ELSEVIER

Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Engineering 00 (2017) 000–000

Procedia Engineering

www.elsevier.com/locate/procedia

26th International Meshing Roundtable, IMR26, 18-21 September 2017, Barcelona, Spain

Toward one billion tetrahedra per minute

Célestin Marot[a,*], Jeanne Pellerin[a], Jonathan Lambrechts[a], Jean-François Remacle[a]

[a]*Université catholique de Louvain, iMMC, Avenue Georges Lemaitre 4, bte L4.05.02, 1348 Louvain-la-Neuve, Belgium*

Abstract

In this research note, we propose a new scalable parallelization scheme to generate the Delaunay tetrahedrization of a given set of points. Our first contribution is a very efficient serial implementation. From this base we developed a multi-threaded version of the Delaunay kernel that concurrently insert points into the tetrahedrization. We use the Hilbert curve coordinates to distribute the work between threads. Our strategy is free from heavy synchronization overhead. The key idea to manage conflicts is to modify the partitions by enlarging the space filled by the Hilbert curve. Our implementation is very simple and outperforms previous existing methods.

Keywords: Delaunay; mesh generation; tetrahedral mesh; parallel; Hilbert; multithread

1. Introduction

As the size of finite element meshes grows, efficient scalable parallel meshing algorithms taking advantage of the capabilities of recent architectures are needed. Indeed, today's meshing algorithms are not well-adapted to new multicore architectures with a large common shared memory but relatively low performance individual computing cores. In this research note, we propose a new parallelization scheme for the Delaunay kernel that is scalable and generates one billion of tetrahedra in 82 seconds on a 64 core Intel® Xeon Phi™7210 @ 1.30GHz. Most of the recently proposed parallel Delaunay methods manage conflicts between threads with some locking strategy [1–5]. Remacle[6] proposed a simpler two-level parallelization of the Delaunay strategy that only uses OpenMP barriers. However, synchronization overheads prevent the good scalability of these approaches.

We first propose a very efficient serial implementation of the Delaunay tetrahedrization (§2). From this base we build a scalable multi-threaded parallelization of the Delaunay kernel (§4). Our strategy to decompose the domain is based on very fast initial sorting of the points to triangulate (§3). To detect thread conflicts we use a simple coloring scheme of the points, and avoid complicated locking strategies, or synchronization barriers. To manage points at the interface of several threads, our idea is to modify the point thread partitioning, displacing the problematic areas to already processed zones. Timings show that our serial implementation is more than 2 times faster than Tetgen[7], that our parallel algorithm is scalable and performs very well on highly different machines: (i) a laptop with a 4 core Intel® Core™i7-6700HQ @ 2.60GHz processor with 16GB of memory and (ii) an Intel Knights Landing© machine with a

[*] Corresponding author. Tel.: +32 10472357
E-mail address: celestin.marot@uclouvain.be

	i7-6700HQ		Xeon Phi	
#vertices	TetGen	Ours	TetGen	Ours
10K	0.060	0.031	0.35	0.20
100K	0.60	0.23	3.77	1.03
1M	6.34	2.09	39.14	8.73
10M	67.92	20.79	419.77	85.11
OilPump 536,538	3.80	1.42	20.16	6.03
Neptune 205,835	1.48	0.55	7.73	2.42

Table 1: Timings in seconds of our serial 3D Delaunay tetrahedrization and comparison with Tetgen [7]. The first four input point sets are randomly generated points in a unit cube. OilPump and Neptune models are from the aim@shape repository.

	i7-6700HQ		Xeon Phi	
#vertices	qsort	radix	qsort	radix
10K	0.0024	0.0018	0.020	0.0034
100K	0.022	0.0039	0.24	0.0058
1M	0.28	0.024	1.43	0.0079
10M	3.16	0.23	15.04	0.027

Table 2: Comparison of our radix sort implementation with the qsort function of the C standard library. The randomly generated input vertices (32 bytes structures) are sorted according to their Hilbert coordinates (4 bytes keys). All timings are in seconds.

64 core Intel® Xeon Phi™7210 @ 1.30GHz and 192GB of memory. The code for the serial Delaunay tetrahedrization has about one thousand lines (not including geometric predicates [8]) and the parallelization adds up three hundreds lines. Our C implementation is open-source and will be available in Gmsh (`www.gmsh.info`).

