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[a], David L. Millman[b], Sylvain Pion[*,c], Johannes Singler[d]

[a] *Departamento de Engenharia Civil, Universidade Federal do Rio de Janeiro, Caixa Postal 68506, Rio de Janeiro, RJ, 21945-970, Brazil*

[b] *Department of Computer Science, University of North Carolina, CB 3175 Sitterson Hall, Chapel Hill, NC 27599-3175, USA*

[c] *Institut National de Recherche en Informatique et en Automatique, 2004 route des Lucioles, BP 93, 06902 Sophia Antipolis, France*

[d] *Fakultät für Informatik, Karlsruhe Institute of Technology, Postfach 6980, 76128 Karlsruhe, Germany*

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 algorithms we describe are (a) 2-/3-dimensional spatial sorting of points, as is typically used for preprocessing before using incremental algorithms, (b) d-dimensional axis-aligned box intersection computation, and finally (c) 3D bulk insertion of points into Delaunay triangulations, which can be used for mesh generation algorithms, or simply for constructing 3D Delaunay triangulations. For the latter, we introduce as a foundational element the design of a container data structure that both provides concurrent addition and removal operations and is compact in memory. This makes it especially well-suited for storing large dynamic graphs such as Delaunay triangulations.

We show experimental results for these algorithms, using our implementations based

[☆] 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.

[*] Corresponding author.

Email addresses: helano@coc.ufrj.br (Vicente H. F. Batista), dave@cs.unc.edu (David L. Millman), Sylvain.Pion@sophia.inria.fr (Sylvain Pion), singler@kit.edu (Johannes Singler)

on the Computational Geometry Algorithms Library (CGAL). 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.

Key words: parallel algorithms, geometric algorithms, Delaunay triangulations, box intersection, spatial sort, compact container, CGAL, multi-core

---

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 [1] 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. [2] and Akl and Lyons [3] 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 hardware architecture targeted.

Programming tools and languages are evolving to better support parallel computing. Between hardware and applications, there are several layers of software. The bottom layer, e. g., OpenMP, contains primitives for thread management and synchronization, which builds on OS capabilities and hardware-supported instructions. On top of that, parallel algorithms

can be implemented in domain specific libraries, as we show in this paper for the Computational Geometry Algorithms Library (CGAL) [4], which is a large collection of geometric data structures and algorithms. Finally, applications can use the implicit parallelism encapsulated in such a library, without necessarily doing explicit parallel programming on this level.

In this paper, we focus on shared-memory parallel computers, specifically multi-core CPUs, which allow simultaneous execution of multiple instructions on different cores. This explicitly excludes distributed memory systems as well as graphical processing units, which have local memory for each processor core and thus 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 [5]. We investigate the following algorithms: (a) 2-/3-dimensional spatial sorting of points, as is typically used for preprocessing before using incremental algorithms, (b) d-dimensional axis-aligned box intersection computation, and finally (c) 3D bulk insertion of points in Delaunay triangulations, which can be used for mesh generation algorithms, or simply for constructing 3D Delaunay triangulations.

The remainder of the paper is organized as follows: Section 2 describes our hardware and software platform; Section 3 contains the description of the thread-safe compact container used by the Delaunay triangulation; Sections 4, 5 and 6 describe our parallel algorithms, the related work and the experimental results for (a), (b) and (c) respectively; we conclude and present future plans in Section 7.

2. Platform

OpenMP. For thread control, several frameworks of relatively high level exist, such as TBB [6] or OpenMP [7]. We decided to rely on the latter, which is implemented by almost all modern compilers. As a new feature, the OpenMP specification in version 3.0

includes the `#pragma omp task` construct. This creates a *task*, i. e., a code block that is executed asynchronously. Creating such tasks 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. If the task model is not fully appropriate, the program can also just run a certain number of threads and make them process the problem.

Libstdc++ parallel mode. The C++ STL implementation distributed with the GCC features a so-called *parallel mode* [8] as of version 4.3, based on the Multi-Core Standard Template Library [9]. It provides parallel versions of many STL algorithms. We use some of these algorithmic building blocks, such as `partition`, `nth_element` and `random_shuffle`. The `partition` algorithm partitions a sequence with respect to a given pivot as in quicksort. Applying `nth_element` to a sequence places the element with a given rank k at index k, and moves the smaller ones to the left, the larger ones to the right. The `random_shuffle` routine is used to permute a sequence randomly.

Evaluation system. We evaluated the performance of our algorithms on an up-to-date machine, featuring two AMD Opteron 2350 quad-core 64-bit processors at 2 GHz and 16 GB of RAM. We used GCC 4.4 (for the algorithms using the task construct), enabling optimization (`-O3` 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, which provide the type of points and accompanying geometric predicates. In each case, we have chosen the kernel that is most efficient while providing appropriate robustness guarantees: `Exact_predicates_inexact_constructions_kernel` for Delaunay triangulation, and `Simple_cartesian<double>` for the other algorithms, since they perform only coordinate comparisons.

3. Thread-safe 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 and simplices.

The geometric data structures of CGAL typically provide iterators over elements such as 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.

Elements are preferably stored in a way that avoids wasting memory for the internal bookkeeping. Moreover, spatial and temporal locality are important factors for performance: 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 removal 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 alternating addition and removal of new and old cells, and addition of new vertices.

3.1. 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 [10]. 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` is most closely comparable to the STL container `list`, since `vector` and `deque` are too constrained for our usage. The main disadvantage of a `list` is

the additional storage of two pointers and the allocator's internal overhead for each element.

The `Compact_container` improves the memory usage over `list` by implementing a list of blocks of consecutive elements. Doing so, it amortizes the allocator's internal overhead together with the pointers between blocks. For best asymptotic behavior, the size of the blocks increases linearly (in practice starting at 16 elements and incremented 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 growth 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).

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 to skip. This involves a trick requiring an element to contain a pointer to 4-byte aligned data, which is the case for many objects such as the vertices and cells of the CGAL Delaunay triangulations, or any kind of graph 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 constraint 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 these bits to mark free elements by setting them to non-zero, and using the rest of the pointer for managing a singly-connected free list of elements.

Removing 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 returned.

