Parallel Geometric Algorithms for Multi-Core Computers

Vicente H. F. Batista, David L. Millman, Sylvain Pion, Johannes Singler

Parallel Geometric Algorithms for Multi-Core Computers

Vicente H. F. Batista
COPPE/UFRJ
Rio de Janeiro, Brazil
helano@coc.ufrj.br

David L. Millman
University of North Carolina
Chapel Hill, USA
dave@cs.unc.edu

Sylvain Pion
INRIA
Sophia-Antipolis, France
Sylvain.Pion@sophia.inria.fr

Johannes Singler
Universität Karlsruhe
Karlsruhe, Germany
singler@ira.uka.de

ABSTRACT

Computers with multiple processor cores using shared memory are now ubiquitous. In this paper, we present several parallel geometric algorithms that specifically target this environment, with the goal of exploiting the additional computing power. The d-dimensional algorithms we describe are (a) spatial sorting of points, as is typically used for preprocessing before using 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. We show experimental results for these algorithms in 3D, using our implementations based on the Computational Geometry Algorithms Library (CGAL[1]). This work is a step towards what we hope will become a *parallel mode* for CGAL, where algorithms automatically use the available parallel resources without requiring significant user intervention.

Categories and Subject Descriptors

I.3.5 [**Computational Geometry and Object Modeling**]: Geometric algorithms, languages, and systems; D.1.3 [**Concurrent Programming**]: Parallel programming

General Terms

Algorithms, Experimentation, Performance

Keywords

parallel algorithms, geometric algorithms, Delaunay triangulations, d-dimension, kd-trees, box intersection, spatial sort, compact container, CGAL, multi-core

[1] `http://www.cgal.org/`

SCG'09, June 8–10, 2009, Aarhus, Denmark.

1. INTRODUCTION

It is generally acknowledged that the microprocessor industry has reached the limits of the sequential performance of processors. Processor manufacturers now focus on parallelism to keep up with the demand for high performance. Current laptop computers all have 2 or 4 cores, and desktop computers can easily have 4 or 8 cores, with many more in high-end computers. This trend incites application writers to develop parallel versions of their critical algorithms. This is not an easy task, from both the theoretical and practical points of view.

Work on theoretical parallel algorithms began decades ago, even parallel geometric algorithms have received attention in the literature. In the earliest work, Chow [11] addressed problems such as intersections of rectangles, convex hulls and Voronoi diagrams. Since then, researchers have studied theoretical parallel solutions in the PRAM model, many of which are impractical or inefficient in practice. This model assumes an unlimited number of processors, whereas in this paper we assume that the amount of available processors is significantly less than the input size. Both Aggarwal *et al.* [1] and Akl and Lyons [2] are excellent sources of theoretical parallel *modus operandi* for many fundamental computational geometry problems. The relevance of these algorithms in practice depends not only on their implementability, but also on the architecture details, and implementation experience, which is necessary for obtaining good insight.

Programming tools and languages are evolving to better serve parallel computing. Between hardware and applications, there are several layers of software. The bottom layer contains primitives for thread management and synchronization. Next, the programming layer manipulates the bottom layer's primitives, for example the OpenMP standard applied to the C++ programming language. On top are domain specific libraries, for example the Computational Geometry Algorithms Library (CGAL), which is a large collection of geometric data structures and algorithms. Finally, applications use these libraries. At each level, efforts are needed to provide efficient programming interfaces to parallel algorithms that are as easy-to-use as possible.

In this paper, we focus on shared-memory parallel computers, specifically multi-core CPUs that allow simultaneous execution of multiple instructions on different cores. This explicitly excludes distributed memory systems as well as graphical processing units that have local memory for each

processor and require special code to communicate. As we are interested in practical parallel algorithms it is important to base our work on efficient sequential code. Otherwise, there is a risk of good relative speedups that lack practical interest and skew conclusions about the algorithms scalability. For this reason, we decided to base our work upon CGAL, which already provides mature codes that are among the most efficient for several geometric algorithms [21]. We investigate 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.

Parallelism could also be introduced in geometric predicates — the low-level numerical routines heavily used by geometric algorithms. In high dimension or with multiple precision arithmetic, multi-core parallelism may help. This paper focuses on the more common case of low dimension with floating-point arithmetic, as used in filtering techniques for exact predicates. In this case, the predicates are too fine-grained for multi-core parallelism, but appropriate support from SIMD vector units might help.

The remainder of the paper is organized as follows: Section 2 describes our hardware and software platform; Sections 3, 4, 5 and 6 describe our parallel algorithms, related work and experimental results for (a), (b), (c) and (d) respectively; finally we conclude and present possible future work in parallel computational geometry. Appendix A contains the description of the thread-safe compact container used by the Delaunay triangulation.

2. PLATFORM

OpenMP For thread control, several frameworks of relatively high level exist, such as TBB [23] or OpenMP. We decided to rely on the latter, which is implemented by almost all modern 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. When a new task is spawned, the runtime system can decide to run it with the current thread at once, or postpone it for processing by an arbitrary thread. The GNU C/C++ compiler (GCC) supports this construct as of upcoming version 4.4.

Libstdc++ parallel mode The C++ STL implementation distributed with the GCC features a so-called *parallel mode* [25] as of version 4.3, based on the Multi-Core Standard Template Library [26]. 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 an up-to-date machine, featuring two AMD Opteron 2350 quad-core processors 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`). If not stated otherwise, each test was run at least 10 times, and the average over all running times was taken.

CGAL Kernels Algorithms in CGAL are parameterized by so-called kernels that provide the type of points and accompanying geometric predicates. In each case, we have chosen a kernel that provides appropriate robustness guarantees and is the most efficient:
`Exact_predicates_inexact_constructions_kernel` for Delaunay, and `Simple_cartesian<double>` for the others, since they perform only coordinate comparisons.

3. SPATIAL SORTING

Problem definition and algorithm Many geometric algorithms implemented in CGAL 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 [3] (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 [15].

Since spatial sorting (either strict Hilbert or BRIO) is an important substep of several CGAL algorithms, the parallel scalability of those algorithms would be limited if the spatial sorting was computed sequentially, due to Amdahl's law. For the same reason, the random shuffling is also worth parallelizing.