2. Serial Delaunay Tetrahedrization

Given a set S of n points in R^3, its Delaunay triangulation (tetrahedrization) is defined as the tetrahedrization such that the circumsphere of each tetrahedra does not contain any point of S. In 3D, the fastest algorithms to build the Delaunay triangulation of S are based on the incremental insertion of the points. When inserting a point in the tetrahedrization, (i) all the tetrahedra whose circumsphere contains the point in its interior are determined and deleted creating a cavity, (ii) this star-shaped cavity is triangulated taking into account the new point. The last step is to compute the adjacencies between the new tetrahedra.

Our serial implementation is based on state of the art schemes that have been proven to be the most efficient. The overall algorithm is similar to the one of Tetgen [7], however our implementation is significantly faster than Tetgen (Table 1). The first reason is that we optimized key steps of the Delaunay kernel. For example, it is possible to accelerate the test checking if the circumsphere of a tetrahedron contains a point or not, the *insphere* test, by pre-computing sub-determinants for each tetrahedra. Another important tuning is a very fast computation of the adjacencies between the new tetrahedra filling a cavity by vectorizing part of the operations. The second reason is that our implementation is more specialized and much shorter than Tetgen's. Our code of about 1,000 lines of C generates only 3D Delaunay triangulations, this is to compare to the 35,000 lines of Tetgen and the management of constrained (non-Delaunay) triangulations.

3. Fast Spatial Sorting

The speed of the incremental Delaunay algorithm we are using depends on the insertion order of the points. Combining a Biased Randomized Insertion Order method (BRIO) [9] with a sorting of the points along a space filling Hilbert curve is the most efficient scheme for Delaunay tetrahedrization. Two consecutive vertices on the Hilbert curve will usually be close to each other (see Fig. 1a and Fig. 2a). The Hilbert coordinate of a point is a unsigned integer which number of bits is 3 times the number of iterations of the Hilbert curve. Hilbert coordinates of all vertices can be computed using multiple threads and vectorization.

To sort the points according to their Hilbert coordinate, we developed a new parallel implementation of radix sort. Radix sorting is a non-comparative sorting algorithm that runs in $O(n)$ time. Its principle is to group integer keys by individual digits that have the same significant position and value, see e.g. [10]. Our parallel implementation uses

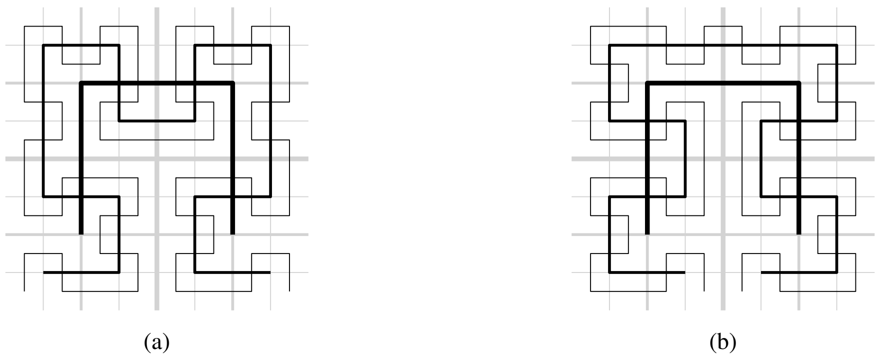
Fig. 1: Three iteration of the construction of the Hilbert (a) and Moore (b) curves.

some vectorization and is optimized to use the number of bits corresponding to the desired precision, i.e. the number of iterations on the Hilbert curve. We use a quite large radix (up to 2048) leading to major improvements of the performances (Table 2). Ten million points are sorted in 0.23 seconds on a standard laptop and one billion points in 5 seconds on a Xeon-Phi machine. This is much more efficient than the C/C++ standard library sorting functions like std::sort and qsort, see the comparisons in Table 2.

4. Parallelisation of the Delaunay Tetrahedrization

4.1. Strategy

To parallelize the Delaunay triangulation kernel, we allow concurrent insertions of points into the same triangulation. A starting triangulation is built by sequentially inserting the first few thousands points. The other points are then inserted in rounds of increasing size, each round being six times larger than the previous one. The m vertices of each round are sorted by their Hilbert coordinates (§3). The work between the n threads is then split by assigning the first m/n points to the first thread, the following m/n points to the second thread, etc. This subdivision of the Hilbert curve is also used to color already inserted vertices. Afterwards, a point can be inserted only if the vertices of its cavity have the same color, i.e. if they have their Hilbert coordinates in the same range. This is sufficient to avoid all possible data race issues. The number of points that can be inserted without any conflict depends highly on the size of the current tetrahedrization. Typically, when dealing with millions of points, success rate reaches 85% at the last insertion round.

4.2. Inserting conflicting vertices

The first method one can think of to insert the remaining vertices is to insert them sequentially. This is however prohibitively expensive, and we explored several other approaches.