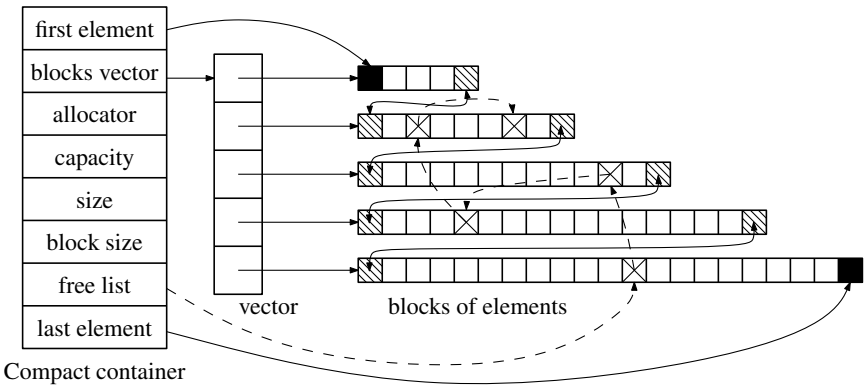


Figure 1: `Compact_container` memory layout.

Figure 1 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 needs of the iterator, and for the same reason all blocks are chained. It also maintains the size (the number of live elements, here 50), the capacity (the maximum achievable size without re-allocation, 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 constraints due to 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 are consecutive in memory, and the iterator order is equivalent to 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.

3.2. Experimental comparison

We have measured the time and memory space used by the computation of a (sequential) 3D Delaunay triangulation of 10 million random points using CGAL, only changing the containers used internally to store the vertices and cells. Using `list`, the program took 149 seconds and used 7744 MB of RAM, while using our `Compact_container` it took 116 seconds and used 5666 MB. The optimal memory size would have been 5098 MB, as computed by the number of vertices and cells times their respective memory sizes (32 and 72 bytes respectively). This means that the internal memory overhead was 52% for `list` and only 11% for `Compact_container`.

3.3. 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 manipulate it concurrently, using several threads. 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.

We extended the `Compact_container` class 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 maintained. 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.

3.4. Benchmark

Figure 2 shows a synthetic benchmark of the parallel `Compact_container` alone, by performing essentially parallel additions together with 20% of interleaved deletions, 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. We think that using it for geometric algorithms will prove it useful even with a lower numbers of elements, since significant computation takes place between the container updates.

4. Spatial sorting

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 [11] (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 for sorting points along a Hilbert space-filling curve as well as a BRIO [12].

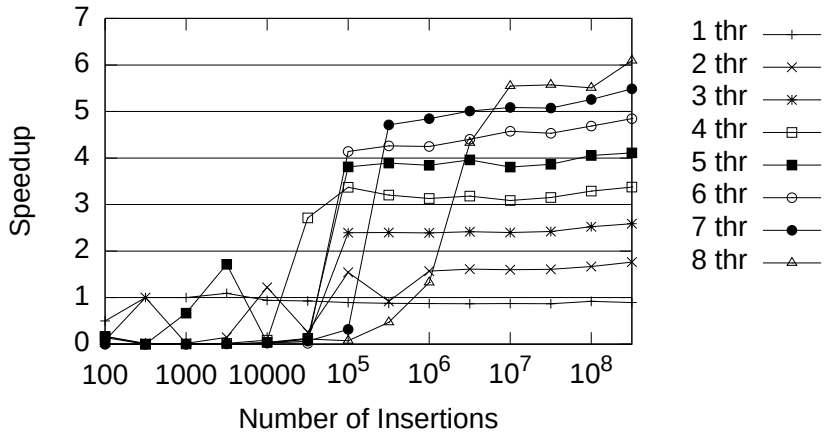


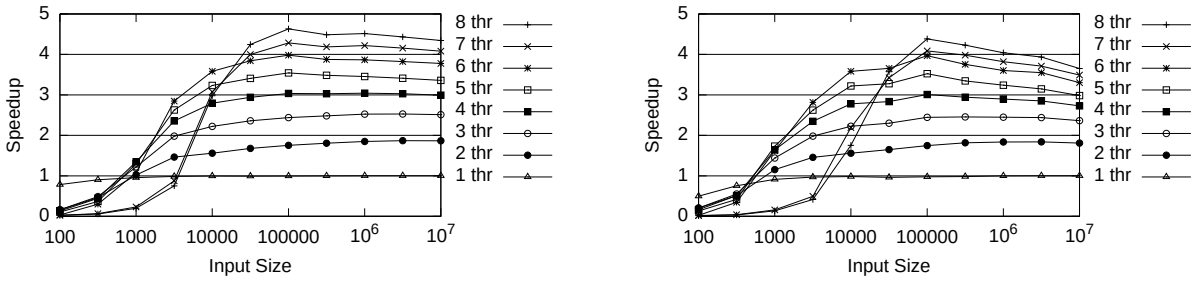
Figure 2: Speedups obtained for the compact container with additions and 20% of deletions.

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 virtual Hilbert curve.

Parallelizing this algorithm is straightforward. The divides for Hilbert sorting are done by the parallel `nth_element` function with the middle position as argument, the recursive subproblems are processed by newly spawned OpenMP tasks. Spawning is stopped as soon as the subproblem size gets below a configurable threshold size, in order to minimize parallelization overhead. When all spawned tasks are finished, the algorithm terminates.

For BRIO, the initial randomization is done using the parallel `random_shuffle`. In the sort phase, it calls the parallelized Hilbert sort as subroutine. Except from that, there is only trivial splitting, so parallelization is complete.



(a) 2D points.

(b) 3D points.

Figure 3: Scalability of our parallel spatial sort implementation for an increasing number of (a) two- and (b) three-dimensional random points.

4.1. Experimental results

The speedup (ratio of the running times between the parallel and sequential versions) obtained for 2D Hilbert sorting are shown in Figure 3a. 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, as presented in Figure 3b, 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.

5. Intersecting dD boxes

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.

5.1. Algorithm

We parallelize the algorithm proposed by Zomorodian and Edelsbrunner [13], which is already used for the sequential implementation in CGAL [14], and proven to perform well in practice.