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. Our interpretation is that the memory bandwidth limit is responsible for this decline. The results for the 3D case are very similar except that the speedup is 10–20% less for large inputs. Note that, for reference, the sequential code sorts 10^6 random points in 0.39s.

4. KD-TREE CONSTRUCTION

Problem definition and algorithm A kd-tree [5] is a fundamental spatial search data structure, allowing efficient queries for the subset of points contained in an orthogonal query box.

[2] `partition` partitions a sequence with respect to a given pivot as in quicksort. `nth_element` permutes a sequence such that the element with a given rank k is placed at index k, the smaller ones to the left, and the larger ones to the right. `random_shuffle` permutes a sequence randomly.

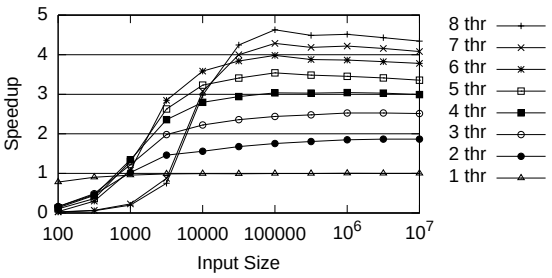


Figure 1: Speedup for 2D spatial sort.

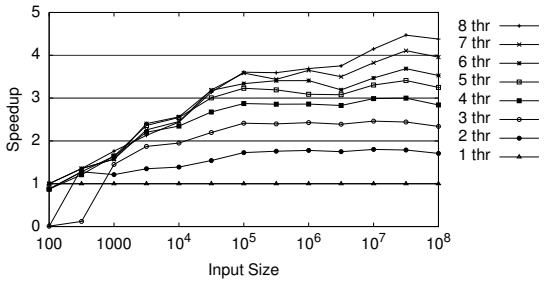


Figure 2: Speedup for kd-tree construction.

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 describe here the construction of the kd-tree for an initially 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. Our implementation is based on the sequential version in CGAL [27].

Experimental results The speedup for the parallel kd-tree construction of 3-dimensional random points with double-precision Cartesian coordinates is shown in Figure 2. The achieved speedup is similar to the spatial sort case, a little less for small inputs. It is worth mentioning that a tree of 10^6 random points is sequentially constructed in 2.74s.

5. D-DIM BOX INTERSECTION

Problem definition We consider the problem of finding all intersections among a set of n iso-oriented d-dimensional boxes. This problem has applications in fields where complex geometric objects are approximated by their bounding box in order to filter them against some predicate.

Algorithm We parallelize the algorithm proposed by Zomorodian and Edelsbrunner [29], which is already used for the sequential implementation in CGAL [18], and proven to perform well in practice. 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). For small subproblems below a certain cutoff size, a base-case quadratic-time algorithm is used to check for intersections.

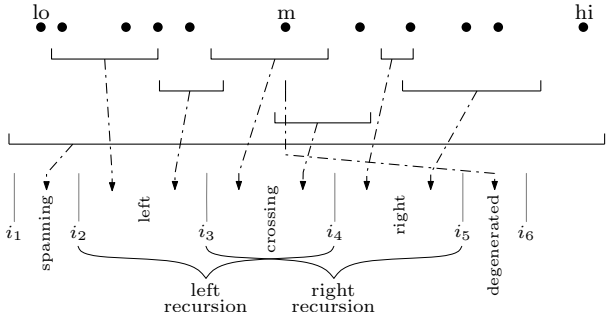


Figure 3: Partitioning the sequence of intervals.

The problem is solved dimension by dimension, recursively. For each dimension, the input consists of two sequences: points and intervals (corners of the boxes and the boxes themselves projected to this dimension, respectively). Here, following the recursive D&C approach, a pivot point m is determined in a randomized fashion, and the sequence of points is partitioned accordingly. The sequence of intervals is also partitioned, but in a more complex way. The sequence L contains all the intervals that have their left end point to the left of m, the sequence R contains all the intervals that have their right end point (strictly[3]) to the right of m. As an exception, degenerated intervals and intervals spanning the full range are treated specially. L and R are passed to the two recursive calls, accompanying the respective points. They can overlap, common elements are exactly the ones crossing m. All these cases are illustrated in Figure 3.

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 for the two recursive conquer calls in the parallel case: as stated before, L and R are not disjoint in general. Although the recursive calls do not *change* the intervals, they may reorder them, so concurrent access is forbidden, even if read-only. Thus, we have to *copy*[4] intervals, which is now explained in detail.

We can reorder the original sequence such that the intervals to the left are at the beginning, the intervals to the right at the end, and the common intervals being placed in the middle. Intervals not contained in any part (degenerated to an empty interval in this dimension) can be moved behind the end. Now, we have five consecutive ranges in the complete sequence. $[i_1, i_2)$ are the intervals spanning the whole region. They are handled separately. $[i_2, i_3)$ and $[i_4, i_5)$ are respectively the intervals for the left and right recursion steps only. $[i_3, i_4)$ are the intervals for both the left and the right recursion steps. $[i_5, i_6)$ are the ignored degenerate intervals.

To summarize, we need $[i_2, i_4) = L$ for the left recursion step, and $[i_3, i_5) = R$ for the right one, which overlap. The easiest way to solve the problem is to either copy $[i_2, i_4)$ or

[3]Whether the comparisons are strict or not, depends on whether the boxes are open or closed. This does not change anything in principle. Here, we describe only the open case.
[4]We could take pointers instead of full objects in all cases since they are only reordered. But this saves only a constant factor and leads to cache inefficiency due to lacking locality, see the Section on Runtime Performance in [18].

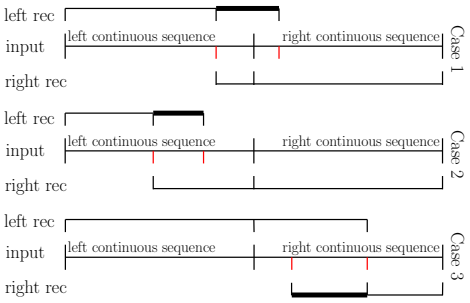


Figure 4: Treating the split sequence of intervals. Fat lines denote copied elements.

