Parallel Multi-Core Geometric Algorithms in CGAL

Vicente H. F. Batista* David L. Millman† Sylvain Pion‡ Johannes Singler§

Abstract

We describe an approach to efficiently use multiple processing cores and shared memory for several geometric algorithms. The d-dimensional algorithms we target are (a) spatial sorting of points, (b) kd-tree construction, (c) axis-aligned box intersection computation, and finally (d) bulk insertion of points in Delaunay triangulations. Underlying these comes also a thread-safe, efficient and memory-wise compact container for storing dynamic geometric data structures. This work has been implemented in the framework of CGAL `http://cgal.org/`. We also show experimental results for these algorithms in the 3D case. This work is hopefully a step towards a *parallel mode* for CGAL, where algorithms automatically use the available parallel resources without requiring significant user intervention.

1 Introduction

Many application domains require computation on geometric objects. There are for example meshes used in medical images visualization, geological data representation, fluid dynamics simulations, geographic information systems, etc. More generally, one can approach geometric computing as finding some proximity relations between objects like points. The result of such computations can usually be represented as a graph, such as a triangulation whose vertices correspond to points and edges, faces or tetrahedra (simplices in general dimension) are also nodes of the graph encoding the incidence relations.

Many core geometric algorithms and data structures have been devised over the past decades, and many proved to be useful in more than one application domain. The Computational Geometry Algorithms Library, CGAL, is a project that aims at gathering efficient, easily adaptable, and robust implementations of such useful core algorithms. The CGAL library is now a large (almost 1 million lines of code total), mature (12 years old, and commercialized since 6 years) and reliable code base. It is written in C++, which helps on the efficiency aspects as well as for building abstractions. The main design guidelines are genericity, robustness and efficiency. The use of generic programming, the fundamental paradigm behind the C++ Standard Template Library (STL), helps in producing data structures and algorithms that are re-usable and adaptable to several particular applied problems. Geometric algorithms are particularly sensitive to numerical roundoff errors that violate mathematical invariants. These problems are efficiently dealt with in CGAL by the careful use of certified arithmetic operations, using techniques based on multiple precision, interval arithmetic and static error propagation analysis.

Nowadays, one cannot think about practical efficiency without wondering how to make good use of parallelism. Parallel computing comes in several flavors: from SIMD vector units, to GPGPUs, to multiple processor cores using shared memory, and finally to distributed systems and the grids in cloud computing. A core geometric data structure like a triangulation is not naturally amenable to vectorization. Indeed, the corresponding 1D object is a linked list, not an array. Moreover, in 1D, a sorted list provides some guarantees of locality, which is not the case for higher dimensional triangulations. For example, the Delaunay triangulation of points sampled on the surface of an object will contain tetrahedra which connect points from several remote sheets of the surface which are visible from each other. So, geometric computing has some

*Universidade Federal do Rio de Janeiro/COPPE, Brazil. Email: `helano@coc.ufrj.br`
†University of North Carolina at Chapel Hill, USA. Email: `dave@cs.unc.edu`
‡GEOMETRICA project-team, INRIA Sophia-Antipolis, France. Email: `Sylvain.Pion@sophia.inria.fr`
§Universität Karlsruhe, Germany. Email: `singler@ira.uka.de`

locality aspects, but there are no a priori rules for this which can be used to split a computation among independant computing units. This means that some inter-processor communication is required to achieve good parallelism for geometric computation, and one goal is of course going to be to reduce the communication needs as much as possible to achieve performance. Multiple core computers, which provide shared memory and efficient atomic operations, are a good platform to start *parallelizing* core geometric algorithms.

In this paper, we investigate the parallelization of the following d-dimensional algorithms: (a) spatial sorting of points, as is typically used for preprocessing during incremental algorithms, (b) kd-tree construction, (c) axis-aligned box intersection computation, and finally (d) bulk insertion of points in Delaunay triangulations for mesh generation algorithms or simply computing Delaunay triangulations. For the sake of getting conclusive outcomes, we decided to base our work upon CGAL, which already provides mature codes that are among the most efficient for several geometric algorithms [6]. Indeed, when evaluating achieved speedups of parallel algorithms, care must be taken as to take an efficient sequential implementation as a base for comparison[1].

