International Journal of Networking and Computing – www.ijnc.org, ISSN 2185-2847
Volume 8, Number 2, pages 240–253, July 2018

Synchronizing Parallel Geometric Algorithms on Multi-Core Machines

Joel Fuentes
Department of Computer Science and Information Technologies
Universidad del Bío-Bío
Chillán, Chile

Fei Luo
School of Information and Engineering
East China University of Science and Technology
Shanghai, China

and

Isaac D. Scherson
Donald Bren School of Information and Computer Sciences
University of California, Irvine
Irvine, USA

Received: February 1, 2018
Revised: May 4, 2018
Accepted: June 2, 2018
Communicated by Fukuhito Ooshita

Abstract

A thread synchronization mechanism dubbed Spatial Locks for parallel geometric algorithms is presented. We show that Spatial Locks ensure thread synchronization on geometric algorithms that perform concurrent operations on geometric surfaces in two-dimensional or three-dimensional space. The proposed technique respects the fact that these operations follow a certain order of processing, i.e. priorities. Parallelizing these kinds of geometric algorithms using Spatial Locks requires only a simple parameter initialization, rather than modifying the algorithms themselves together with their internal data structures. A parallel algorithm for mesh simplification is chosen to show the Spatial Locks usefulness when parallelizing geometric algorithms with ease on multi-core machines. Experimental results illustrate the advantage of using this synchronization mechanism where significant computational improvement can be achieved compared to alternative approaches.

Keywords: Geometric algorithms, spatial hashing, synchronization mechanisms, locks

1 Introduction

Geometric algorithms play an important role in many applications such as geographic information systems, computer aided design, molecular biology, medical imaging, computer graphics, and robotics. They are known for being highly compute-intensive, especially when constructing or updating models in 3D space. The problem becomes harder when models are big and highly detailed, which involves more computational resources needed such as processors and memory to process them. With the advent of computing systems with many CPUs per chip, well-known as multi-core and many-core machines, parallelizing these algorithms has become a desired objective in order to achieve significant computational improvement.

The parallelization of algorithms that update models in 2D or 3D space has been commonly solved by dividing the model into sub-models, processing each sub-model with different threads in parallel to finally merge the individual results, taking special care of the stitches. Even though these algorithms are known to be embarrassingly parallel and show good performance, they bogdown when the update operations must be performed in certain order or with priorities. For example, in the simplification algorithm for triangulated meshes, parallelizing the simplification by decomposing the mesh into submeshes and run the algorithm sequentially on each part can lead to bad quality results [9].

Multi-threaded algorithms that perform a mixed set of operations in 3D space are also appealing. A well-known example is the 3D Delaunay triangulation, which is typically implemented using shared containers for vertices and cells. Building a Delaunay triangulation requires threads to perform efficient alternating addition and removal of new and old cells, and addition of new vertices, updating the shared containers. These operations come out as the algorithm runs based on the current properties of the mesh.

We introduce a new synchronization mechanism called Spatial Locks that allows thread synchronization on shared-memory and multi-core architectures. This synchronization becomes very useful when the algorithm to be parallelized performs updates over objects in 2D or 3D space by following a certain order of processing. Even though the example problem in this paper is the synchronization of concurrent updates done by geometric algorithms on shared objects in 2D and 3D spaces, it can also be applied to other kinds of algorithms that might need this type of synchronization.

The remainder of this article is divided into four sections. Firstly, in the next subsection the related work is reviewed; in section 2 Spatial Locks are formalized with their paradigm and implementation; in section 3 we describe some geometric algorithms that can benefit from the use of Spatial Locks, highlighting the mesh simplification algorithm with a proposed parallelization and its corresponding experimental results. Lastly, in section 4 conclusions are given.

1.1 Related Work

Many synchronization mechanisms for shared-memory systems have been proposed over the years. Locks, Semaphores, Monitors and many other variants are used by most of the multi-threaded applications that require synchronization. Even though they were proposed several years ago, the design of efficient multi-core locks is still a hot research topic [10, 15, 7], considering also that the current trend is to include more CPU cores per chip.

Attempts to parallelize Geometric Algorithms have been focusing on partitioning the space into sub-regions, computing a sub-solution for each sub-region, and finally merging all of them into a final result. Some examples are the algorithms proposed in [6, 5] to solve the 3D Delaunay Triangulation. Another well-known approach is acquiring exclusive access to the containing sub-region and cells around it, as presented in [13] for the parallelization of randomized incremental construction algorithms. Batista et al. presented in [1] a few strategies to parallelize some geometric algorithms such as vertex-locking strategy, cell-locking strategy and other variants. These strategies use additional variables within vertices and through atomic operations they guarantee synchronization. Additionally, priority locks are used to avoid deadlock that occurs when a thread, that might already own other locks, waits for a lock owned by another thread. Our proposal differs from related works by offering a simpler approach that does not require modifications on geometric algorithms' data structures or surface information, in fact it only requires a simple initialization and the set of points –minimum and maximum points to be precise– representing the surface being updated in 2D or 3D space to guarantee mutual exclusion. Moreover, it provides better performance comparing with vertex-locking strategies, especially when the vertex degree of objects is high.