Since axis-aligned boxes intersect if and only if their projected intervals intersect in all dimensions, the algorithms can proceed dimension by dimension, reducing the potentially

intersecting subsets of boxes. Two intervals intersect if and only if one contains the lower endpoint of the other. Thus, the algorithm takes the lower endpoints of the first set of intervals, and for each reports all intervals of the second set of intervals that contain it. This routine is called *stabbing*. By doing this vice-versa, all intersecting intervals are found.

To implement the stabbing efficiently over all dimensions, a complex data structure comprised of nested segment and range trees is proposed. However, its worst-case space consumption is $O(n \log^d n)$. Since this is unacceptable, the trees are not actually stored in memory, but constructed and traversed on the fly using a D&C algorithm, which needs only logarithmic extra memory (apart from the possibly quadratic output). For small subproblems below a certain cutoff size, a base-case quadratic-time algorithm checks for intersections.

As stated before, the problem is solved dimension by dimension, recursively. For each dimension, the input consists of two sequences: points and intervals (lower endpoints of the boxes and the boxes themselves projected to this dimension, respectively). Here, following the 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) to the right of m. 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.

As an exception, degenerated intervals and intervals spanning the full range are treated specially. Both 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 4.

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, there is a particular 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

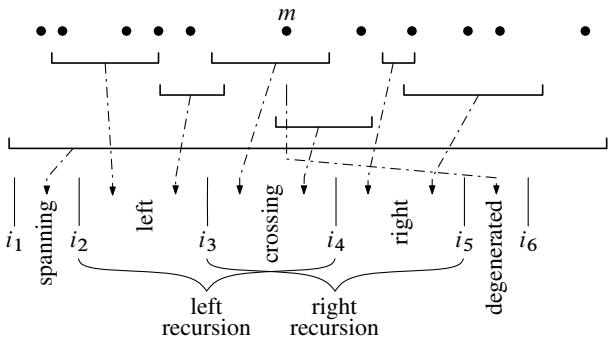


Figure 4: Partitioning the sequence of intervals.

is forbidden, even if read-only. Thus, we have to *copy* intervals. Note that 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 [14].

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. The ranges $[i_1, i_2)$ are the intervals spanning the whole region. They are handled separately. Ranges $[i_2, i_3)$ and $[i_4, i_5)$ are respectively the intervals for the left and right recursion steps only, $[i_3, i_4)$ correspond to the intervals for both the left and the right recursion steps, and $[i_5, i_6)$ 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 $[i_3, i_5)$. 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

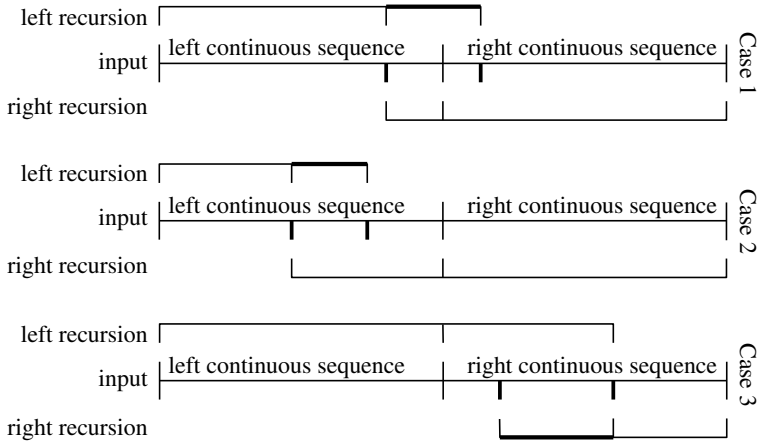


Figure 5: Treating the split sequence of intervals. Thick lines denote copied elements.

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 sketch: 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 5.

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.

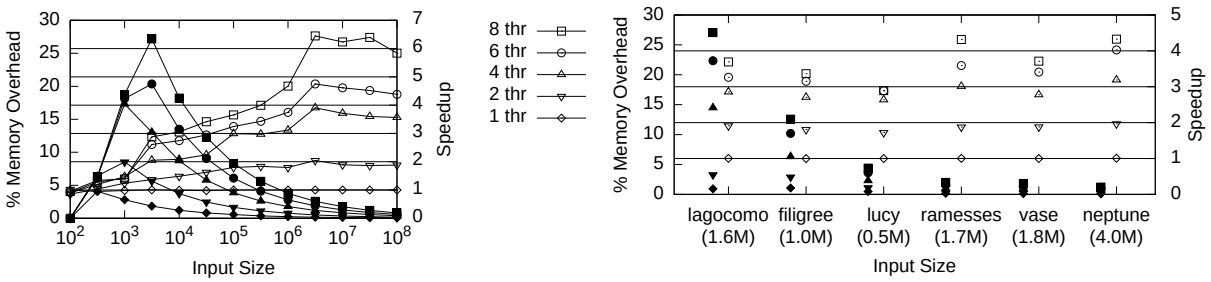
- Spawn a new task if there are less than a certain number of tasks t_{max} (tuning parameter) in the task queue. Since OpenMP does not allow to inspect its internal task queue, we have to count the number of currently active tasks manually, using atomic operations on a counter. This strategy can effectively 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 task spawning procedure where all three conditions must be fulfilled.

5.2. Experimental results

Three-dimensional boxes with integer coordinates were randomly generated as in [13] such that the expected number of intersections for n boxes is $n/2$. The memory overhead



(a) Randomly generated integer coordinates.

(b) Real-world data sets.

Figure 6: Results on the performance of our parallel algorithm for intersecting d-dimensional boxes. Speedup is denoted by empty marks, relative memory overhead by filled ones.

numbers refer to the *algorithmic* 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.

For the results in Figure 6a, we used $c_{min} = 100$, $t_{max} = 2 \cdot p$, and $s = 1 \cdot n$, where p 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. The speedups are quite good, reaching more than 6 for 8 cores, and being just below 4 for 4 threads. It is worth emphasizing that the sequential code performs the intersection of 10^6 boxes in 1.86s.

Figure 6b 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 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 stating that 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.

6.1. 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. [15]. 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 achieved only a 3-fold speedup for triangulating 8000 points on an nCUBE 2 with 8 processors. Cignoni et al. [16] 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 on 8 processors for 20000 random points.