The paper is organized as follows. Section 2 describes our hardware and software platform for reference. We begin in Section 3 with the description of the compact container that we use for storing geometric graphs. Sections 4, 5, 6 and 7 then describe our parallel algorithms and experimental results for (a), (b), (c) and (d) respectively. Note that a more detailed description of these algorithms can be found in [2].

2 Platform

OpenMP We opted for OpenMP, which is implemented by almost all up-to-date compilers. The OpenMP specification in version 3.0 includes the `#pragma omp task` construct. This creates a *task*, a code block executed asynchronously, that can be nested recursively. The enclosing region may wait for all direct children tasks to finish using `#pragma omp taskwait`. A `#pragma omp parallel` region at the top level provides a user specified number of threads to process the tasks.

Libstdc++ parallel mode The C++ STL implementation distributed with the GCC features a so-called *parallel mode* [7] based on [8]. It provides parallel versions of many STL algorithms. We use some of these algorithmic building blocks, such as `partition`, `nth_element` and `random_shuffle`[2].

Evaluation system We evaluated the performance of our algorithms on a two AMD Opteron 2350 quad-core processors machine at 2 GHz and 16 GB of RAM. We used GCC 4.3 and 4.4 (prerelease, for the algorithms using the task construct), enabling optimization (`-O2` and `-DNDEBUG`).

3 Thread-safe compact container

CGAL currently provides a sequential version of a compact container, which is an STL container-like data structure allowing: iteration over its elements, addition and removal of elements in amortized constant time, stability of the iterators under these operations, and good memory locality (elements added consecutively have good probability of being not far away in memory). This data structure is used to store graphs such as Delaunay triangulations, but it can also store kd trees for example. This container is compact in the sense that the wasted memory is on the order of $O(\sqrt{n})$, where n is the number of stored elements, at least for cases where essentially element additions are performed (algorithms doing decimations of elements need further consideration). Note that the STL `list` wastes $O(n)$ memory, as well as `vector` which in addition is not as flexible in the operations.

In order to allow easy parallelization of several geometric algorithms using it, we made this container thread-safe, by allowing concurrent addition and removal of elements, with minimal constant-time overhead.

[1] Empty loops parallelize very well. Even turning on compiler optimization completely changes the deal.

[2] `partition` partitions a sequence w.r.t. a given pivot. `nth_element` permutes a sequence such that the element with a given rank k is placed at index k and the smaller ones to the left. `random_shuffle` permutes a sequence randomly.

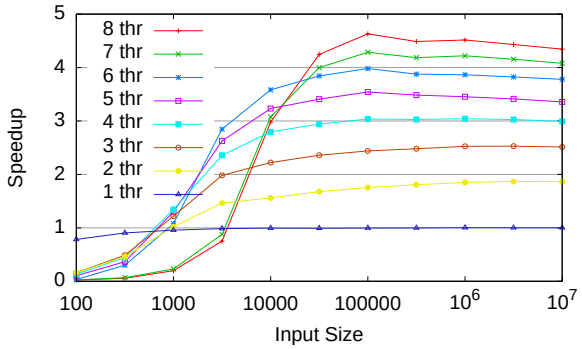


Figure 1: Speedup for 2D spatial sort.

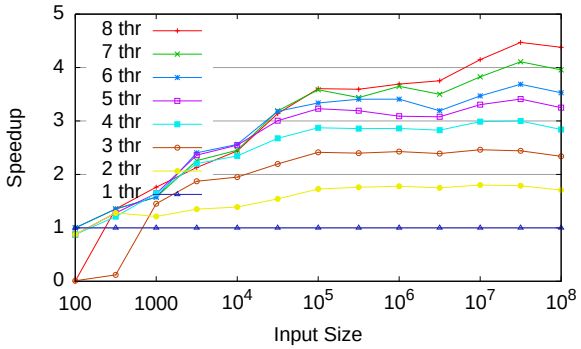


Figure 2: Speedup for kd tree construction.