Reduction. Using a reduction by recursively merging pairs of thread partitions to completely eliminate boundaries is a slightly better idea. However the fraction of solved conflicts will be very low and this strategy is not scalable on more than a dozen of threads. Take for example a domain subdivided into many cubes (one cube for one thread). Each of them has 6 boundaries on which conflicting vertices lie. At the first merge, one boundary is removed, so that $\frac{1}{6}$ of the remaining vertices are inserted. At the second merge, only $\frac{1}{6}$ of the remaining $\frac{5}{6}$ vertices are inserted.

Hilbert Curve Shifting. A second idea to obtain different partitions is to shift the Hilbert curve, then coordinates are shifted and partitions are modified. The issue is that shifting does not significantly modify the partition boundaries. Indeed the Hilbert curves are inherently separated in each dimension at the center of the bounding box (Fig. 1a).

Bounding Box Expansion. Another, more indirect, strategy to change the Hilbert coordinates is to modify the bounding box that the Hilbert curve fills (Fig. 2b). All Hilbert coordinates are computed anew for all the tetrahedrization

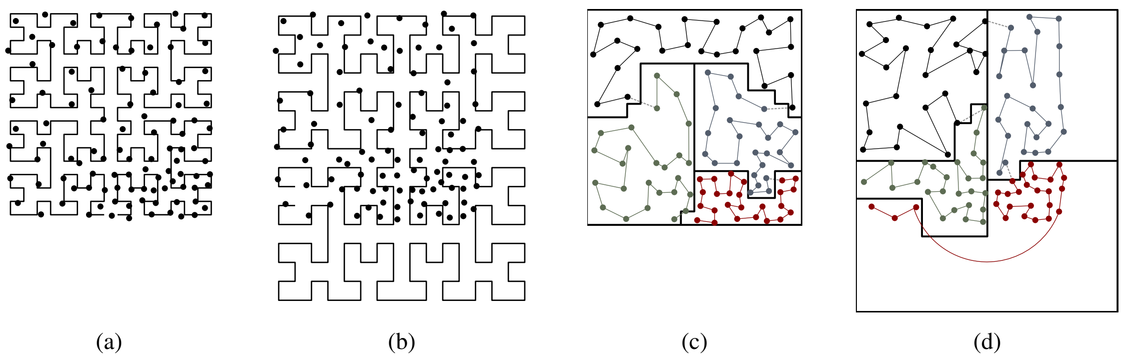

Fig. 2: To partition in two different ways a set of points, boxes of two sizes are used to compute the Hilbert curve. (a) the fitted bounding box (b) the enlarged bounding box. Partitioning both curves to dispatch the work between 4 threads gives different partitions (c) and (d) such that conflicting points in one of them will not be for the other.

vertices and all points to insert. The points to insert are then sorted according to these new coordinates and the partitions are modified. To minimize common boundaries with the first partitioning, we also apply a shift to the curve (Fig. 2). This allows to insert an important part of the vertices that were in conflict.

Implementation. Our implementation is adaptive. Depending on the success rate α (percentage of conflicting points inserted) and the number of threads n, we use a sequential insertion when $\alpha \leq \frac{1}{n}$, a reduction scheme when $\frac{1}{n} < \alpha \leq \frac{1}{6}$, and a bounding box expansion by default. Note that in our implementation we used a modified Hilbert curve, the Moore curve, that loops (Fig 1b).

4.3. *Performances of the Parallel Tetrahedrization*

We tested our parallel algorithm for various number of vertices on two highly different machines: a laptop with a 4 core Intel® Core™i7-6700HQ @ 2.60GHz[1] processor with 16GB of memory and an Intel Knights Landing© machine with a 64 core Intel® Xeon Phi™7210 @ 1.30GHz with 192GB of memory.

Although there are some vectorized parts in our implementation, the core of the Delaunay algorithm is still not suited for architectures heavily relying on SIMD instructions. However, our implementation is still very competitive on the Xeon Phi, a machine that usually require full vectorization of the program in order to overcome its slow cores (Fig. 3). We expect our implementation to reach the billion tetrahedra per second on a faster machine. Our tests on both machines show the very good scaling of our parallel algorithm as long as the number of threads is inferior to the number of cores (Fig. 4). Indeed, our implementation does not benefit from hyper-threading on the Xeon Phi.

5. Conclusion

We proposed an efficient serial 3D Delaunay triangulation. We also implemented a very effective spatial sorting algorithm that combines a fast way to compute Hilbert coordinates with a state of the art radix sort. Then, we designed a scalable parallel algorithm that partitions the domain taking advantage of our fast spatial sorting.

