12].

The goal of the suggested research is building of a unified algorithmic platform for solving complex tasks of computational geometry.

2 Formulation of the Problem and Solution Method

Problem Let S be a set of N points in the space R^2. A unified effective parallel algorithm for solving complex of tasks operating under the set S has to be developed with lower complexity estimation $\Omega(NlogN)$ (for single processor computer). An implementation has to provide capability of including other different algorithms for solving tasks of the computational geometry.

2.1 Algorithmic models

The algorithmic model of existing systems. If some set of tasks of computational geometry has to be solved then typically a set of separate algorithms is used. Each of algorithms uses its own input and output data structures. The generalized scheme of computations for such approach can be represented as shown on Fig. 1. In the worst case these tasks are executed sequentially. For each task one can distinguish the following stages of execution:

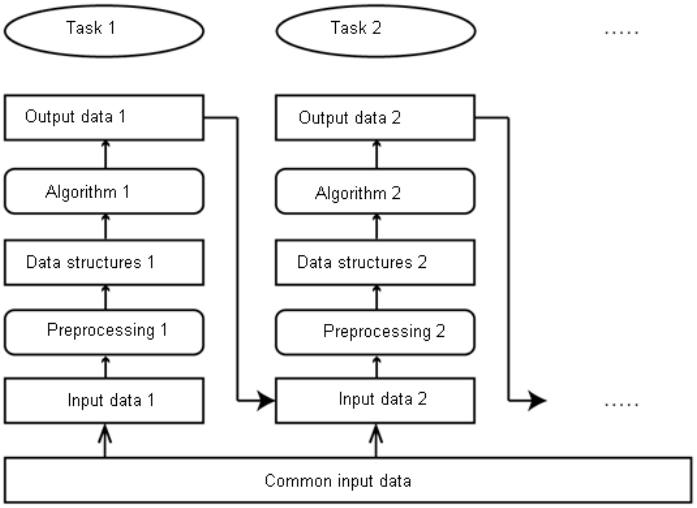

Figure 1. The computational scheme of existing systems for solving a complex of tasks.

1. Preliminary processing – preparing input data and internal data structures which are necessary for work of an algorithm solving the task;
2. Execution of the algorithm – direct solving of target task;
3. Transforming output data for transferring them to the next algorithm if it is necessary.

Such approach obviously creates additional inconveniences for use. In this case the problem of transforming output data of certain algorithms to input data of other algorithms needs to be solved additionally. Moreover extra memory increase appears inevitably as far as for solving each task a separate data structure needs to be supported. It is worth noting that in the general case parallel implementation of such approach is possible on the level of separate tasks but not a system as a whole.

The algorithmic model of the suggested approach. Unlike the described classical approach we decided to create system having single universal data structure unified for all tasks. Thus a system of visual modeling is exempted of the primary redundancy – separate data structures for each task. Hence expenses for used memory decrease. The necessity in preliminary processing procedures for each algorithm in a complex disappears: only one procedure which prepares initial data structure is needed. This data structure is used for all tasks solving. This leads to execution time reducing. A system should be built such way which enables parallelism on the whole system's level in addition to parallelism on the tasks level. For achieving established goals we decided to use "divide-and-conquer" strategy and concatenable queues as the universal data structure [12]. Thus, we have the following computational scheme presented on Fig 2. In this way our scheme can conventionally be divided into the following stages: *preliminary processing, recursive partitioning (recursive descent),merging subsets' results (recursive ascent).*

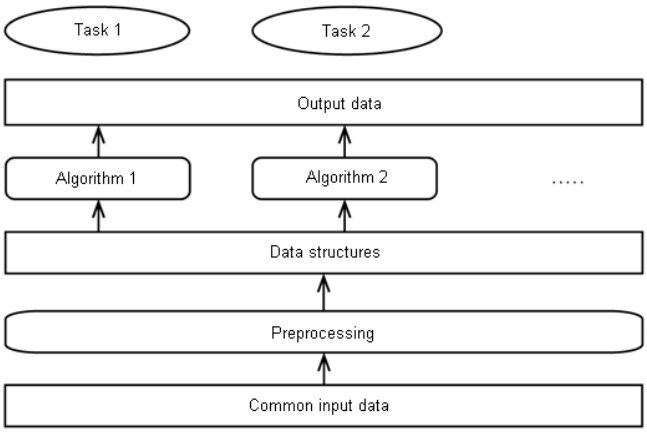

Figure 2. The computational scheme of the suggested system for solving a complex of tasks.