4 Spatial sorting

Many geometric algorithms are incremental, and their speed depends on the order of insertion for locality reasons in geometric space and in memory. For cases where some randomization is still required for complexity reasons, the Biased Randomized Insertion Order method [1] (BRIO) is an optimal compromise between randomization and locality. Given n randomly shuffled points and a parameter α, BRIO recurses on the first $\lfloor \alpha n \rfloor$ points, and spatially sorts the remaining points. For these reasons, CGAL provides algorithms to sort points along a Hilbert space-filling curve as well as a BRIO.

The sequential implementation uses a divide-and-conquer (D&C) algorithm. It recursively partitions the set of points with respect to a dimension, taking the median point as pivot. The dimension is then changed and the order is reversed appropriately for each recursive call, such that the process results in arranging the points along a Hilbert curve. Parallelizing this algorithm is straightforward. The partitioning is done by calling the parallel `nth_element` function, and the parallel `random_shuffle` for BRIO. The recursive subproblems are processed by newly spawned OpenMP tasks.

Experimental results The speedup (ratio of the running times between the parallel and sequential versions) obtained for 2D Hilbert sorting are shown in Figure 1. For a small number of threads, the speedup is good for problem sizes greater than 1000 points, but the efficiency drops to about 60% for 8 threads. We blame the memory bandwidth limit for this decline. The results for the 3D case are very similar except that the speedup is 10–20% less for large inputs.

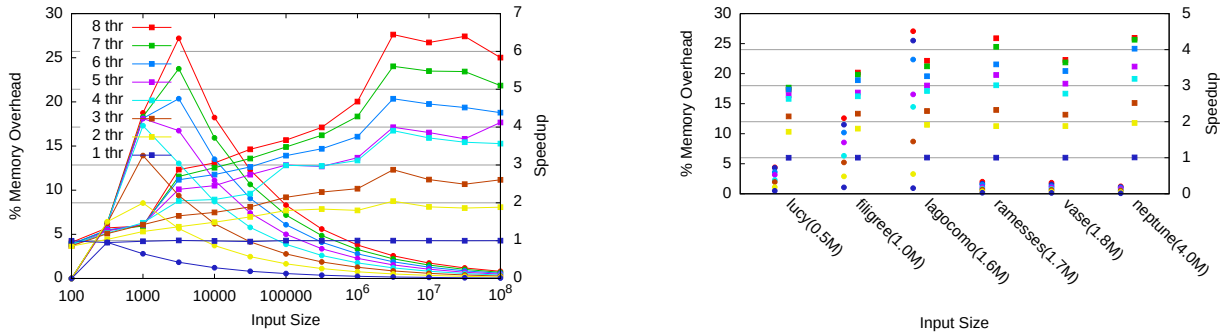
5 Kd tree construction

A kd tree [3] is a fundamental spatial search data structure, allowing efficient queries for the subset of points contained in an orthogonal query box. In principle, a kd tree is a dynamic data structure. However, it is unclear how to do balancing dynamically, so worst-case running time bounds for the queries are only given for trees constructed offline. Also, insertion of a single point is hardly parallelizable. Thus, we chose the construction of the kd tree for a given set of points. The approach is actually quite similar to spatial sorting. The algorithm partitions the data and recursively constructs the subtrees for each half in parallel.

Experimental results The speedup for the parallel kd tree construction of 3D random points is shown in Figure 2. The achieved speedup is similar to the spatial sort case, a little less for small inputs.

6 D-dimensional box intersection

We consider the problem of finding all intersections among a set of n iso-oriented d-dimensional boxes. This problem simplifies the intersection between complex shapes by approximating them by their bounding boxes.



(a) Randomly generated integer coordinates

(b) Real world data sets

Figure 3: Intersecting boxes. Speedup is denoted by squares, relative memory overhead by circles.