Lee et al. [17], focusing on distributed memory systems, improved this algorithm by exploiting a projection-based partitioning scheme [18], 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 speedup around 6.5 on 8 processors for 10000 randomly distributed points. However, even their best partitioning method took 75% of the total elapsed time.

The method of Blelloch et al. [18] treats the 2D case using the well-known relation with

three-dimensional convex hulls. Instead of directly solving the 3D convex hull problem, another reduction step to the 2D lower hull problem is performed. 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 for uniformly distributed points on a shared-memory SGI Power Challenge.

To accelerate merging, the algorithm of Chen et al. [19] determines the set of triangles that may be affected by neighboring processors. Together, these triangles form the *affected zone* of an individual processor, which can be computed in $O(n)$ time. Experiments on an IBM SP2 cluster with 16 nodes shown that it did not scale linearly for uniform point distributions, but speedups grown for larger problems. The processing of 1–16 million points using 8 processors resulted in speedups of 5.6–6.0, respectively.

In geographical information systems, a terrain is usually represented as a triangulated irregular network (TIN), which in turn can be computed by using two-dimensional Delaunay triangulation algorithms. In this context, Puppo et al. [20] have designed a parallel algorithm for computing a TIN on massive distributed machines. It constructs an initial triangulation serially, and then adds new points in parallel so as to match terrain approximation restrictions. Since the resulting triangulation does not necessarily satisfy the empty circle property, the standard edge flip procedure [21] is concurrently applied to all non-Delaunay edges. Conflicts between neighboring triangles are solved by means of a mutual exclusion mechanism based on priorities corresponding to the quality of the approximation provided by each triangle. Authors reported an 80-fold speedup for triangulating 16K points on a Connection Machine CM-2 with 16K processors.

Further results on two-dimensional parallel algorithms avoiding complex D&C implementations were simultaneously presented by Chrisochoides and Sukup [22] and Okusanya and Peraire [23]. Both works tackled the problem of parallelizing a mesh generation algorithm, whose kernel was the Bowyer-Watson algorithm, for distributed memory systems. Based on preliminary results, Chrisochoides and Sukup [22] presumed that their algorithm would demonstrate linear scalability. Unfortunately, Okusanya and Peraire [23] did not present results on the parallel Bowyer-Watson performance, but only for a parallel implementation

of a recursive edge-swapping algorithm [24]. In this case, one million triangles were generated 3 times faster when using 8 processors of an IBM SP2 machine. These algorithms were afterwards extended to three-dimensions by Chrisochoides and Nave [25] and Okusanya and Peraire [26], who obtained speedups around 2.8 and 2.3, respectively, on 8 processors and producing roughly 2 million tetrahedra.

Early attempts on the parallelization of randomized incremental construction algorithms were reported by Kohout et al. [27]. They observed that topological changes caused by point insertion procedures are likely to be extremely local. When a thread needs to modify the triangulation, it first 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 reached speedups of 1.3 and 3.6 using 2 and 4 threads, respectively, on a four-socket Intel Itanium at 800MHz with 4MB of cache and 4GB of 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 devised by Blandford et al. [28]. It employs a compact data structure and follows a Bowyer-Watson approach, maintaining an association between uninserted points and their containing tetrahedra [29]. A coarse triangulation is sequentially built using a separate triangulator (Shewchuk's Pyramid [30]) 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 of RAM per processor, spending 6–8% of the total running time in Pyramid. Their work targeted very large triangulations (about $2^{30.5}$ points on 64 processors), as they also used 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 fit in 16GB of memory.

Recently, Antonopoulos et al. [31, 32] have evaluated multi-grained schemes for generating two-dimensional Delaunay meshes, targeting clusters built from simultaneous multi-

threaded (SMT) processors. At the top level, the algorithm constructs a coarse triangulation that is decomposed into sub-domains, which in turn are assigned to processors. Synchronizations across sub-domain boundaries are carried out during the meshing process at this level. In the medium-grained level, threads execute the Bowyer-Watson algorithm concurrently, checking conflicts between co-workers by means of hardware-supported atomic operations. The fine-grained implementation consists of computing the conflict regions induced by each point in parallel. Results shown that a combination between coarse- and medium-grained implementations provided the best performance gains, while the fine-grained scheme slowed the code down mainly due to synchronization overheads.

An emerging method for concurrency control is the *transactional memory*, which is usually supported by software, hardware, or a combination of both [33]. Kulkarni et al. [34] have considered the applicability of transactional memory for generating two-dimensional Delaunay meshes by iterative refinement. Based on theoretical arguments, they concluded that current transactional memory implementations may prevent parallelism at all due to ineffective scheduling of conflicting transactions and excessively conservative interference detection. In this direction, Scott et al. [35] evaluated two transactional memory back-ends and coarse- and fine-grained locks applied to the merging phase of a 2D D&C algorithm. The best solution was provided by coarse-grained locks, which was justified by the fact that transactions represented only a tiny part of the execution time. Altogether, transactional memory models are still in development and much of the work has been concentrated in reducing transaction overheads to make them competitive to lock-based alternatives [36].

6.2. Sequential framework

CGAL provides 2D and 3D incremental algorithms [37] for Delaunay triangulation, and a similar approach has also been implemented for d dimensions [38]. Points are inserted iteratively in a BRIO, doing a *locate step* followed by an *update step*. The locate step finds the cell c that contains q, by employing a *remembering stochastic walk* [39], which 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, i. e., by checking the empty sphere property for all neighbors of c in a breadth-first search. The conflicting cells are then removed, leaving a "hole", and the triangulation is updated by creating new cells connecting q to the vertices on the boundary of the "hole".

A vertex stores its coordinates and a pointer to one of the incident cells. A cell stores pointers to its vertices and neighbors, plus a three-valued enumeration, which memoizes their conflict status during the update step. Vertices and cells are themselves stored in two *compact containers* (see Section 3). Note that there is also a fictitious "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 vertices), therefore its address is stable, while a cell can be destroyed by subsequent insertions.

6.3. Parallel algorithm

We attack the problem of constructing $\mathcal{DT}(S)$ in parallel by allowing concurrent insertions into the same triangulation, dividing the input points over all threads. Our scheme is similar to those in [27, 28], 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 at the beginning. 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, one for each thread. 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 either in the vertices or in the cells.