2.2 The algorithmic platform for solving a complex of tasks

Preliminary processing. As the input we have a set containing N different points. Two sets of points are built: sorted by x and y coordinates. It is known that complexity of sorting on single processor computer is $\Omega(NlogN)$. In our case we can use parallel algorithms of sorting with complexity $O(logN)$ [13].

Recursive partitioning. The input of this stage is ordered sets of points and as an output we have to obtain AVL-tree which satisfies the following conditions:

1) an order of points during traversal of the tree from left to right must be the same as in the set of input points;

2)each node of the tree must be balanced by the heights of its two sub-trees; difference of their heights must not exceed 1.

Such tree is built by the following algorithm:

1) build tree for one point, constructing one node and storing this point in it;

2) build tree for several points; divide this set of point into two sets which contain the same number of elements (if total number is odd than left side contains one point more). Build node in which the left child is a tree built for the left sub-set and right child is built for the right sub-set. This stage can be parallelized by executing recursive functions on separate processors. The obtained tree is balanced due to partitioning it into two sub-sets containing equal number of elements.

Merging results. During merging results for sub-sets two instances of data structures are merged into one. The chosen data structure provides merging in time $\Omega(N)$. Thus total time of algorithm execution is $\Omega(NlogN)$ for a single processor computer.

Interrelation of algorithms. Let us consider a spectrum of tasks for a set of points on the plane which can be solved with "divide-and-conquer" strategy in time $\Omega(NlogN)$ for a single processor computer. These are such tasks: convex hull; Voronoi diagram; Delaunay triangulation; nearest pair; all nearest neighbors. Interrelation between all intermediate results and input data of algorithms are shown on Fig 3.

The built implementation allows inserting other algorithms for sets of points which can effectively work using "divide-and-conquer" strategy. Taking into account that all algorithms have common parts of preliminary processing and recursive descend it was decided to subdivide a mechanism of "divide-and-conquer" to a separate procedure containing balanced binary tree and implementing the following scheme:

1. Check if the input set is not trivial;
2. If this is trivial case then execute algorithm for trivial case (e.g. convex hull for 1 point is this point); else goto 3;
3. Apply algorithm recursively for left and right subsets;
4. Merge results of left and right subsets;
5. Go to the next stage of recursive merge.

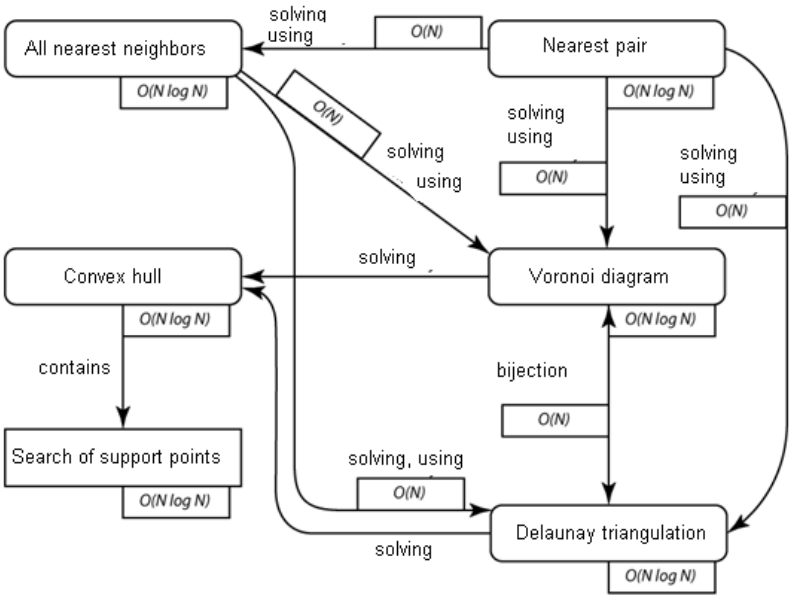

Figure 3. Interrelation of Basic Algorithms.

3 Peculiarities of Parallel Implementation

For parallel implementation the standardized library MPI was used. In particular, it gives flexibility of use and enables cross-platform development. Different platforms suggest various means for thread creation and control. Such specific operations as threads creation, data transfer between them were delegated to MPI which exists for many platforms including clusters and supercomputers. The main mean for data transfer in MPI is event passing. In general case number of threads is limited by computational environment only. On practice as soon as complexity of the algorithm is low $\Omega(NlogN)$ much time is spent on passing events between processors. That is why if an input data set is small then such parallelization can only increase execution time. In the general case time spent on event passing can be estimated as the following:

$$t(S) = t_const + k * |S|, \tag{1}$$

where S – set of transferred data (in our case this is solution for a task for some subset of points) t_{const}- some constant delay caused by network transfer (in the case of cluster) or by an operation system (in case of a processes running on a same computer); k – some coefficient which defines time for packing and unpacking data and network or computer bus performance.