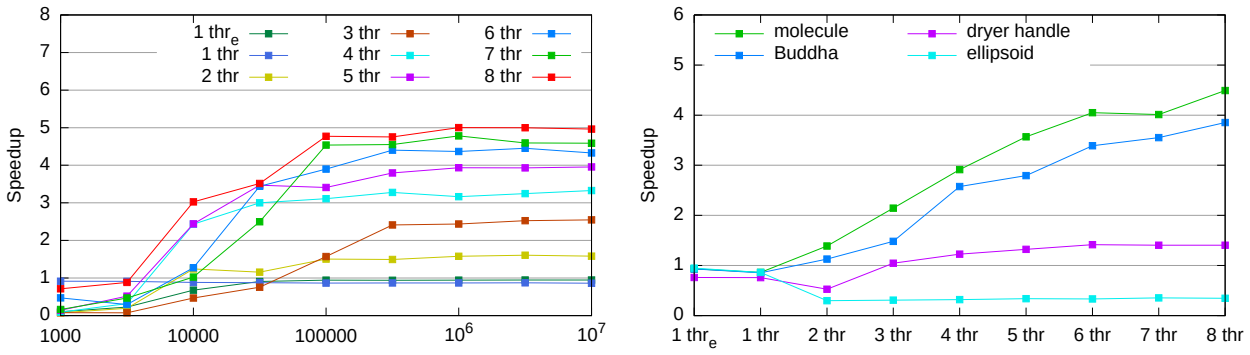
We parallelize the algorithm described in [10], which is used as the sequential implementation in CGAL, and practically efficient. The algorithm is described in terms of nested segment and range trees, leading to an $O(n \log^d n)$ space algorithm in the worst case. Since this is too much space overhead, the trees are not actually constructed, but traversed on the fly. So we end up with a D&C algorithm using only logarithmic extra memory (apart from the possibly quadratic output). Again, the D&C paradigm promises good parallelization opportunities. We can assign the different parts of the division to different threads, since their computation is usually independent. However, we have a detail problem in the two most important recursive conquer calls: the data they process is in general not disjoint. Worse, although they do not *change* the input elements, the recursive calls may reorder them. This is a problem for parallelization, since we cannot just pass the same data to both calls if they are supposed to run in parallel. Thus, we have to copy. Details of the rest of the algorithm are described in [2].

Experimental results 3D boxes with integer coordinates were randomly generated as in [10] such that the expected number of intersections for n boxes is $n/2$. Results are shown in Figure 3(a). The memory overhead is limited to 100%, but as we can see, the relative memory overhead is much lower in practice, below 20% for not-too-small inputs. The speedups are quite good, reaching more than 6 for 8 cores, and being just below 4 for 4 cores. Note that, for reference, the sequential code performs the intersection of 10^6 boxes in 1.86s. Figure 3(b) shows the results for real-world data. We test 3D models for self-intersection, by approximating each triangle with its bounding box. The memory overhead stays reasonable. The speedups are a bit worse than for the random input of the equivalent size. This could be due to the much higher number of found intersections ($\sim 7n$).

7 Bulk insertion into Delaunay triangulations

Given a set S of n points in $\mathbb{R}^d$, a triangulation of S partitions the convex hull of its points into simplices (cells) with vertices in S. The Delaunay triangulation $\mathcal{DT}(S)$ is characterized by the empty sphere property that states the circumsphere of any cell does not contain any other point of S in its interior. A point q is said to be *in conflict* with a cell in $\mathcal{DT}(S)$, if it belongs to the interior of the circumsphere of that cell, and the *conflict region* of q is defined as the set of all such cells.

Sequential framework CGAL provides 2D and 3D incremental algorithms [4]. After a spatial sort using a BRIO, points are iteratively inserted using a *locate step* followed by an *update step*. The locate step finds the cell containing q by navigating using orientation tests and the adjacency relations between cells and starting at some cell incident to the vertex created by the previous insertion. The update step determines the conflict region of q using the Bowyer-Watson algorithm [5, 9], that is, by checking the empty sphere property for all the neighbors of the cell containing q, recursing using the adjacency relations again. The



(a) Random points.

(b) Other data sets.