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

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

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 an expectedly close starting point for the locate step. Therefore, each thread divides its own range again into several parts of almost equal size, 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.

Load Balancing. As the insertion time of points may vary greatly depending on their location, and the work load of threads has geometric locality, some threads may take a much longer time to finish their work share. To counter the effects of such bad load balancing, we apply *work stealing* dynamically. A thread which is out of work steals half of the remaining points from a part of a random other thread. This is done for each part, so interleaving is functional again afterwards. We use only atomic operations, with no explicit communication

with the victim thread. To lower the overhead of the atomic operations while inserting the points, we reserve and steal chunks of points, e. g., 100.

Vertex-locking Strategies. There are several ways of choosing the vertices to lock. The *simple-vertex-locking 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 also propose an *improved-vertex-locking 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 that this very vertex be locked. This rule implies that a thread can change a cell only 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 likely contented 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. This allows to exclude the furthest vertex from q, with the following caveat: all vertices whose incident cell pointer points 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 *only* these) 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, i. e., changing the convex hull. 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. However, computing distances takes time, which may or may not be won back. Therefore it is also possible to simply pick the vertices to lock in any order, whichever is the most convenient and efficient.

So far we have considered simple exclusive locks, but there also exists another kind of locks, called *shared locks* or read-write locks in the literature, which are a bit more costly to handle, but provide the following interesting property. A shared lock can be held by an unbounded number of *reader* threads simultaneously, but it can alternatively be held exclusively by one *writer* thread. We say that the locks are r-locked and w-locked, respectively. Using this semantic, we can now build the *shared-vertex-locking strategy*.

Recall what the constraints are: (a) there can be at most one thread writing to a cell, (b) there cannot be a thread writing to a cell with others reading it, (c) the vertex whose cell pointer points to this cell needs to be locked, and (d) a thread having write access to a cell should prevent other threads from having write access to its neighbors, that is, it should have read access to them. It follows easily from these constraints that the optimal number of vertices locked are the following. A thread is allowed to write to a cell as long as it holds on its vertices at least: $\lceil \frac{d+1}{2} \rceil$ w-locks, one r-lock, and w-locks to all vertices whose cell pointers point to this cell. A thread is allowed to read a cell as long as it holds on its vertices at least: $\lfloor \frac{d+3}{2} \rfloor$ r-locks, and if there is at least one vertex whose cell pointer points to this cell, one of them must be w-locked until the number of r-locks is reached.

This strategy gives even more freedom and allows more threads to share the triangulation, especially in higher dimension.

Cell-locking Strategy. The *cell-locking strategy* considers cells instead of vertices. We re(-ab-)use the three-valued enumeration stored in the cells for the purpose of locking. In the sequential case, this integer can have the values 0, 1, and 2. In the parallel case, we simply replace 1 and 2 by values which differ for each thread, for example $1+2\times$`tid` and $2+2\times$`tid`, with `tid` an integer identifying the thread.

When performing point location, it is enough to lock the current cell for testing the orientation predicates, then lock the next cell before releasing the lock on the current one. For the update step, it is enough to only lock the cells as needed, so those in conflict or on the boundary of the conflict hole will be locked.

Similarly to the vertex locking strategies, some care must be taken when "entering" the triangulation, that is, when getting a pointer to a cell incident to a vertex. Somehow, this operation logically requires locking the vertex, and it must return a pointer to a locked cell. Fortunately, this particular case can be achieved efficiently using the following trick: we atomically set this pointer to `NULL` in order to indicate that the vertex is locked, and we take care of using such locking on the vertices incident to the cells which are about to be destroyed.

This strategy has several advantages over locking the vertices. First, it does not require more memory than the sequential case, the data structure can in fact be left unchanged. Second, the number of locking operations is reduced. Third, it does not trigger any artificial sharing issue. Finally, the algorithm follows a flow which is more similar to the sequential case, and this allows to share the code more easily using generic programming techniques.

6.4. Experimental results

We have implemented our parallel algorithm based on the 3D CGAL code [40] with all described locking strategies. We carried out experiments on six different point sets, including three synthetic and three real-world data. The synthetic data consist of evenly distributed points in a cube, 10^6 points lying on the surface of an ellipsoid of axes lengths 1, 2 and 3, and 10^4 points equally distributed on two non-parallel lines. The latter case generates a triangulation of quadratic complexity, and we only tested it with the *cell-locking* strategy.

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 15.84s.

Figures 7a, 7b through 10a, 10b show the achieved speedups. Variables subscripted with e denote values for the "empty" lock traits, i. e., without actually locking.

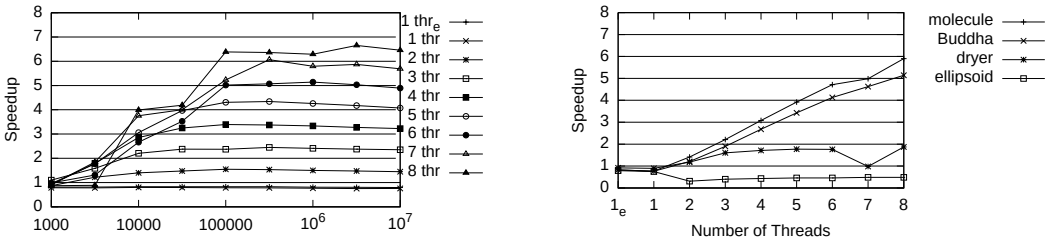
Overall, the *cell-locking* strategy clearly wins over the others. It is able to reach a speedup of 7 with 8 cores for 10^5 random points or more, and it does also well for the other data sets, except for two-lines, which could be expected.

Among the *vertex-locking* strategies, the *improved-vertex-locking* appears to be the best compromise. The relative cleverness of the *shared-vertex-locking* unfortunately cannot recover the overall overhead that its logic adds at such a low level. Even though the simple variant achieves the best speedup for random points, it shows its limitation on surfacic data sets, where the speedup is at most 2 and even much less than 1 for the ellipsoid. The convexity of the latter point set exhibits a lot of contention with this strategy, because of the sharing of the infinite vertex.