$[i_3, i_5)$. But this is inefficient, since for well-shaped data sets (having a relatively small number of intersections), the part $[i_3, i_4)$, which is the only one we really need to duplicate, will be quite small. Thus, we will in fact copy only $[i_3, i_4)$ to a newly allocated sequence $[i'_3, i'_4)$. Now we can pass $[i_2, i_4)$ to the left recursion, and the concatenation of $[i'_3, i'_4)$ and $[i_4, i_5)$ to the right recursion. However, the concatenation must be made implicitly only, to avoid further copying. The danger arises that the number of these gaps might increase in a sequence range as recursion goes on, leading to overhead in time and space for traversing them, which counteracts the parallel speedup.

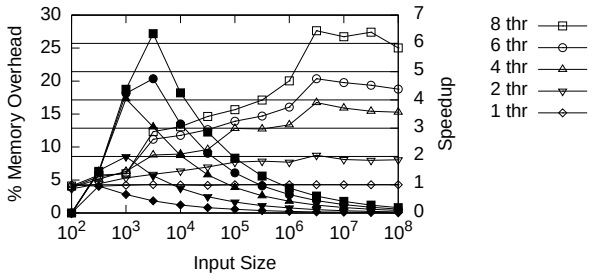
However, we will now prove that this can always be avoided. Let a *continuous sequence* be the original input or a copy of an arbitrary range. Let a continuous range be a range of a continuous sequence. Then, a sequence range consisting of *at most two* continuous ranges always suffices for passing a partition to a recursive call.

Proof: We can ignore the ranges $[i_1, i_2)$ and $[i_5, i_6)$, since they do not take part in this overlapping recursion, so it is all about $[i_2, i_3)$, $[i_3, i_4)$, and $[i_4, i_5)$. *Induction begin:* The original input consists of one continuous range. *Induction hypothesis:* $[i_1, i_6)$ consists of at most two continuous ranges. *Inductive step:* $[i_1, i_6)$ is split into parts. If i_3 is in its left range, we pass the concatenation of $[i_2, i_3)[i'_3, i'_4)$ (two continuous ranges) to the left recursion step, and $[i_3, i_5)$ to the right one. Since the latter is just a subpart of $[i_1, i_6)$, there cannot be additional ranges involved. If i_3 is in the right range of $[i_1, i_6)$, we pass the concatenation of $[i'_3, i'_4)[i_4, i_5)$ (two continuous ranges) to the right recursion step, and $[i_2, i_4)$ to the left one. Since the latter is just a subpart of $[i_1, i_6)$, there cannot be additional ranges involved. The three cases and their treatment are shown in Figure 4.

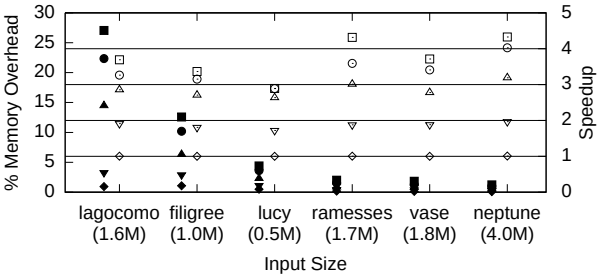
Deciding whether to subtask The general question is how many tasks to create, and when to create them. Having many tasks exploits parallelism better, and improves load balancing. On the other hand, the number of tasks T should be kept low in order to limit the memory overhead. In the worst case, all data must be copied for the recursive call, so the size of additional memory can grow with $O(T \cdot n)$. Generally speaking, only *concurrent* tasks introduce disadvantages, since the additional memory is deallocated after having been used. So if we can limit the number of concurrent tasks to something lower than T, that number will count. There are several criteria that should be taken into account when deciding whether to spawn a task.

- Spawn a new task if the problem to process is large enough (both the number of intervals and the number of points are beyond a certain threshold value c_{min} (tuning parameter)). This strategy strives to amortize for the task creation and scheduling overhead. However, in this setting, the running time overhead can be proportional to the problem size, because of the copying. In the worst case, a constant share of the data must be copied a logarithmic number of times, leading to excessive memory usage.
- Spawn a new task if there are less than a certain number of tasks t_{max} (tuning parameter) in the task queue. Since OpenMP does not allow to inspect its internal task queue, we have to count the number of currently active tasks manually, using atomic operations on a counter. This strategy can effectively limit the number of concurrently processed tasks, and so the memory consumption indirectly.
- Spawn a new task if there is memory left from a pool of size s (tuning parameter). This strategy can effectively limit the amount of additional memory, guaranteeing correct termination.

In fact, we combine the three criteria to form a hybrid. All three conditions must be fulfilled.



(a) Randomly generated integer coordinates



(b) Real world data sets

Figure 5: Intersecting boxes. Speedup is denoted by empty marks, relative memory overhead by filled ones.

Experimental results Three-dimensional boxes with integer coordinates were randomly generated as in [29] such that the expected number of intersections for n boxes is $n/2$.

For the results in Figure 5a, we used $c_{min} = 100$, $t_{max} = 2 \cdot t$, and $s = 1 \cdot n$, where t is the number of threads. 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[5]. The speedups are quite good, reaching more than 6 for 8 cores, and being just below 4

[5]The memory overhead numbers refer to the *algorithmic*

for 4 threads. Note that, for reference, the sequential code performs the intersection of 10^6 boxes in 1.86s.

Figure 5b shows the results for real-world data. We test 3-dimensional models for self-intersection, by approximating each triangle with its bounding box, which is a common application. 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 ($\sim 7n$).

6. BULK DELAUNAY INSERTION

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. The conflict region is non-empty, since it must contain at least the cell the point lies in, and is known to be connected.

Related work Perhaps the most direct method for a parallel scheme is to use the D&C paradigm, recursively partitioning the point set into two subregions, computing solutions for each subproblem, and finally merging the partial solutions to obtain the triangulation. Either the divide or the merge step are usually quite complex, though. Moreover, bulk insertions of points in already computed triangulations is not well supported, as required for many mesh refinement algorithms.