2 Spatial Locks

The basic idea behind the proposed Spatial Locks mechanism is to protect 2D or 3D regions by dividing them into space cells and using lock-protection for each acquired cell.

The novelty in Spatial Locks stems from the merging of two well-known concepts, namely Spatial Hashing and Axis-aligned Bounding Box (AABB). Spatial hashing is the process by which a 3D or 2D domain space is projected into a 1D hash table [11]. The hash function takes any given 2D or 3D positional data and returns a unique grid cell that corresponds to a 1D bucket in the hash table. The hash function for hashing 2D to 1D can be as simple as $hash = x * conversion\_factor + y * conversion\_factor * width$, where $conversion\_factor$ is computed by $1/cell\_size$ and $width$ is the number of uniformly-sized cells per axis. The cell size is defined by the user and will depend on the algorithm domain. It is called spatial hash because the cell index of a data element can be obtained in constant time by its coordinates (e.g., x, y and z) with the hash function. Evidently, a 2D or 3D matrix can be used instead of the hash table and using a simplified hash function will be enough to get the corresponding cell: $grid[x * conversion\_factor,\ y * conversion\_factor]$. The programmer has the option to manage this addresses himself or leave this work to the compiler by using matrices. Figure 1 shows the elementary use of spatial hashing, where geometric points are hashed into cells placed in a 3D grid.

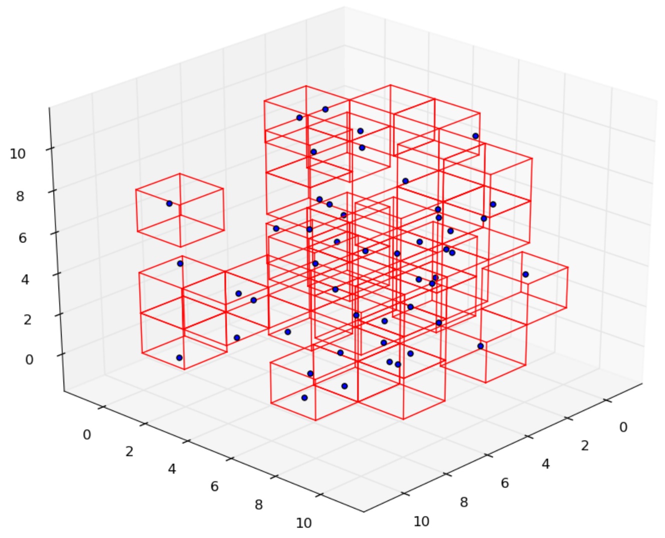

Figure 1: Spatial hashing where objects are mapped into uniformly-sized cells

An AABB is a rectanguloid whose faces are aligned with the coordinate axes of its parent coordinate system. An AABB can be represented or constructed with just minimum and maximum extents along each axis. For our purpose, AABBs will represent the building boxes of geometric shapes, so Spatial Locks can be seen as AABBs placed in a 2D or 3D grid indicating that threads are performing concurrent updates in them. Thus, the logic behind Spatial Locks lies on protecting threads' updates on sections of shapes or objects by placing AABBs into the spatial hash table.

2.1 Paradigm and Interface

Many applications exist with geometric algorithms that update objects in 2D or 3D space. The objective of using Spatial Locks is to allow safe parallelization of these algorithms by guaranteeing mutual exclusion. This synchronization mechanism works by protecting objects being updated by one thread from other thread's updates over the same object. It maintains an internal spatial hash table that reflects the status of concurrent updates by threads, identifying in which cells threads are performing concurrent updates. Synchronization can be achieved from the user perspective by means of the Spatial Locks' functions $lock(object)$ and $unlock(object)$, where *object* corresponds to either a set of points defining a section of a shape or an AABB of that section. The $lock(object)$ and $unlock(object)$ functions work similarly to regular mutexes, i.e. the first function is blocking while the latter is not.

Synchronizing thread's updates does not only mean locking subsections of the grid for exclusive access, yet providing enough functionality to give different possibilities for thread synchronization and achieve the best performance and parallelism. Hence, this primitive also provides $check(object)$ and $tryToLock(object)$ functions which are not blocking. As its name describes, $check(object)$ returns a boolean value indicating whether the cell to which *object* belongs is occupied or not. On the other hand, $tryToLock(object)$ tries to acquire the corresponding cell for *object* but only considering one attempt. A boolean value is returned indicating whether the cell was acquired or not.

Imagine that an algorithm needs to perform updates to several parts of a 3D shape, and these updates can be performed by any thread in any part of the shape, so dividing the shape and assigning each part to each thread is not the best option –updates can be heavily located on a certain section of the shape, overloading a single thread and as a result serializing the entire execution–. Threads can perform their updates safely by using $lock(object)$ and $unlock(object)$ as regular locks. However, better performance can be achieved by implementing a less blocking technique if the updates do not require a specific order of processing. For example, $tryToLock(object)$ can be used instead of $lock(object)$, and if the object's cell is already taken by another thread this update is enqueued to be tried later and a different object update can be performed instead.