As it was said above we can parallelize computations on input data partitioning stage with help of recursive descent. One can count that time spent on computations of tasks for two sets of points is approximately the same (sets are not exactly the same but can differ on number of recursions). Hence one can conclude that number of processors should be 2^n, where n – is certain number. While moving down to recursion on its depth n, computations for 2^n processors are run. Each of processors computes tasks on a set with approximate power $\frac{N}{2^n}$ of points, where N – is total number of input points. The generalized scheme of a parallel computational process is shown on Fig. 4.

Sure, we can use any number of processes which is not power of 2. Let total number of processors be 2^{n+p}, where $2^{n+p} < 2^{(n+1)}$. Under such conditions a part processes will solve task for set of points with power $\frac{N}{2^n}$, while another part of processes will solve task for set of power $\frac{N}{2^n+1}$. Such redundancy will not result in performance increase of algorithm. Thus, execution time for 2^{n+p} processes will be comparable with execution time for 2^n processes. It is worth mentioning that for using 2^{n+p} processes a logic of parallelization has to be complicated that is why namely 2^n processes were selected for the implementation. Theoretically it is possible to obtain acceleration on 2^{n+p} processes. For this one should find such number q, which satisfies $p = c * 2^q$. Parallel execution should be run down to depth of recursion $\lceil logN \rceil - q$. With this subtasks are solved for sets of points having power $\frac{N}{2^q}$. The total number of such tasks is $2^{\lceil logN \rceil - q}$. Thus, $2^{\lceil logN \rceil - q}$ sub-tasks are solved on 2^{n+p} processes. The total number of parallel executions is:

$$\lceil 2^{\lceil logN \rceil - q}/(2^n + p) \rceil = \lceil 2^{\lceil logN \rceil - q}/(2^n + c * 2^q) \rceil = \lceil 2^{\lceil logN \rceil - q}/(2^q * 2^{n-q} + c * 2^q) \rceil = \lceil 2^{\lceil logN \rceil - q}/(2^q(*2^{n-q} + c)) \rceil = \lceil 2^{\lceil logN \rceil - 2q}/(2^{n-q} + c) \rceil$$

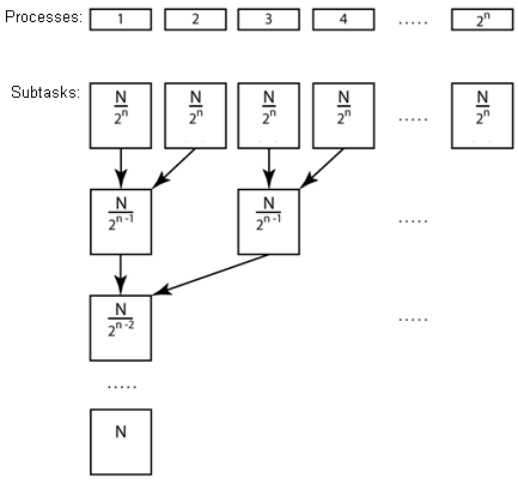

Figure 4. The generalized scheme of a parallel computational process.

As far as $0 < c < 2^{n-1}$, then:

$$2^{\lceil logN \rceil - n - 2q + min(1,q)} \leq \lceil 2^{\lceil logN \rceil - 2q} / (2^{n-q} + c) \rceil < 2^{\lceil logN \rceil - n - q}$$

This way we solve tasks of $\lceil logN \rceil - q$ of recursion depth in time which is less than $2^{\lceil logN \rceil - n - q} * \Omega(\frac{N}{2^q} * log\frac{N}{2^q})$. In the previous case when we use 2^n processes this time is equal to $2^{\lceil logN \rceil - n} * \Omega(\frac{N}{2^q} * log\frac{N}{2^q})$. As soon as $q \geq 0$ we can achieve increase of performance.

It is worth noting that such algorithm can lead to significant increase of number of messages which have to be transferred between processes and to complication of a parallelized algorithm's logic. From other side we can gain some performance increase if number of processes is greater than number of processor cores which are used for computation. In this case correct distribution of workload is delegated to an operation system. Modern operation systems can perform it pretty well. Thus for obtaining maximum performance one should run parallel computations on $2^{\lceil logu \rceil}$ processes, where u - number of cores on a computer or a cluster, taking into account that all cores have the same characteristics.