For the *simple-vertex-locking* and *cell-locking* strategies, Figures 11 and 12 give more details on the time spent in various steps of the algorithm: spatial sort, bootstrap, locate and update. All steps achieve good scalability in the random case, happy Buddha and molecule. However, it gets much worse for the dryer handle and ellipsoid instances when using the *simple-vertex-locking* strategy.

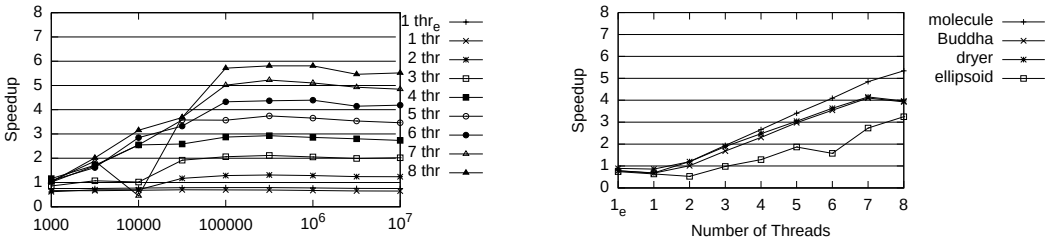
6.5. Tuning parameters

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 penalize small data sets.



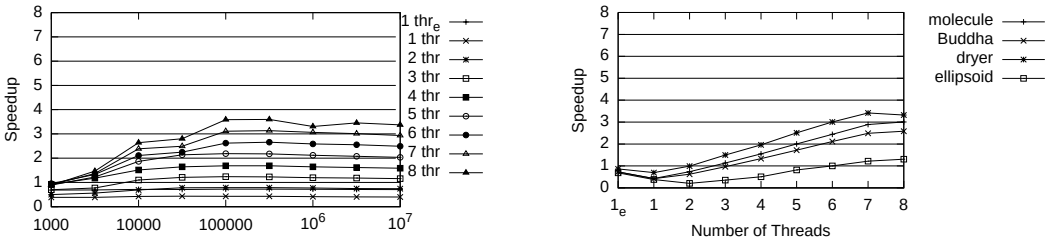
(a) Random points. (b) Other data sets.

Figure 7: Speedups achieved using the *simple-vertex-locking* strategy.



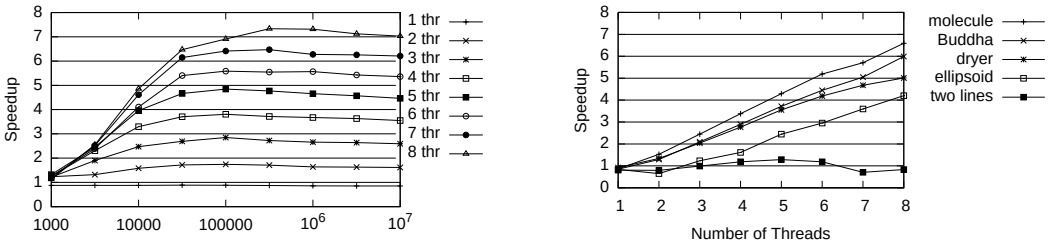
(a) Random points. (b) Other data sets.

Figure 8: Speedups achieved using the *improved-vertex-locking* strategy.



(a) Random points. (b) Other data sets.

Figure 9: Speedups achieved using the *shared-vertex-locking* strategy.



(a) Random points. (b) Other data sets.

Figure 10: Speedups achieved with the *cell-locking* strategy.

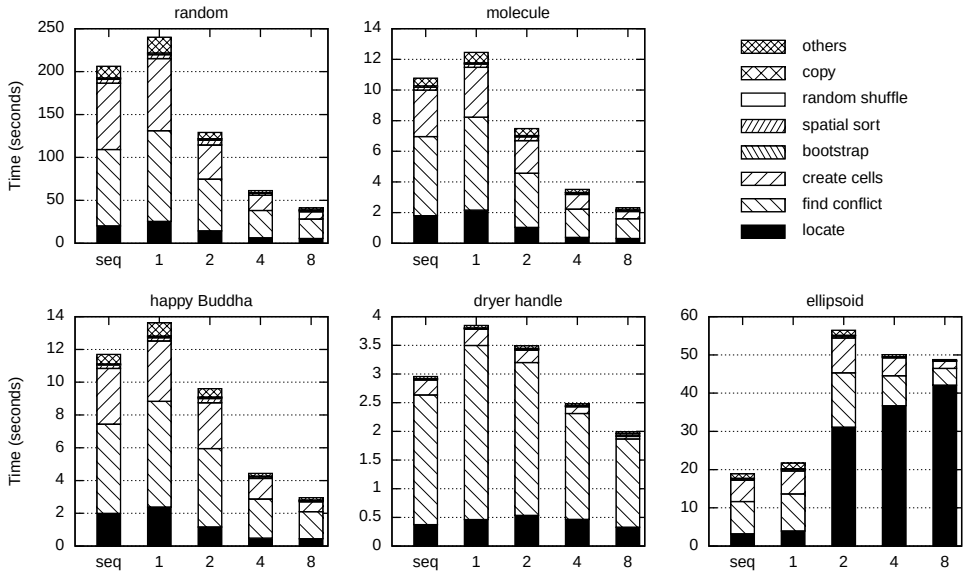


Figure 11: Breakdown of the running times for the sequential (seq) and parallel algorithms for 1, 2, 4 and 8 threads using the *simple-vertex-locking* strategy. In all cases, we used bootstrap size equal to $100p$ and interleaving degree 2.

Based on the *simple-vertex-locking* strategy, 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. Figure 13 illustrates the performance degradation introduced by using this mechanism, even with only one vertex per lock. Note that this effect gets more pronounced when incrementing the number of vertices locked at once.