2.2 Implementation

The implementation of Spatial Locks in this paper assumes hardware support for atomic read-modify-write operations on a single memory location. The fundamental operation is *compare and swap (CAS)*, specified as *int CAS(addr, old, new)*, which checks in one atomic operation the equality between the value on *addr* and the value on *old*. If equal, it changes the value of *addr* to *new* value and returns a flag when the change is successful.

The $lock(object)$ operation is shown in Algorithm 1. It can be seen that the spatial hash table is updated by using the *compare and swap (CAS)* atomic operation, which guarantees thread safety and allows thread synchronization. The minimum and maximum indices obtained from the AABB's minimum and maximum points serve as guidelines for obtaining the cell indices to be locked by the current thread. These cell indices are calculated by the function $getCell(point)$ that performs the hashing operations explained at the beginning of section 2. If the minimum and maximum cell indices are equal, only one cell has to be taken; else 2 or 4 can be taken in 2D space and 2, 4 or 8 in 3D space. For the latter scenario, thread synchronization and safety is achieved by using an internal mutex, otherwise deadlock-freedom cannot be guaranteed even with CAS operations.

The situation when the minimum and maximum cell indices are not equal is handled as follows: based on AABB's minimum and maximum points, new 2D/3D points are created to potentially lock additional cells. For instance, for an object in 2D space up to 4 cells can be locked when the coordinates (x, y) of its minimum and maximum AABB points are different, and their two additional cell indices are obtained from the points ($max.x$, $min.y$) and ($min.x$, $max.y$). A similar procedure is performed for objects in 3D space, where the additional cell indices are obtained from the points ($min.x$, $min.y$, $max.z$), ($min.x$, $max.y$, $min.z$), ($max.x$, $min.y$, $min.z$), ($min.x$, $max.y$, $max.z$), ($max.x$, $min.y$, $max.z$), and ($max.x$, $max.y$, $min.z$). Observe that some of these new points can fall into the same cell index, and the average number of cells to be locked depends directly on the spatial hash table's cell size.

An alternative approach to handle lock conflict when a thread attempts to take multiple cells is using priorities. So instead of using a mutex as in Algorithm 1, lock conflicts can be handled by using priority locks where each thread is given a unique priority. If the acquiring thread has a higher priority it simply waits for the lock to be released. Otherwise, it retreats, releasing all previous cells and restarting the operation. This approach also avoids deadlocks and guarantees progress.

Algorithm 1: Lock object in Spatial Hash Table

```
Input : object represented as an AABB
minIndex = getCell(aabb.minPoint);
maxIndex = getCell(aabb.maxPoint);
if minIndex == maxIndex then
    while true do
        if spatialHashTable[minIndex] == 0 then
            if CAS(spatialHashTable[minIndex], 0, 1) then
                break;
            end
        end
    end
else
    mutex.lock();
    while true do
        if table[minIndex] == 0 then
            if CAS(spatialHashTable[minIndex], 0, 1) then
                break;
            end
        end
    end
    while true do
        if table[maxIndex] == 0 then
            if CAS(spatialHashTable[maxIndex], 0, 1) then
                break;
            end
        end
    end
    // continue with other indices
    // AABB can be part of 2, 4 cells in 2D
    // and 2, 4 or 8 in 3D
    mutex.unlock();
end
```

Note that before performing the CAS operation in Algorithm 1, we first check if the cell is available (if value is 0). It might seem redundant since the following CAS operation also checks if the cell is 0, nonetheless this presents better performance than using only CAS operations due to cache coherency and the costs of the atomic operations on modern architectures [12].

The $unlock(object)$ operation, shown in Algorithm 2, is very simple and also similar to $lock(object)$, however the internal mutex is not needed. Once the cell indices are obtained from the AABB, these are immediately released by using *CAS* operations. It is guaranteed that the thread that acquires a specific cell, is the same who releases it. It is also guaranteed that the status of every cell that will be released by a thread is *locked* (value 1 of *table* in Algorithm 1 and 2); so no additional verification is needed by the time of releasing cells within this operation.

Lemma 1. *Spatial Locks satisfy mutual exclusion when objects fall into the same unique cell.*

Proof. By contradiction. Suppose that 2 threads, A and B, are going to update the objects X and Y respectively, which fall into the same cell S in 3D space. It means that both X's and Y's minIndex and maxIndex are equal. These 2 threads first call $lock(object)$, so the order of events from Algorithm 1 is:
$read_A(cell_S = false) \rightarrow write_A(cell_S = true) \rightarrow UpdateObject_A(X)$
$read_B(cell_S = false) \rightarrow write_B(cell_S = true) \rightarrow UpdateObject_B(Y)$

Algorithm 2: Unlock object in Spatial Hash Table

```
Input : object represented as an AABB
minIndex = getCell(aabb.minPoint);
maxIndex = getCell(aabb.maxPoint);
if minIndex == maxIndex then
    spatialHashTable[minIndex].store(0);
else
    spatialHashTable[minIndex].store(0);
    spatialHashTable[maxIndex].store(0);
    // continue with other indices
    // AABB can be part of 2, 4 cells in 2D
    // and 2, 4 or 8 in 3D
end
```