4 Testing performance of parallel algorithm

For testing different point data sets were generated randomly. The following run-time computer configuration was used: Intel Core i5 760 3.8 GHz, RAM 12 Gb; Windows 7 Professional 64. Algorithm was implemented using C++ compiler included to Microsoft Visual Studio 2010 with x64 option enabled to work with big data sets. MPI version which was used: MPI: HPC Pack 2008 R2 MS-MPI SP3. All auxiliary operations (like file operations) were excluded from time counting. Only meaningful algorithm time was measured. The graph of dependency of execution time on number of points for the sequential variant is shown on Fig. 5 a. The diagram of performance gain for MPI processes use is presented on Fig. 5b. As one can see complexity of the algorithm is actually $\Omega(NlogN)$ and the developed parallel implementation really accelerates process of task solving.

5 Conclusions

The conducted research has shown that building unified algorithmic platform for basic tasks of computational geometry gives possibility to simplify and to speed up solving of complex of tasks for a same set of points. Taking into account that the most of tasks of computational geometry are connected to processing of point sets the suggested approach can serve as a common methodology for wide spectrum of tasks for this discipline. The stage of partitioning suggested in the work can be applied not only for tasks of computational geometry having point set as an input data but can be generalized for solving tasks of other types. Common merging stages can be unified for other classes of tasks of computational geometry. The only change is specific merge procedures which reflect essence of the concrete task. It was also shown that on practice algorithm is well parallelized and has good performance marks for a parallel environment.

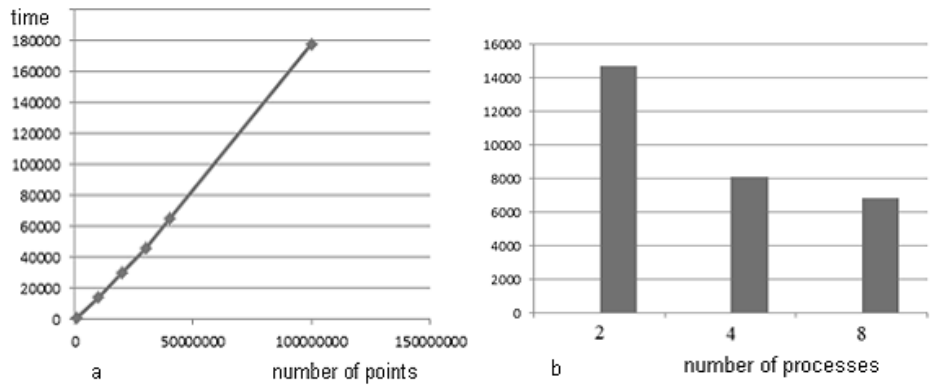

Figure 5. a)The dependency of Execution Time on Number of Points for the Sequential Variant; b)the diagram of performance gain depending on number of processors.

References

[1] B. Aggarwal, B. Chazelle, C. O'Dunlaing, C. K. Yap: Parallel computational geometry. *Algorithmica*, 3:293-327, 1988.

[2] Atallah M.J., Cole R., Goodrich M.T. Cascading divide-and-conquer: A technique for designing parallel algorithms. *SIAM J. Comput.* , 18: 499-532, 1989.

[3] Cole R., Goodrich M.T.: Optimal parallel algorithms for polygon and point-set problems. *Algorithmica*,7: 3-23, 1992.

[4] Amato N.M., Goodrich M.T., Ramos E.A.: Parallel algorithms for higher-dimensional convex hulls. *In Proc.: 35th Annu. IEEE Sympos. Found. Comput. Sci.*, pp. 683-694, 1994.

[5] Chen D. : Efficient geometric algorithms on the EREW PRAM. IEEE Trans. Parallel Distrib. Syst., 6, pp. 41-47 (1995).

[6] Berkman O., Schieber B., Vishkin U.: A fast parallel algorithm for finding the convex hull of a sorted point set. *Internat. J. Comput. Geom. Appl.*, 6:231–242,1996.

[7] Goodman J.E., O'Rourke J.: Handbook of Discrete and Computational Geometry. *N. Y.: Chapman and Hall/CRC Press*, 2004.

[8] Akl S.G., Lyons K.A.: Parallel Computational Geometry. *Englewood Cliffs: Prentice-Hall*, 1993.

[9] JaJa J.: An Introduction to Parallel Algorithms. *Amsterdam:Addison-Wesley*, 1992.

[10] Van Leeuwen J.: Handbook of Theoretical Computer Science. *Amsterdam: Elsevier/The MIT Press*, 1990.

[11] Reif J. H.: Synthesis of Parallel Algorithms. *San Mateo:Morgan Kaufmann*, 1993.

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

[13] Cole R.: Parallel merge sort. *In Proc.: 27th IEEE FOCS Symposium*, pp. 511-516, 1986.