A feasible parallel 3D implementation was first presented by Cignoni *et al.* [13]. In a complex divide step, the *Delaunay wall* is constructed, the set of cells splitting regions, before working in parallel in isolation. As pointed out by the authors, this method suffers from limited scalability due to the cost of wall construction. It achieves only a 3 times speedup triangulating 8000 points on an nCUBE 2 with 8 processors. Cignoni *et al.* [12] also designed an algorithm where each processor triangulates its set of points in an incremental fashion. Although this method does not require a wall, tetrahedra with vertices belonging to different processors are constructed multiple times. A speedup of 5.34 was measured for 8 processors for 20000 random points.

Lee *et al.* [20], focusing on distributed memory systems, improved this algorithm by exploiting a projection-based partitioning scheme [7], eliminating the merging phase. They showed that a simpler non-recursive version of this procedure led to better results for almost all considered inputs. The algorithm was implemented on an INMOS TRAM network of 32 T800 processors and achieved a 6.5 times speedup on 8 processors with 10000 randomly distributed points. However, even their best partitioning method took 75% of the total elapsed time.

The method of Blelloch *et al.* [7] treats the 2D case using the well-known relation with three-dimensional convex hulls. Instead of directly solving a convex hull problem, another reduction step to 2D lower hull is carried out. It was shown that the resulting hull edges are already Delaunay edges, but the algorithm requires an additional step to construct missing edges. They obtained a speedup of 5.7 on 8 CPUs with uniform point distribution on a shared-memory SGI Power Challenge.

More recently, parallel algorithms avoiding the complexity of the D&C algorithms were published. Kohout *et al.* [19] proposed parallelizing randomized incremental construction. This is based on the observation that topological changes caused by point insertion are likely to be extremely local. When a thread modifies the triangulation, it acquires exclusive access to the containing tetrahedron and a few cells around it. For a three-dimensional uniform distribution of half a million points, their algorithm reaches speedups of 1.3 and 3.6 using 2 and 4 threads, respectively on a 4 processors Intel Itanium at 800MHz with a 4MB cache and 4GB RAM. We observed, however, that their sequential speed is about one order of magnitude lower than the CGAL implementation, which would make any parallel speedup comparison unfair.

Another algorithm based on randomized incremental construction was proposed by Blandford *et al.* [6]. It employs a compact data structure and follows a Bowyer-Watson approach [10, 28], maintaining an association between uninserted points and their containing tetrahedra [14]. A coarse triangulation is sequentially built using a separate triangulator (Shewchuk's Pyramid [24]) before threads draw their work from the subsets of points associated with these initial tetrahedra. This is done in order to build an initial triangulation sufficiently large so as to avoid thread contention. For uniformly distributed points, their algorithm achieved a relative speedup of 46.28 on 64 1.15-GHz EV67 processors with 4GB RAM per processor, spending 6% to 8% of the total running time in Pyramid, but with the smallest instance consisting of 23 million points. Their work targeted huge triangulations ($2^{30.5}$ points on 64 processors), as they also use compression schemes which would only slow things down for more common input sizes. In this paper, we are also interested in speeding up smaller triangulations, whose size ranges from a thousand to millions of points (in fact, we tested up to 31M points, which fits in 16GB of memory).

Sequential framework CGAL provides 2D and 3D incremental algorithms [9] and a similar approach has also been implemented in d dimensions [8]. 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 using a *remembering stochastic walk* [16] that starts at some cell incident to the vertex created by the previous insertion, and navigates using orientation tests and the adjacency relations between cells. The update step determines the conflict region of q using the Bowyer-Watson algorithm [10, 28], that is, by checking the empty sphere property for all the neighbors of the cell containing q, recursing using the adjacency relations again. The 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 Appendix A). Note that there is also an infinite vertex linked to the convex hull through infinite cells. Also worth noting for the sequel is that once a vertex is created, it never moves (this paper does not consider removing ver-

overhead only. For software engineering reasons, e. g., preserving the input sequence, the algorithm may decide to copy the whole input in a preprocessing step.

tices), therefore its address is stable, while a cell can be destroyed by subsequent insertions.

Parallel algorithm We attack the problem of constructing $\mathcal{DT}(S)$ in parallel by allowing concurrent insertions into the same triangulation, and spreading the input points over all threads. Our scheme is similar to [6, 19], but with different point location, load management mechanisms and locking strategies.

First, a *bootstrap* phase inserts a small randomly chosen subset S_0 of the points using the sequential algorithm, in order to avoid contention for small data sets. The size of S_0 is a tuning parameter. 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 (location and updating steps), but 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. Concerning the storage, a thread-safe variant of the container used in the sequential setting has been devised (see Appendix A).

Locking and retreating Threads *read* the data structure during the locate step, but only the update step *locally modifies* the triangulation. To guarantee threads safety, both procedures lock and unlock some vertices.

A *lock conflict* occurs when a thread attempts to acquire a lock already owned by another thread. Systematically waiting for the lock to be released is not an option since a thread may already own other locks, potentially leading to a deadlock. Therefore, lock conflicts are handled by *priority locks* where each thread is given a unique priority (totally ordered). 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. This approach avoids deadlocks and guarantees progress. The implementation of priority locks needs attention, since comparing the priority and acquiring a lock need to be performed atomically[6].

Interleaving A retreating thread should continue by inserting a far away point, hopefully leaving the area where the higher priority thread is operating. On the other hand, inserting a completely unrelated point is impeded by the lack of a expectedly close starting point for the locate step. Therefore, each thread divides its own range into several parts of roughly equal sizes, and keeps a reference vertex for each of them to restart point location. The number of these parts is a tuning parameter of the algorithm. It starts to insert points from the first part. Each time it has to retreat, it switches to the next part in a round-robin fashion. Because the parts are constructed from disjoint ranges of the Hilbert-sorted sequence, vertices taken from different parts are not particularly likely to be spatially close and trigger conflicts. This results in an effective compromise between locality of reference and conflict avoidance.

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. During the locate step, this means locking the $d+1$ vertices of the current cell, then, when moving to a neighboring cell, locking the opposite vertex and releasing the unneeded lock. During the update step, all vertices of all cells in conflict are locked, as well as the vertices of the cells that share a face with those in conflict, since those cells are also tested for the *insphere* predicate, and at least one of their neighbor pointers will be updated. Once the new cells are created and linked, the acquired locks can be released. This strategy is simple and easily proved correct. However, as the experimental results show, high degree vertices become its bottleneck.