Without loss of generality, assume that A was the last thread to update the object X. Since thread A still entered its critical section, it must be true that thread A reads $cell_S == false$. Thus it follows that $read_B(cell_S = false) \rightarrow write_B(cell_S = true) \rightarrow UpdateObject_B(Y)$ $read_A(cell_S = false) \rightarrow write_A(cell_S = true) \rightarrow UpdateObject_A(X)$
This is impossible because there is no other write $false$ to $cell_S$ between $write_B(cell_S = true)$ and $read_A(cell_S = false)$. Contradiction. □

Lemma 2. *Spatial Locks satisfy mutual exclusion when objects fall into multiple cells, having or not common cells between them.*

Proof. Suppose that 2 threads, A and B, are going to update the objects X and Y respectively in 3D space. Object X falls into the cells S_1 and S_2, and object Y falls into S_2 and S_3. These 2 threads first call $lock(object)$, so the order of events from Algorithm 1 is:
$mutex.lock() \rightarrow read_A(cell_{S_1} = false) \rightarrow write_A(cell_{S_1} = true) \rightarrow read_A(cell_{S_2} = false) \rightarrow write_A(cell_{S_2} = true) \rightarrow mutex.unlock() \rightarrow UpdateObject_A(X)$
$mutex.lock() \rightarrow read_B(cell_{S_2} = false) \rightarrow write_B(cell_{S_2} = true) \rightarrow read_B(cell_{S_3} = false) \rightarrow write_B(cell_{S_3} = true) \rightarrow mutex.unlock() \rightarrow UpdateObject_B(Y)$
By simply holding the mutual exclusion property of the shared $mutex$, both sequences of events A and B are sequentialized. □

Theorem 1. *Spatial Locks (Algorithm 1 and 2) satisfy mutual exclusion.*

Proof. Follows from Lemma 1 and 2. □

Theorem 2. *Spatial Locks (Algorithm 1 and 2) satisfy deadlock-freedom*

Proof. By contradiction. Without loss of generality, suppose thread A waits forever in the $lock(object)$ function. Particularly, it runs $while()$ forever waiting $cell_S$ becomes $false$. If thread B also got stuck in its $while()$ loop, then it must have read $true$ from $cell_S$. But since $cell_S$ cannot be $true$ from the beginning, the hypothesis that thread B also got stuck in its $lockObject()$ method is impossible. Thus, thread B must be able to enter its critical section. Then after thread B finishes its critical section and calls $unlock()$ method, $cell_S$ becomes false, and this triggers thread A to enter its own critical section. Hence, thread A must not wait forever at its $while()$ loop. □

The $check(object)$ operation can be implemented by using *load* operations on the atomic cells from the spatial hash table. If all the cells are available, the object is reported as available to be updated. On the other hand, the $tryToLock(object)$ operation works by doing something very similar to Algorithm 1, but instead of waiting until the cell is available, it returns immediately. In the scenario of having multiples cells to be acquired, when there is one that is not available along the way it rolls back by releasing the ones already acquired.

2.3 Settings and restrictions

The use of Spatial Locks requires initial settings according to the set of objects to be updated and the algorithm itself. Let S be the set of objects or sub-shapes that will be updated by the algorithm. In geometric terms every object $s \in S$ corresponds to a set of 2D or 3D points. The following invariants must be held when using Spatial Locks to synchronize concurrent updates over S; where the Γ operator returns the size of an object's AABB:

- The dimension of the spatial hash table T covers all the objects in S:

$$\Gamma(T) \geq \Gamma(\bigcup_{i=1}^{n} S_i)$$

- Let c be the cell size of the spatial hash table. Then,

$$\Gamma(c) > \Gamma(s), \forall s \in S$$

Since the performance of a concurrent algorithm using Spatial Locks depends directly on the chosen value c, the spatial hash table can be re-built with a new c' any time during the execution of the algorithm. However, these invariants must preserve with the remaining objects in S and the new T'.

3 Geometric Algorithms with Spatial Locks

Geometric algorithms that perform a large amount of updates on different parts of a shape can benefit from using Spatial Locks for their multi-threaded implementations. Some examples are algorithms for polygon mesh processing [3], triangulated surface mesh deformation [18], triangulated surface mesh refinement [16], 3D surface subdivision methods [17], and so on. We parallelized the triangulated surface mesh simplification algorithm [4] with the use of Spatial Locks as an illustrative application of this synchronization mechanism. Results are given in the following subsections.

3.1 Mesh Simplification

Surface mesh simplification is the process of reducing the number of faces used in a surface mesh while keeping the overall shape, volume and boundaries preserved as much as possible [4]. It can be considered as the opposite of the subdivision or mesh refinement. Models are usually represented as triangulated meshes and their simplification algorithms can be used to automatically generate them in different resolutions, so that designers only need to model the finest level. The size of models are reduced dramatically, so they are also used in streaming and network applications. Figure 2 shows different levels of simplifications of a hand obtained with this algorithm.

Due to the considerable processing time, simplification algorithms are normally used as a preprocessing step. Considering that the average performance of computing edge-collapse simplification with quadric error metrics, one the most common methods for mesh simplification, is about 50,000 operations per second [9]. Although the parallelization of this algorithm seems to be simple, most of the edge-collapsing simplification algorithms work sequentially due mainly to the large neighborhood information required for the computation of optimally ordering of operations, where triangles are simplified following priorities and costs.