7. Conclusion and future work

We have described parallel algorithms for three fundamental geometric problems, targeted at shared-memory multi-core architectures, which are ubiquitous nowadays. These are 2/3-dimensional spatial sorting of points, d-dimensional axis-aligned box intersection computation, and bulk insertion of points into 3D Delaunay triangulations. Experiments show significant speedup over their already efficient sequential original counterparts, as well as good comparison to previous work for the Delaunay computation, for moderately sized

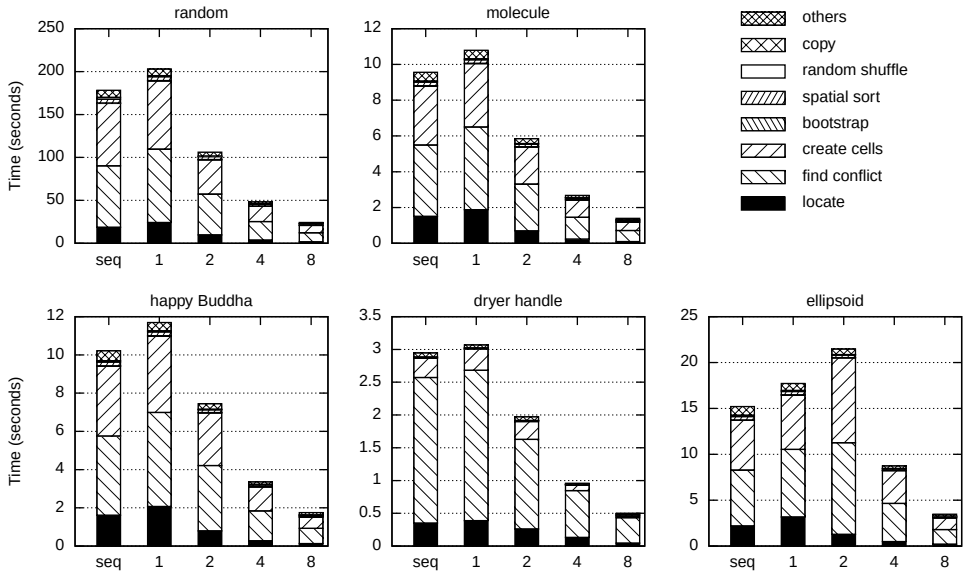


Figure 12: Breakdown of the running times for the sequential (seq) and parallel algorithms for 1, 2, 4 and 8 threads using the *cell-locking* strategy. In all cases, we used bootstrap size equal to $100p$ and interleaving degree 2.

problems.

We focused on the parallelism for multi-core and at the algorithmic level, but it might also be interesting to investigate parallelism in the evaluation of the geometric predicates. One could for example envision using either multi-core or SIMD vector units parallelism, depending on the dimension and algebraic degree of the predicate, and whether floating-point or multi-precision arithmetics are used.

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

Moreover, given the high difficulty of properly benchmarking such complex implementations from all interesting angles, it could be a good idea to organize some friendly competition around the theme of parallel Delaunay triangulations at some point.

	Bootstrap Size				Interleaving Size			
# threads	$8p$	$32p$	$128p$	$512p$	1	2	4	8
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)
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)
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)

Table 1: Speedups and percentage of retreated vertices, indicated between parentheses, 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). When analyzing the bootstrap (interleaving, resp.) the interleaving size (bootstrap size, resp.) was maintained equal to 2 ($100p$, resp.). The indirection overhead was computed using bootstrap and interleaving sizes equal to $100p$ and 2, respectively. Both interleaving and bootstrap tests were conducted without the lock sharing mechanism.

References

[1] A. L. Chow, Parallel algorithms for geometric problems, Ph.D. thesis, University of Illinois, Urbana-Champaign, IL, USA (1980).

[2] A. Aggarwal, B. Chazelle, L. J. Guibas, C. Ó'Dúnlaing, C.-K. Yap, Parallel computational geometry, Algorithmica 3 (1–4) (1988) 293–327.

[3] S. G. Akl, K. A. Lyons, Parallel computational geometry, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[4] CGAL, Computational Geometry Algorithms Library, `http://www.cgal.org/`.

[5] Y. Liu, J. Snoeyink, A comparison of five implementations of 3D Delaunay tessellation, in: J. E. Goodman, J. Pach, E. Welzl (Eds.), Combinatorial and Computational Geometry, Vol. 52 of MSRI Publications, Cambridge University Press, 2005, pp. 439–458.

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

[7] OpenMP Architecture Review Board, OpenMP Application Program Interface, Version 3.0 (May 2008).

[8] J. Singler, B. Kosnik, The GNU libstdc++ parallel mode: software engineering considerations, in: Proc. 1st Intl. Worksh. Multicore Software Eng., 2008, pp. 15–22. doi:10.1145/1370082.1370089.

[9] J. Singler, P. Sanders, F. Putze, MCSTL: The multi-core standard template library, in: Proc. 13th Eur. Conf. Parallel and Distributed Comp., 2007, pp. 682–694.

[10] M. Hoffmann, L. Kettner, S. Pion, R. Wein, STL extensions for CGAL, in: CGAL Editorial

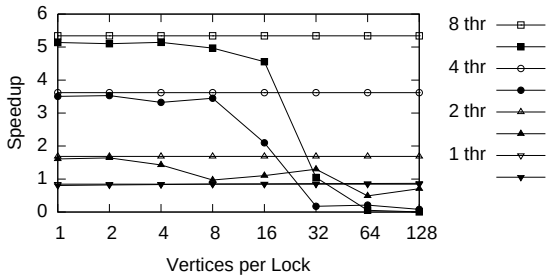


Figure 13: The achieved speedups of our parallel algorithm with and without the lock sharing scheme enabled. Filled marks indicate lock sharing, while unfilled ones correspond to unshared locks. For both versions, we triangulated 131K random points with bootstrap and interleaving sizes equal to $100p$ and 2, respectively.

Board (Ed.), CGAL Manual, 3.5 Edition, 2009, `http://www.cgal.org/Manual/3.5/doc_html/cgal_manual/packages.html#Pkg:StlExtension`.

[11] N. Amenta, S. Choi, G. Rote, Incremental constructions con BRIO, in: Proc. 19th ACM Symp. Comp. Geom., 2003, pp. 211–219. doi:10.1145/777792.777824.

[12] C. Delage, Spatial sorting, in: CGAL Editorial Board (Ed.), CGAL Manual, 3.5 Edition, 2009, `http://www.cgal.org/Manual/3.5/doc_html/cgal_manual/packages.html#Pkg:SpatialSorting`.