Figure 4: Speedups for 3D Delaunay triangulation.

conflict region is then removed, creating a "hole", and the triangulation is updated by creating new cells connecting q to the vertices on the boundary of the "hole". From a storage point of view, a vertex stores its point and a pointer to an incident cell, and a cell stores pointers to its vertices and neighbors. Vertices and cells are themselves stored in two *compact containers* (see Section 3).

Parallel algorithm We parallelize by allowing concurrent insertions into the same triangulation, and spreading the input points over all threads. First, a *bootstrap* phase inserts a small random subset of the points using the sequential algorithm, in order to avoid contention for small data sets. Next, the remaining points are Hilbert-sorted in parallel, and the resulting range is divided into almost equal parts attributed to all threads. Threads then insert their points using an algorithm similar to the sequential case, with the addition that threads protect against concurrent modifications to the same region of the triangulation. This protection is performed using fine-grained locks stored in the vertices.

A *lock conflict* occurs when a thread attempts to acquire a lock already owned by another thread. To avoid deadlocks, lock conflicts are handled by *priority locks* where each thread is given a unique priority. If the acquiring thread has a higher priority it simply waits for the lock to be released. Otherwise, it *retreats*, releasing all its locks and restarting an insertion operation, possibly with a different point (the choice of which is described in [2]). This approach avoids deadlocks and guarantees progress.

Locking strategies There are several ways of choosing the vertices to lock. A *simple strategy* consists in locking the vertices of all cells a thread is currently considering. This strategy is simple and easily proved correct. However, as the experimental results show, high degree vertices become a bottleneck with this strategy. We therefore propose *improved strategies* which are described in [2].

Experimental results Our 3D implementation currently uses the simple locking strategy. We carried out experiments on 5 different point sets: 2 synthetic and 3 real-world data. The former consist of evenly distributed points in a cube, and 10^6 points on the surface of an ellipsoid. The real instances are composed of points on the surfaces of a molecule, a Buddha statue and a dryer handle. Figures 4(a) and 4(b) show the results. We observe that a speedup of almost 5 is reached with 8 cores for 10^5 random points. However, we note that surfacic data sets are not so positively affected, and we blame the simple locking strategy for that.

8 Conclusion

We have described several parallel algorithms for 4 core geometric problems, efficient on multi-core architectures. In the future, we plan to tackle more algorithms and propose their integration in CGAL.

References

[1] N. Amenta, S. Choi, and G. Rote. Incremental constructions con brio. In *SCG '03: Proceedings of the nineteenth annual symposium on Computational geometry*, pages 211–219, 2003.

[2] V. H. F. Batista, D. L. Millman, S. Pion, and J. Singler. Parallel geometric algorithms for multi-core computers. Research Report 6749, INRIA, 2008. `http://hal.inria.fr/inria-00343804/`.

[3] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[4] J.-D. Boissonnat, O. Devillers, S. Pion, M. Teillaud, and M. Yvinec. Triangulations in CGAL. *Comput. Geom. Theory Appl.*, 22:5–19, 2002.

[5] A. Bowyer. Computing Dirichlet tesselations. *The Computer Journal*, 24(2):162–166, 1981.

[6] Y. Liu and J. Snoeyink. A comparison of five implementations of 3D delaunay tessellation. In J. E. Goodman, J. Pach, and E. Welzl, editors, *Combinatorial and Computational Geometry*, volume 52 of *MSRI Publications*, pages 439–458. Cambridge University Press, 2005.

[7] J. Singler and B. Kosnik. The libstdc++ parallel mode: Software engineering considerations. In *International Workshop on Multicore Software Engineering (IWMSE)*, 2008.

[8] J. Singler, P. Sanders, and F. Putze. MCSTL: The multi-core standard template library. In *Euro-Par '07: Proceedings of the thirteenth European Conference on Parallel and Distributed Computing*, pages 682–694, 2007.

[9] D. F. Watson. Computing the n-dimensional Delaunay tesselation with application to Voronoi polytopes. *The Computer Journal*, 24(2):167–172, 1981.

[10] A. Zomorodian and H. Edelsbrunner. Fast software for box intersections. *Int. J. Comput. Geometry Appl.*, 12(1-2):143–172, 2002.