For our study we chose the mesh simplification algorithm presented in [14, 8, 4]. The algorithm proceeds in two stages. In the first stage, called collection stage, an initial collapse cost is assigned to each and every edge in the surface mesh and maintained in a priority queue. In the second stage, called collapsing stage, edges are processed in order of increasing cost. Some processed edges are collapsed while some are just discarded. Collapsed edges are replaced by a vertex and the collapse cost of all the edges now incident on the replacement vertex is recalculated, affecting the order of

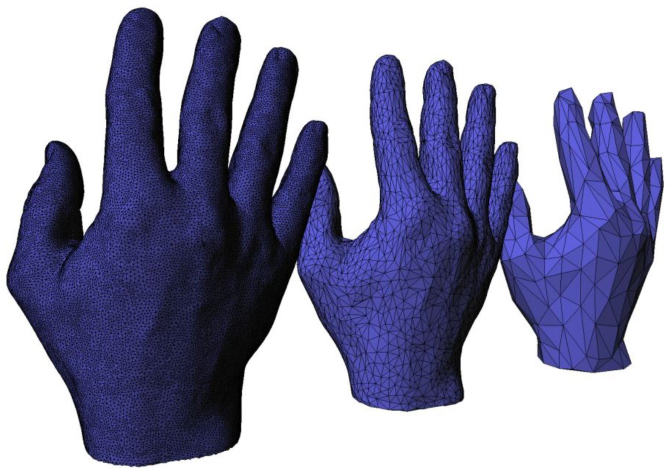

Figure 2: Surface mesh simplification algorithm applied to a mesh with different simplification levels [4].

the remaining unprocessed edges. The process ends when a desired number of triangles is reached or the collapse cost is over a specific threshold. Notice that processing edges by their collapse costs has been proven to provide better quality results than any other ordering.

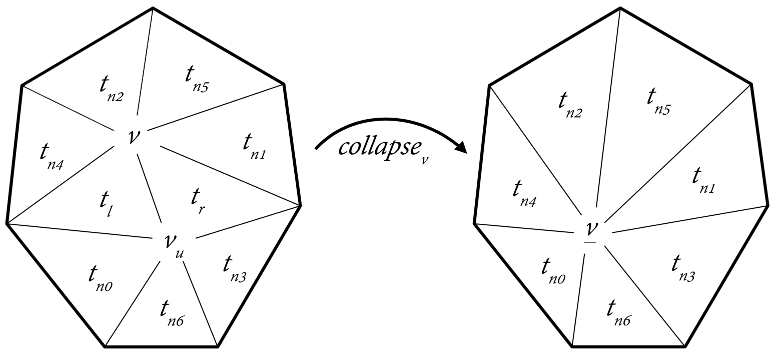

Figure 3: Contraction of the edge $(v,\, v_u)$ into a single vertex. The shaded triangles become degenerate and are removed during the contraction.

Figure 3 illustrates the edge-collapse operation over the edge (v, v_u). This operation takes the edge (v, v_u) and substitutes its two vertices v and v_u with a new vertex $\underline{v}$. In this process, the triangles t_l and t_r are collapsed to edges, and are discarded. The remaining edges and triangles incident upon v and v_u, e.g., t_{n0}, t_{n1}, t_{n2}, t_{n3}, t_{n4}, t_{n5} and t_{n6} respectively, are modified such that all occurrences of v and v_u are substituted with $\underline{v}$. The computation involved on this process can be summarized as: computing the cost of collapsing the edge e, choosing the position of the vertex $\underline{v}$ that replaces the edge, and updating the adjacent triangles.

3.1.1 Parallel Mesh Simplification using Spatial Locks

Parallelizing the Mesh Simplification Algorithm implies synchronizing all the threads' updates on the shared mesh. The specific region to synchronize on every update is given by the edge to be collapsed and its adjacent triangles, as it is shown in Figure 3. Even though the external edges of

Algorithm 3: Parallel Mesh Simplification using Spatial Lock

```
Input   : Set of edges
Output: Reduced set of edges
for every iteration do
    parallel for every edge $e$ do
        err = calculateError($e$);
        if $err > threshold$ then
            continue;
        else
            aabb = createAABB(getTriangle($e$));
            spatialLocks.lock(aabb);
            collapseEdge($e$);
            spatialLocks.unlock(aabb);
        end
    end
    spatialLocks.rebuild();
end
```

adjacent triangles (t_{n0}, t_{n1}, and so on) are not modified during the edge-collapse operation, their vertices are part of several edges being updated, so they cannot be collapsed by another thread at the same time. However, these external edges can be borderline of two parallel edge-collapse operations.

The parallelization of the aforementioned algorithm is explained as follows:

- A concurrent priority queue P is used to store all the edges with their costs as priorities. If priorities are relaxed a concurrent vector can be used instead.
- Every thread takes an edge e from P and analyzes its quadratic errors.
- If an edge is set to be collapsed, the corresponding AABB given by the edge's adjacent triangles is locked in the spatial hash table.
- Once the edge is collapsed and the shaded triangles removed, the corresponding AABB is removed safely from the spatial hash table.