[13] A. Zomorodian, H. Edelsbrunner, Fast software for box intersections, Intl. J. Comp. Geometry Appl. 12 (1-2) (2002) 143–172.

[14] L. Kettner, A. Meyer, A. Zomorodian, Intersecting sequences of dD iso-oriented boxes, in: CGAL Editorial Board (Ed.), CGAL Manual, 3.5 Edition, 2009, `http://www.cgal.org/Manual/3.5/doc_html/cgal_manual/packages.html#Pkg:BoxIntersectionD`.

[15] P. Cignoni, C. Montani, R. Perego, R. Scopigno, Parallel 3D Delaunay triangulation, Comput. Graph. Forum 12 (3) (1993) 129–142.

[16] P. Cignoni, D. Laforenza, C. Montani, R. Perego, R. Scopigno, Evaluation of parallelization strategies for an incremental Delaunay triangulator in E^3, Conc. Pract. and Exp. 7 (1) (1995) 61–80.

[17] S. Lee, C.-I. Park, C.-M. Park, An improved parallel algorithm for Delaunay triangulation on distributed memory parallel computers, Parallel Processing Letters 11 (2/3) (2001) 341–352.

[18] G. E. Blelloch, J. C. Hardwick, G. L. Miller, D. Talmor, Design and implementation of a practical parallel Delaunay algorithm, Algorithmica 24 (3) (1999) 243–269. doi:10.1007/PL00008262.

[19] M.-B. Chen, T.-R. Chuang, J.-J. Wu, Parallel divide-and-conquer scheme for 2D Delaunay triangulation, Conc. and Comp. Pract. and Exp. 18 (12) (2006) 1595–1612.

[20] E. Puppo, L. S. Davis, D. De Menthon, Y. A. Teng, Parallel terrain triangulation, Intl. J. GIS 8 (2)

(1994) 105–128.

[21] C. L. Lawson, Software for C^1 surface interpolation, in: J. Rice (Ed.), Mathematical Software III, Academic Press, 1977, pp. 161–194.

[22] N. Chrisochoides, F. Sukup, Task parallel implementation of the Bowyer-Watson algorithm, in: Proc. 5th Intl. Conf. Num. Grid Generation, 1996, pp. 773–782.

[23] T. Okusanya, J. Peraire, Parallel unstructured mesh generation, in: Proc. 5th Intl. Conf. Num. Grid Generation, 1996, pp. 719–729.

[24] P. J. Green, R. Sibson, Computing Dirichlet Tessellations in the Plane, Comput. J. 21 (2) (1978) 168–173.

[25] N. Chrisochoides, D. Nave, Parallel Delaunay mesh generation kernel, International Journal for Numerical Methods in Engineering 58 (2) (2003) 161 – 176.

[26] T. Okusanya, J. Peraire, 3D parallel unstructured mesh generation, in: Trends in Unstructured Mesh Generation, Vol. 220 of Applied Mechanics Division, ASME, 1997, pp. 109–115.

[27] J. Kohout, I. Kolingerová, J. Žára, Parallel Delaunay triangulation in E^2 and E^3 for computers with shared memory, Parallel Computing 31 (5) (2005) 491–522.

[28] D. K. Blandford, G. E. Blelloch, C. Kadow, Engineering a compact parallel Delaunay algorithm in 3D, in: Proc. 22nd Symp. Comp. Geom., 2006, pp. 292–300. doi:10.1145/1137856.1137900.

[29] K. L. Clarkson, P. W. Shor, Applications of random sampling in computational geometry, II, Disc. & Comp. Geom. 4 (5) (1989) 387–421.

[30] J. R. Shewchuk, Tetrahedral mesh generation by Delaunay refinement, in: Proc. 14th ACM Symp. Comp. Geom., 1998, pp. 86–95.

[31] C. D. Antonopoulos, F. Blagojevic, A. N. Chernikov, N. P. Chrisochoides, D. S. Nikolopoulos, Algorithm, software, and hardware optimizations for Delaunay mesh generation on simultaneous multithreaded architectures, Journal of Parallel and Distributed Computing 69 (7) (2009) 601 – 612.

[32] C. D. Antonopoulos, F. Blagojevic, A. N. Chernikov, N. P. Chrisochoides, D. S. Nikolopoulos, A multigrain Delaunay mesh generation method for multicore SMT-based architectures, Journal of Parallel and Distributed Computing 69 (7) (2009) 589 – 600.

[33] M. Herlihy, N. Shavit, The Art of Multiprocessor Programming, Morgan Kaufmann, Burlington, MA, USA, 2008.

[34] M. Kulkarni, L. P. Chew, K. Pingali, Using transactions in Delaunay mesh generation, in: Proc. 1st Worksh. Transact. Memory Workloads, 2006.

[35] M. L. Scott, M. F. Spear, L. Dalessandro, V. J. Marathe, Delaunay triangulation with transactions and barriers, in: Proc. 10th Intl. Symp. Workload Characterization, 2007, pp. 107–113.

[36] C. Caşcaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, S. Chatterjee, Software transac-

tional memory: why is it only a research toy?, Commun. ACM 51 (11) (2008) 40–46.

[37] J.-D. Boissonnat, O. Devillers, S. Pion, M. Teillaud, M. Yvinec, Triangulations in CGAL, Comp. Geom. Theory & Appl. 22 (2002) 5–19. doi:10.1016/S0925-7721(01)00054-2.

[38] J.-D. Boissonnat, O. Devillers, S. Hornus, Incremental construction of the Delaunay triangulation and the Delaunay graph in medium dimension, in: Proc. 25th ACM Symp. Comp. Geom., 2009, pp. 208–216. doi:10.1145/1542362.1542403.

[39] O. Devillers, S. Pion, M. Teillaud, Walking in a triangulation, Intl. J. Found. Comput. Sci. 13 (2) (2002) 181–199.

[40] S. Pion, M. Teillaud, 3D triangulations, in: CGAL Editorial Board (Ed.), CGAL Manual, 3.5 Edition, 2009, `http://www.cgal.org/Manual/3.5/doc_html/cgal_manual/packages.html#Pkg:Triangulation3`.