We therefore propose an *improved strategy* that reduces the number of locks and particularly avoids locking high degree vertices as much as possible. It works as follows: reading a cell requires locking at least two of its vertices, changing a cell requires locking at least d of its vertices, and changing the incident cell pointer of a vertex requires it to be locked. This rule implies that a thread can change a cell without others reading it, but it allows some concurrency among reading operations. Most importantly, it allows reading and changing cells without locking all their vertices, therefore giving some leeway to avoid locking high degree vertices. During the locate step, keeping at most two vertices locked is enough: when using neighboring relations, choosing a vertex common with the next cell is done by choosing the one closest to q (thereby discarding the infinite vertex). During the update step, a similar procedure needs to be followed except that once a cell is in conflict, it needs to have d vertices locked, which allows to exclude the furthest vertex from q, with the following caveat: all vertices whose incident cell pointer point to this cell also need to be locked. This measure is necessary so that other threads starting a locate step at this vertex can access the incident cell pointer safely. Once the new cells are created and linked, the incident cell pointers of the locked vertices are updated (and those only) and the locks are released.

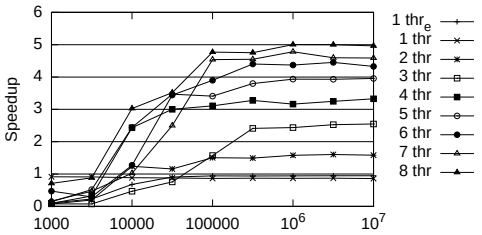
The choice of attempting to exclude the furthest vertex is motivated by the consideration of *needle* shaped simplices for which it is preferable to avoid locking the singled-out vertex as it has a higher chance of being of high degree. For example, the infinite vertex will be locked only when a thread needs to modify the incident cell it points to. Similarly, performing the locate step in a data set associated with an arbitrary surface will likely lock vertices which are on the same sheet as q.

Experimental results We have implemented our parallel algorithm based on the 3D CGAL code [22] using the simple locking strategy only, at the moment. We carried out experiments on five different point sets, including two synthetic and three real-world data. The synthetic data consist of evenly distributed points in a cube, and 1 million points lying on the surface of an ellipsoid of axes lengths 1, 2 and 3. The real instances are composed of points on the surfaces of a molecule, a Buddha statue, and a dryer handle containing 525K, 543K and 50K points respectively. For reference, the original sequential code computes a triangulation of 10^6 random points in 16.57s.

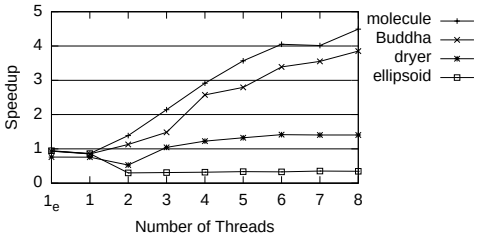
Running time measurement has proven very noisy in practice, especially for small instances. For this reason, we iterated the measurements a number of times depending on the input size.

Figures 6a and 6b show the achieved speedups. We observe that a speedup of almost 5 is reached with 8 cores for 10^5 random points or more. However, we note that the point

[6]Since this is not efficiently implementable using OpenMP primitives, we used our own implementation employing spin locks based on hardware-supported atomic operations.



(a) Random points.



(b) Other data sets.

Figure 6: Speedups of our algorithm on different inputs. Variables subscripted with e denote values for the "empty" lock traits, i. e., without actually locking.

sets corresponding to surfaces are not so positively affected.

The cost of the algorithm can be broken down into several parts. These include the locate and update steps, as well as the bootstrap and spatial sort. This breakdown is shown in Figure 7 for all point sets we have considered. All steps have achieved good scalability in the random case, happy Buddha and molecule. It slows down, however, in the dryer handle and ellipsoid instances. This reveals an important problem of the simple locking strategy, which interacts badly with high degree vertices, like the infinite vertex which is connected to all points on the convex hull.

Tuning parameters In order to empirically select generally good values for the parameters which determine the size of S_0 (we chose $100p$, where p is the number of threads) and the interleaving degree (we chose 2), we have studied their effect on the speedup as well as the number of retreats. Table 1 shows the outcome of these tests for 131K random points. A small value like 2 for the interleaving degree already provides most of the benefit of the technique. The bootstrap size has no significant influence on the running time for large data sets, but it affects the number of retreats which may affect small data sets.

We also experimented with several (close-by) vertices sharing a lock, trying to save time on acquisitions and releases. However, the necessary indirection and the additional lock conflicts counteracted all improvement. Table 1 illustrates the performance degradation introduced by using this mechanism, even with only one vertex per lock. Reference [4] provides detailed experimental results on these parameters for different data sets.

7. CONCLUSION AND FUTURE WORK

We have described new parallel algorithms for four fundamental geometric problems, especially targeted at shared-memory multi-core architectures, which are increasingly accessible. These are d-dimensional spatial sorting of points, kd-tree construction, axis-aligned box intersection computation, and bulk insertion of points in Delaunay triangulations. Experiments in 3D show significant speedup over their already efficient sequential original counterparts, as well as good comparison to previous work for the Delaunay computation for problems of reasonable size.

In the future, we plan to extend our implementation to cover more algorithms, and then submit it for integration in CGAL once it is stable enough, to serve as a first stone towards a *parallel mode* which CGAL users will be able to benefit from transparently.

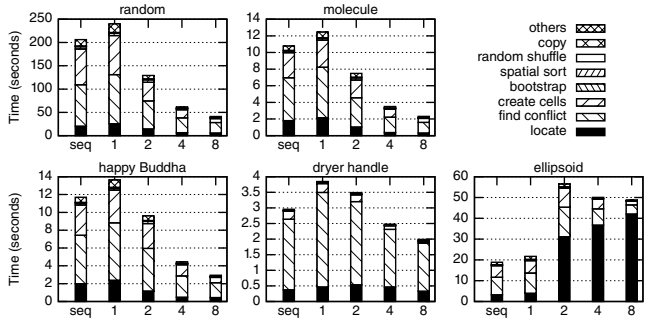


Figure 7: Time break down of our algorithm with bootstrap size $100p$ and interleaving degree 2.