A simplified version of the parallel mesh simplification algorithm (PMS) is shown in Algorithm 3. Note that the use of *parallel for* does not mean the subdivision of range for every thread, but the parallelization for obtaining an edge from the priority queue at every iteration. If two threads are attempting to update adjacent triangles, only one will succeed locking the triangle given by its AABB and the other must wait.

The complete mesh simplification process can be done on several iterations depending on the level of simplification desired by the user. After every iteration, triangles have become bigger, so the *rebuil()* function re-builds the spatial hash table with a greater cell size, as explained in section 2.3. The new cell size is calculated similarly to its initial value at the beginning of the process: it must be greater than every remaining triangle's AABB in the mesh.

Recall that processing edge-collapse operations based on their costs as priorities is very crucial for the quality of the resulting mesh. Therefore, the property is maintained even when there could be threads that might reflect their results later than others due to the operating system scheduler. Under normal conditions, once a thread acquires a Spatial Lock for an edge-collapse operation, it is guaranteed to perform this operation in a finite number of steps.

Two other variants of the parallel mesh simplification algorithm were implemented. In the first variant, called *PMS-RP*, the cost priorities are relaxed and threads attempt to acquire Spatial Locks by using the non-blocking function $tryToLock(object)$. If the corresponding cell is taken by another thread, the next edge is taken from the queue and the same process is tried again. The second variant, called *PMS-MBB*, allows having bigger cells and multiples AABBs per cell. For this purpose an AABB tree [2] (which provides add, remove and intersection operations) with a regular mutex are used in every cell, so the protection of an edge-collapse operation is given by the presence of its AABB in the tree rather than a specific flag within the cell.

3.2 Experimental Results

A set of experiments were carried out in order to evaluate the performance of the *PMS* algorithm using Spatial Locks and to compare it to its sequential version as well as the parallel variants *PMS-RS*, *PMS-MBB* and one using internal locks in vertices. Algorithms were implemented in the C++11 programming language. The experiments were carried out on a Dual 14 Core Intel(R) Xeon(R) CPU E5-2695 v3, with a total of 28 physical cores running at 2.30GHz. Hyperthreading was disabled. The computer runs Linux 3.19.0-26-generic, in 64-bit mode. This machine has per-core L1 and L2 caches of sizes 32KB and 256KB, respectively and a per-processor shared L3 cache of 35MB, with a 32GB DDR RAM memory and 1TB SSD. Algorithms were compared in terms of running times using the usual high-resolution (nanosecond) C functions in *time.h*.

Table 1: Performance of the PMS algorithm implemented with Spatial Locks

Model	# triangles	# removed triangles	1 thr (sec.)	2 thr (sec.)	4 thr (sec.)	6 thr (sec.)	8 thr (sec.)
Bunny	69,664	34,832	0.3	0.26	0.23	0.20	0.20
Head	281,724	140,862	1.4	1.3	1.1	1.0	0.9
Wall	651,923	325,961	3.3	2.8	2.5	2.2	2.0
Einstein	674,038	337,018	4.1	3.7	3.5	3.0	2.7
Motors	1,516,759	758,360	13.5	9.1	5.5	4.4	3.8
Facew	2,402,732	1,001,364	22.1	18.4	14.2	7.2	5.9
Castle	3,136,234	1,218,116	32.1	25.8	17.2	9.1	7.4

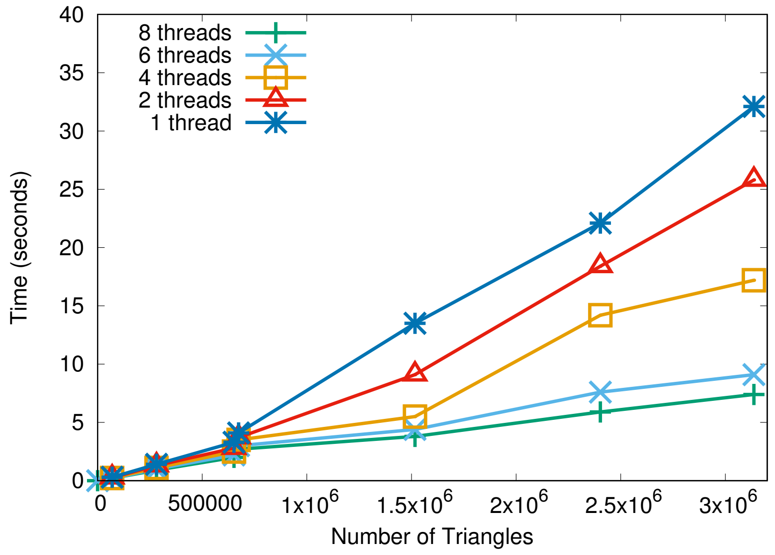

Figure 4: Total time spent by the parallel mesh simplification algorithm in simplifying surface meshes with up to 3,136,234 triangles.

Table 1 shows detailed information of mesh simplifications using the original algorithm (sequential with one single thread) and PMS using Spatial Locks with up to 8 threads. Meshes with more than

1 million of triangles are particularly of our interest, since their simplifications require a big amount of edge-collapse operations. These meshes, which have from 60K to 3.2M triangles, were simplified using a 0.5 simplification ratio with both algorithms. All the surface meshes were obtained from [19].