[1] 3.50Ghz in Turbo Boost

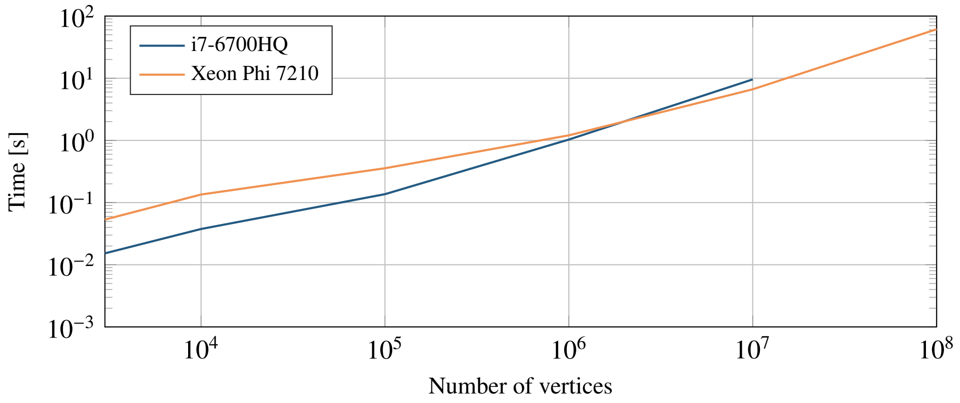

Fig. 3: Performances of our 3D parallel Delaunay kernel for points uniformly distributed in a unit cube

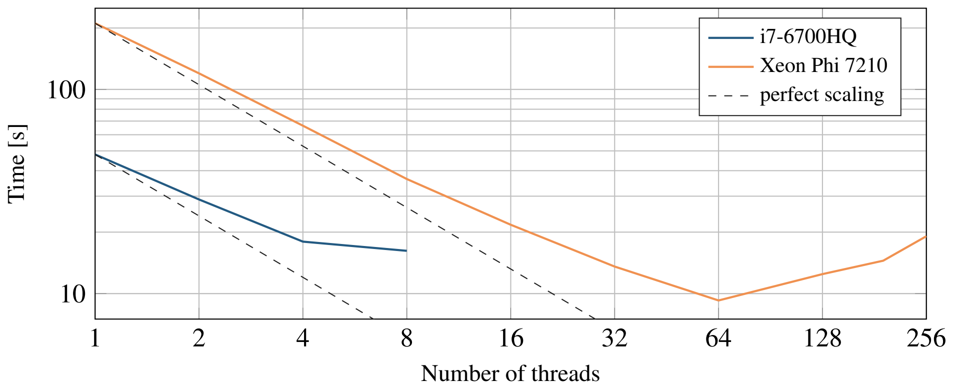

Fig. 4: Scaling of our 3D parallel Delaunay kernel for 15 million vertices (about 100 million tetrahedra).

Acknowledgements

This research is supported by the European Research Council (project HEXTREME, ERC-2015-AdG-694020).

References

[1] J. Kohout, I. Kolingerová, J. Žára, Parallel Delaunay triangulation in E2 and E3 for computers with shared memory, Parallel Computing 31 (2005) 491–522.
[2] N. Chrisochoides, D. Nave, Parallel Delaunay mesh generation kernel, International Journal for Numerical Methods in Engineering 58 (2003) 161–176.
[3] D. K. Blandford, G. E. Blelloch, C. Kadow, Engineering a compact parallel delaunay algorithm in 3d, ACM Press, 2006, p. 292.
[4] V. H. Batista, D. L. Millman, S. Pion, J. Singler, Parallel geometric algorithms for multi-core computers, Computational Geometry 43 (2010) 663–677.
[5] S. Lo, 3d Delaunay triangulation of 1 billion points on a PC, Finite Elements in Analysis and Design 102-103 (2015) 65–73.
[6] J.-F. Remacle, A two-level multithreaded Delaunay kernel, Computer-Aided Design 85 (2017) 2–9.
[7] H. Si, TetGen, a Delaunay-Based Quality Tetrahedral Mesh Generator, ACM Transactions on Mathematical Software 41 (2015) 1–36.
[8] J. Shewchuk, Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates, Discrete & Computational Geometry 18 (1997) 305–363.
[9] N. Amenta, S. Choi, G. Rote, Incremental constructions con BRIO, ACM Press, 2003, p. 211.
[10] M. Zagha, G. E. Blelloch, Radix sort for vector multiprocessors, in: Proceedings of the 1991 ACM/IEEE conference on Supercomputing, ACM, 1991, pp. 712–721.