8. ACKNOWLEDGMENTS

This work has been supported by the INRIA Associated Team GENEPI, the NSF-INRIA program REUSSI, the Brazilian CNPq sandwich PhD program, the French ANR program TRIANGLES, and DFG grant SA 933/3-1. We also thank Manuel Holtgrewe for his implementation work on the parallel kd-trees.

9. REFERENCES

[1] Alok Aggarwal, Bernard Chazelle, Leonidas J. Guibas, Colm Ó'Dúnlaing, and Chee-Keng Yap. Parallel computational geometry. *Algorithmica*, 3(1–4):293–327, 1988.

[2] Selim G. Akl and Kelly A. Lyons. *Parallel computational geometry*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[3] Nina Amenta, Sunghee Choi, and Günter Rote. Incremental constructions con brio. In *Proc. 19th ACM Symp. Comp. Geom.*, pages 211–219, 2003.

[4] Vicente H. F. Batista, David L. Millman, Sylvain Pion, and Johannes Singler. Parallel geometric algorithms for multi-core computers. Research Report 6749, INRIA, 2008. `http://hal.inria.fr/inria-00343804`.

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

[6] Daniel K. Blandford, Guy E. Blelloch, and Clemens Kadow. Engineering a compact parallel Delaunay algorithm in 3D. In *Proc. 22nd Symp. Comp. Geom.*, pages 292–300, 2006.

[7] Guy E. Blelloch, Jonathan C. Hardwick, Gary L. Miller, and Dafna Talmor. Design and implementation

Table 1: Speedups and percentage of retreated vertices (between parenthesis) for triangulating 131K random points with different values of bootstrap and interleaving, and for two locking schemes (not shared and shared with one vertex per lock).

	Bootstrap Size				Interleaving Size				Indirection Overhead	
#	$8p$	$32p$	$128p$	$512p$	1	2	4	8	not shared	shared
2	0.88 (2.08)	1.26 (0.62)	1.54 (0.05)	1.60 (0.03)	0.92 (7.33)	1.52 (0.11)	1.39 (0.03)	1.50 (0.08)	1.70 (0.12)	1.43 (0.15)
4	0.71 (34.24)	3.18 (3.24)	3.43 (0.39)	3.51 (0.16)	3.37 (4.79)	3.40 (0.54)	3.40 (0.34)	3.40 (0.96)	3.68 (0.47)	3.59 (0.56)
8	0.36 (10.25)	4.77 (5.93)	4.98 (1.07)	4.43 (0.43)	4.83 (10.00)	4.84 (2.19)	5.00 (0.48)	4.60 (0.70)	5.47 (1.94)	5.28 (2.18)

of a practical parallel Delaunay algorithm. *Algorithmica*, 24(3):243–269, 1999.

[8] Jean-Daniel Boissonnat, Olivier Devillers, and Samuel Hornus. Incremental construction of the Delaunay triangulation and the Delaunay graph in medium dimension. In *Proc. 25th ACM Symp. Comp. Geom.*, 2009.

[9] Jean-Daniel Boissonnat, Olivier Devillers, Sylvain Pion, Monique Teillaud, and Mariette Yvinec. Triangulations in CGAL. *Comp. Geom. Theory & Appl.*, 22:5–19, 2002.

[10] Adrian Bowyer. Computing Dirichlet tesselations. *Comput. J.*, 24(2):162–166, 1981.

[11] Anita Liu Chow. *Parallel algorithms for geometric problems*. PhD thesis, University of Illinois, Urbana-Champaign, IL, USA, 1980.

[12] Paolo Cignoni, Domenico Laforenza, Claudio Montani, Raffaele Perego, and Roberto Scopigno. Evaluation of parallelization strategies for an incremental Delaunay triangulator in E^3. *Conc. Pract. and Exp.*, 7(1):61–80, 1995.

[13] Paolo Cignoni, Claudio Montani, Raffaele Perego, and Roberto Scopigno. Parallel 3D Delaunay triangulation. *Comput. Graph. Forum*, 12(3):129–142, 1993.

[14] Kenneth L. Clarkson and Peter W. Shor. Applications of random sampling in computational geometry, II. *Disc. & Comp. Geom.*, 4(5):387–421, 1989.

[15] Christophe Delage. Spatial sorting. In CGAL Editorial Board, editor, *CGAL Manual*. 3.4 edition, 2008.

[16] Olivier Devillers, Sylvain Pion, and Monique Teillaud. Walking in a triangulation. *Int. J. Found. Comput. Sci.*, 13(2):181–199, 2002.

[17] Michael Hoffmann, Lutz Kettner, Sylvain Pion, and Ron Wein. STL extensions for CGAL. In CGAL Editorial Board, editor, *CGAL Manual*. 3.4 edition, 2008.

[18] Lutz Kettner, Andreas Meyer, and Afra Zomorodian. Intersecting sequences of dD iso-oriented boxes. In CGAL Editorial Board, editor, *CGAL Manual*. 3.4 edition, 2008.

[19] Josef Kohout, Ivana Kolingerová, and Jiří Žára. Parallel Delaunay triangulation in E^2 and E^3 for computers with shared memory. *Parallel Computing*, 31(5):491–522, 2005.

[20] Sangyoon Lee, Chan-Ik Park, and Chan-Mo Park. An improved parallel algorithm for Delaunay triangulation on distributed memory parallel computers. *Parallel Processing Letters*, 11(2/3):341–352, 2001.

[21] Yuanxin Liu and Jack Snoeyink. A comparison of five implementations of 3D Delaunay tessellation. In Jacob E. Goodman, János Pach, and Emo Welzl, editors, *Combinatorial and Computational Geometry*, volume 52 of *MSRI Publications*, pages 439–458. Cambridge University Press, 2005.

[22] Sylvain Pion and Monique Teillaud. 3D triangulations. In CGAL Editorial Board, editor, *CGAL Manual*. 3.4 edition, 2008.

[23] James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, Inc., 2007.

[24] Jonathan R. Shewchuk. Tetrahedral mesh generation by Delaunay refinement. In *Proc. 14th ACM Symp. Comp. Geom.*, pages 86–95, 1998.