It can be seen that as the number of triangles in the surface mesh increases, the runtime for all variants also increases. However, the increment of PMS' runtime with several threads is slower than its sequential counterpart, getting much better performance with bigger meshes. It is due to the fact that with bigger meshes threads have less probabilities to interfere each other within the same spatial hash table's cell, getting parallelism benefits. Figure 4 shows the runtime trend of all the variants when the number of triangles increases. It turned out that for some experiments, specifically those with small meshes (under 100K triangles), the sequential algorithm showed better or very similar performance than our parallel implementation. This observation can be used as a key factor when deciding whether to implement a parallel geometric algorithms with Spatial Locks or avoid its use.

We also counted the number of cells that *lock*(*object*) operations occupied with different configurations for the spatial hash table. As it can be anticipated, with greater cell size the number of occupied cells by one *lock* operation is smaller. i.e. with a cell size of 4 times the average of edge size, the *lock* operations that occupied only 1 cell was over 89%, and the percentage of *lock* operations using 8 cells was only 2%. The trade-off is given by adjusting the cell size to have less *lock* operations falling on 8 cells (worst case), but as the cell size is increased there is less parallelism due to the fact of having cells covering more space and threads spending more time in the locking phase. For this specific algorithm, we obtained the best performance with a cell size of 3 times the average of the edge size, where the *lock* operations occupying 1 cell were around 86% of the total, and those falling on 8 cells represented only 2% of the total.

To make the comparison fair, we implemented a parallel mesh simplification algorithm with internal locks using a similar approach to the one proposed in [1] (called fine-grained locks). To guarantee thread safety when collapsing edges, the internal locks are managed in 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.

Table 2: Performance of mesh simplification algorithms using internal locks and Spatial Locks

Model	# triangles	# removed triangles	PMS Internal Locks (sec.)	PMS Spatial Locks (sec.)
Bunny	69,664	34,832	0.3	0.2
Head	281,724	140,862	1.4	0.9
Wall	651,923	325,961	3.3	2.0
Einstein	674,038	337,018	4.1	2.7
Motors Car	1,516,759	758,360	11.5	3.8
Facew	2,402,732	1,001,364	26.7	5.9
Castle	3,136,234	1,218,116	20.1	7.4

Similarly to previous experiments, we carried out several simplifications on different models. Table 2 and Figure 5 summarizes and plots respectively the running times of the mesh simplification algorithms using internal locks and Spatial Locks. Similar to previous setup, both algorithms were tested with 8 threads. As it can be seen, the alternative with Spatial Locks performs all the simplifications in less time, being approximately 3 times faster than the alternative with internal locks. Additionally, it is not only a better alternative in terms of performance, but its use is also much less invasive since there is no need to modify internal data structures as we did with internal locks approach.

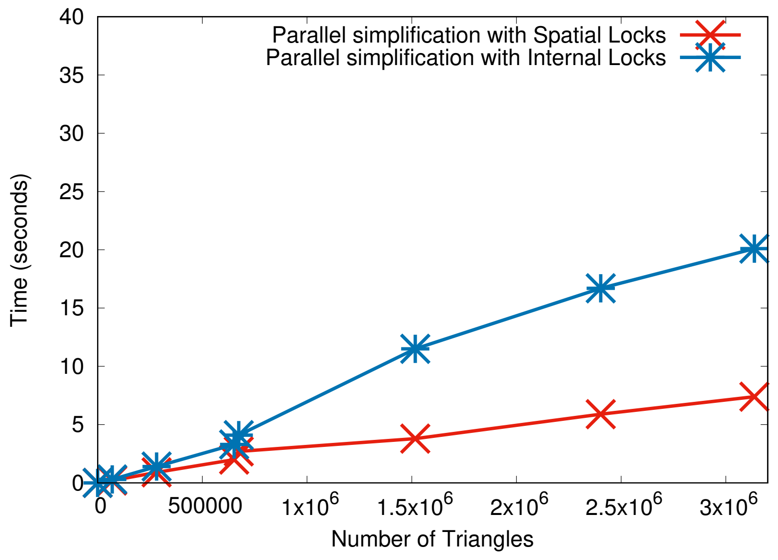

Figure 5: Total time spent by the mesh simplification algorithm using internal locks and Spatial Locks.

There are several other synchronization alternatives that can be created from the concept of Spatial Hashing. We also measured a couple of them explained in the previous section. Figure 5 illustrates the throughput (number of removed triangles per second) by PMS and its two variants: *PMS-RP* and *PMS-MBB*. In this scenario PMS-RP obtains the highest throughput since waits for available cells are avoided by relaxing priorities when choosing edges to collapse. Recall that this variant might lead to worse quality results for their resulting meshes. On the other hand PMS-MBB, that allows more than one AABB per cell, presents lower throughput than PMS. It can be explained by the use of local mutexes as well as the complex operations that imply using the AABB-tree with insertions and removals.