[25] Johannes Singler and Benjamin Kosnik. The libstdc++ parallel mode: Software engineering considerations. In *Int. Worksh. Multicore Software Eng.*, 2008.

[26] Johannes Singler, Peter Sanders, and Felix Putze. MCSTL: The multi-core standard template library. In *Proc. 13th Eur. Conf. Parallel and Distributed Comp.*, pages 682–694, 2007.

[27] Hans Tangelder and Andreas Fabri. dD spatial searching. In CGAL Editorial Board, editor, *CGAL Manual*. 3.4 edition, 2008.

[28] David F. Watson. Computing the n-dimensional Delaunay tesselation with application to Voronoi polytopes. *Comput. J.*, 24(2):167–172, 1981.

[29] Afra Zomorodian and Herbert Edelsbrunner. Fast software for box intersections. *Int. J. Comp. Geometry Appl.*, 12(1-2):143–172, 2002.

APPENDIX

A. THREADED, COMPACT CONTAINER

Many geometric data structures are composed of large sets of small objects of the same type or a few different types, organized as a graph. Delaunay triangulations, for example, are often represented as graphs connecting vertices, simplices and eventually k-simplices. It is also the case for kd-trees and other trees which are recursive data structures manipulating many nodes of the same type.

The geometric data structures of CGAL typically provide iterators over the elements such as the vertices, in the same spirit as the STL containers. In a nutshell, a container encapsulates a memory allocator together with a means to iterate over its elements, the iterator.

For efficiency reasons, elements are preferably stored in a way that avoids wasting memory for the internal bookkeeping. Moreover, spatial and temporal locality are important factors: the container should attempt to keep elements that have been added consecutively close to each other in memory, in order to reduce cache thrashing. The operations that must be efficiently supported are the addition of a new element and the release of an obsolete element, and both must not invalidate the iterators to other elements.

A typical example is the 3D Delaunay triangulation, which is using a container for the vertices and a container for the cells. Building a Delaunay triangulation requires efficient alternate addition and removal of new and old cells, and addition of new vertices.

A compact container To this effect, and with the aim of providing a container that can be re-used in several geometric data structures, we have designed a container with the desired properties. A non thread-safe version of our container is already available in CGAL as the `Compact_container` class [17]. Its key features are: (a) amortized constant time addition and removal of elements, (b) very low asymptotic memory overhead and good memory locality.

Note that we use the term *addition* instead of the more familiar *insertion*, since the operation does not allow to specify *where* in the iterator sequence a new element is to be added. This is generally not an issue for geometric data structures that do not have meaningful linear orders.

The `Compact_container` can be compared to mostly two STL containers : `vector` and `list`.

- A `vector` is able to add new elements at its back in amortized constant time, but erasing any element can be performed in only linear time. Its memory overhead is typically a constant factor (usually 2) away from the optimal, since elements are allocated in a consecutive array, which is resized on demand. Moreover, addition of an element may invalidate iterators if the capacity is exceeded and a resizing operation is triggered, which is at best inconvenient. Consider the elements being the nodes of a graph, referring to each other. It could nevertheless be useful under some circumstances, like storing vertices if a bound on their numbers is known in advance. One of its advantages is that its iterator provides random access, i. e. the elements can be easily numbered.
- A `list` allows constant time addition and removal anywhere in the sequence, and its iterator is not invalidated on these operations. The main disadvantage of a `list` is that it stores nodes separately, and the need for an iterator implies that typically two additional pointers are stored per node, plus the allocator's internal overhead for each node. Singly-connected lists would decrease this overhead slightly by removing one pointer, but there would still be one remaining, plus the internal allocator overhead.

Note that the STL `deque` does not improve over `vector` for our purpose, since it still only offers linear time complexity for removing elements in general.

The `Compact_container` improves over these by mixing advantages from both, in the form which can be roughly described as a list of blocks. It allocates elements in consecutive blocks, which reduces the allocator's internal overhead. In order to reduce it at best asymptotically, it allocates blocks of linearly increasing size (in practice starting at 16 elements and increasing 16 by 16 subsequently). This way, n elements are stored in $O(\sqrt{n})$ blocks of maximum size $O(\sqrt{n})$. There is a constant memory overhead per block (assuming the allocator's internal bookkeeping is constant), which causes a sub-linear waste of $O(\sqrt{n})$ memory in the worst case. This choice of block size evolution is optimal, as it minimizes the sum of a single block size (the wasted memory in the last block which is partially filled) and the number of blocks (the wasted memory which is a constant per block).

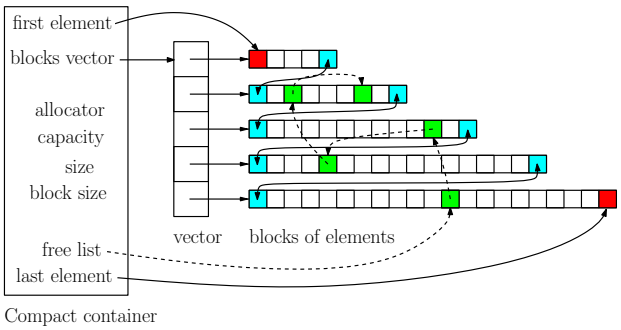


Figure 8: `Compact_container` memory layout.

Each block's first and last elements are not available to the user, but used as markers for the needs of the iterator, so that the blocks are linked and the iterator can iterate over the blocks. Allowing removal of elements anywhere in the sequence requires a way to mark those elements as free, so that the iterator knows which elements it can skip. This is performed using a trick, which requires an element to contain a pointer to 4-byte aligned data. This is the case for many objects such as CGAL's Delaunay vertices and cells, or any kind of node that stores a pointer to some other element. Whenever this is not possible, for example when storing only a point with only floating-point coordinates, an overhead is indeed triggered by this pointer. The 4-byte alignment requirement is not a big issue in practice on current machines as many objects are required to have an address with at least such an alignment, and it has the advantage that all valid pointers have their two least significant bits zeroed. The `Compact_container` uses this property to mark free elements by setting these bits to non-zero, and using the rest of the pointer for managing a singly-connected free list.

Erasing an element then simply means adding it to the head of the free list. Adding an element is done by taking the first element of the free list if it is not empty. Otherwise, a new block is allocated, all its elements are added to the free list, and the first element is then used.

Figure 8 shows the memory layout of the `Compact_container`. In the example, 5 blocks are allocated and 5 elements are on the free list. We see that the container maintains pointers to the first and last elements for the iterator's needs, and for the same reason all blocks are chained, hence the first and last elements of each block are not usable. It also maintains the size (the number of live elements, here 50), the capacity (the maximum achievable size without reallocation, here 55) and the current block size (here 21). In addition, but not strictly necessary, a vector stores the pointers to all blocks, in order to be able to reach the blocks more efficiently when allowing block de-allocation. Indeed, de-allocating a block in the middle of the blocks sequence (which could be useful to release memory) prevents the predictability of the size of each block, and hence the constant time reachability of the end of the blocks, which is otherwise the only way to access the next block. A practical advantage of this is that it allows to destroy a container in $O(\sqrt{n})$ time instead of $O(n)$, when the element's destructor is trivial (completely optimized away) as is often the case.

This design is very efficient as the needs for the iterator cause no overhead for live elements, and addition and removal of elements are just a few simple operations in most cases. Memory locality is also rather good overall: if only additions are performed, then the elements become consecutive in memory, and the iterator order is even the order of the additions. For alternating sequences of additions and removals, like a container of cells of an incremental Delaunay triangulation might see, the locality is still relatively good if the points are inserted in a spatial local order such as Hilbert or BRIO. Indeed, for a point insertion, the Bowyer-Watson algorithm erases connected cells, which are placed consecutively on the free list, and requires new cells, consecutively as well, from the free list, to re-triangulate the hole. So, even if some shuffling is unavoidable, this simple mechanism takes care of maintaining a relatively good locality of the data on a large scale.

Experimental comparison We have measured the time and memory space used by the computation of a (sequential) 3D Delaunay triangulation of 1 million random points using CGAL, only changing the containers used internally to store the vertices and cells. The experiment was conducted on a MacBook Pro with a 2.33 GHz 32 bit Intel processor, 2 GB of RAM, and using the GCC compiler version 4.3.2 with optimization level 2. Using `list`, the program took 19.3 seconds and used 389 MB of RAM, while using our `Compact_container` it took 13.3 seconds and used 288 MB. The optimal memory size would have been 258 MB, as computed by the number of vertices and cells times their respective memory sizes (28 and 36 bytes respectively). This means that the internal memory overhead was 51% for `list` and only 11% for `Compact_container`.

We also performed the same experiment, this time for 10 million points, on a 64 bit machine with 16 GB of RAM under Linux. We observed almost the same internal memory overheads but with a smaller time difference of still 28%. For 30 million points, the internal memory overhead for `Compact_container` went down to 8.6%.

Fragmentation Some applications like mesh simplification build a large triangulation and then select a significant fraction of the vertices and remove them. In such cases, the `Compact_container` is not at its best, since it never releases memory blocks to the system automatically. In fact, such an application would produce a very large free list, with the free elements being spread all over the blocks, producing high fragmentation. Moreover, the iterator would be slower as it skips the free elements one by one (this would even violate the complexity requirements of standard iterators whose increment and decrement operations must be amortized constant time). The `Compact_container` does not provide a specific function to handle this issue, and the recommended way to improve the situation is to copy the container, which moves its elements to a new, compact area.

Parallelization Using the `Compact_container` in the parallel setting required some changes. A design goal is to have a shared data structure (e.g., a triangulation class), and change it using several threads concurrently. So the container is required to support concurrent addition and removal operations. At such a low level, thread safety needs to be achieved in an efficient way, as taking locks for each operation would necessarily degrade performance, with lots of expected contention.

The way we extended the `Compact_container` class is that we chose to have one independent free list per thread, which completely got rid of the need for synchronization in the removal operation. Moreover, considering the addition operation, if the thread's free list is not empty, then a new element can be taken from its head without need for synchronization either, and if the free list is empty, the thread allocates a new block, and adds its elements to its own free list. Therefore, the only synchronization needed is when allocating a new block, since (a) the allocator may not be thread-safe and (b) all blocks need to be known by the container class so a vector of block pointers is collected. Since the size of the blocks is growing as $O(\sqrt{n})$, the relative overhead due to synchronization also decreases as the structure grows.

Note that, since when allocating a new block, all its elements are put on the current thread's free list, it means that they will initially be used only by this thread, which also helps locality in terms of threads. However, once an element has been added, another thread can remove it, putting it on its own free list. So in the end, there is no guarantee that elements in a block are "owned" forever by a single thread, some shuffling can happen. Nevertheless, we should obtain a somewhat "global locality" in terms of time, memory, and thread (and geometry thanks to spatial sorting, if the container is used in a geometric context).

A minor drawback of this approach is that free elements are more numerous, and the wasted memory is expected to be $O(t\sqrt{n})$ for t threads, each typically wasting a part of a block (assuming an essentially incremental algorithm, since here as well, no block is released back to the allocator).

Element addition and removal are operations which are then allowed to be concurrent. Read-only operations like iterating can also be performed concurrently.

Benchmark Figure 9 shows a synthetic benchmark of the parallel `Compact_container` alone, by performing essentially parallel additions together with 20% of interleaved deletions, and comparing it to the sequential original version. We see that the container scales very nicely with the number of threads as soon as a minimum number of elements is reached. This benchmark is synthetic, since no computation is performed between the insertions and deletions. We can hope that using it for geometric algorithms will prove it useful even with lower numbers of elements, although this is hard to measure.

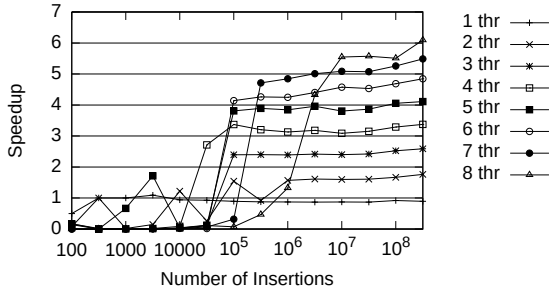


Figure 9: Speedups obtained for the compact container with additions and 20% of deletions.