4 Conclusions

Spatial Locks constitute a useful synchronization mechanism that allows to make parallel geometric algorithms thread-safe. Based on Spatial Hashing and Axis-aligned Bounding Boxes (AABB), they provide constant-time lock/unlock operations when updating an object in 2D or 3D space. It has been proven that this synchronization mechanism satisfies *mutual exclusion* and *deadlock-freedom*, fundamental properties for any thread synchronization mechanism. Many geometric algorithms can benefit from Spatial Locks given that its use does not require significant changes on the implementation of the geometric algorithms themselves. Our experiments show that highly parallel executions with important speed-up can be obtained when using this synchronization mechanism for mesh simplification processes and big meshes.

References

[1] Vicente HF Batista, David L Millman, Sylvain Pion, and Johannes Singler. Parallel geometric algorithms for multi-core computers. *Computational Geometry*, 43(8):663–677, 2010.

[2] Gino van den Bergen. Efficient collision detection of complex deformable models using aabb trees. *Journal of Graphics Tools*, 2(4):1–13, 1997.

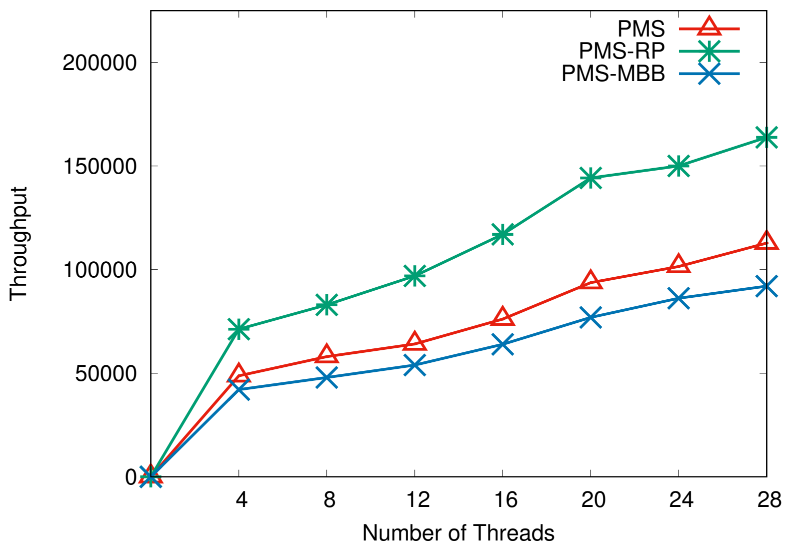

Figure 6: Total time spent by both algorithms in simplifying a surface mesh with 2,436,234 triangles.

[3] Mario Botsch, Leif Kobbelt, Mark Pauly, Pierre Alliez, and Bruno Lévy. *Polygon mesh processing*. CRC press, 2010.

[4] Fernando Cacciola. Triangulated surface mesh simplification. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.10 edition, 2017.

[5] Paolo Cignoni, Domenico Laforenza, Raffaele Perego, Roberto Scopigno, and Claudio Montani. Evaluation of parallelization strategies for an incremental delaunay triangulator in e3. *Concurrency and Computation: Practice and Experience*, 7(1):61–80, 1995.

[6] Paolo Cignoni, Claudio Montani, Raffaele Perego, and Roberto Scopigno. Parallel 3d delaunay triangulation. In *Computer Graphics Forum*, volume 12, pages 129–142. Wiley Online Library, 1993.

[7] David Dice, Virendra J Marathe, and Nir Shavit. Lock cohorting: a general technique for designing numa locks. In *ACM SIGPLAN Notices*, volume 47, pages 247–256. ACM, 2012.

[8] Michael Garland and Paul S Heckbert. Surface simplification using quadric error metrics. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 209–216. ACM Press/Addison-Wesley Publishing Co., 1997.

[9] Nico Grund, Evgenij Derzapf, and Michael Guthe. Instant level-of-detail. In *VMV*, pages 293–299, 2011.

[10] Hugo Guiroux, Renaud Lachaize, and Vivien Quema. Multicore locks: The case is not closed yet. *USENIX Annual Technical Conference*, pages 649–662, 2016.

[11] Erin J Hastings, Jaruwan Mesit, and Ratan K Guha. Optimization of large-scale, real-time simulations by spatial hashing. In *Proc. 2005 Summer Computer Simulation Conference*, volume 37, pages 9–17, 2005.

[12] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2011.

[13] 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.

[14] Peter Lindstrom and Greg Turk. Fast and memory efficient polygonal simplification. In *Proceedings of the Conference on Visualization '98*, VIS '98, pages 279–286, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[15] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. Fast and portable locking for multicore architectures. *ACM Transactions on Computer Systems (TOCS)*, 33(4):13, 2016.

[16] Pedro Rodriguez, Maria Cecilia Rivara, and Isaac D. Scherson. Exploiting the memory hierarchy of multicore systems for parallel triangulation refinement. *Parallel Processing Letters*, 22(03):1250007, 2012.

[17] Le-Jeng Andy Shiue. 3D surface subdivision methods. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.10 edition, 2017.

[18] Olga Sorkine and Marc Alexa. As-rigid-as-possible surface modeling. In *Symposium on Geometry processing*, volume 4, 2007.

[19] Qingnan Zhou and Alec Jacobson. Thingi10k: A dataset of 10, 000 3d-printing models. *CoRR*, abs/1605.04797, 2016.