Using Transactions in Delaunay Mesh Generation

Milind Kulkarni
Cornell University
milind@cs.cornell.edu

L. Paul Chew
Cornell University
chew@cs.cornell.edu

Keshav Pingali
Cornell University
pingali@cs.cornell.edu

Abstract

Mesh generation is a key step in graphics rendering and in using the finite-element method to solve partial differential equations. The goal of mesh generation is to discretize the domain of interest using polygonal elements such as triangles (in 2-D) or tetrahedra (in 3-D).

One popular mesh generation algorithm is Delaunay mesh generation, which produces meshes with certain quality guarantees that are important for problems in which the geometry of the problem changes with time. Delaunay mesh generation works by iterative refinement of a coarse initial mesh. The sequential algorithm repeatedly looks for a "bad" mesh element that does not satisfy the quality constraints, computes a neighborhood of that element called its *cavity*, and replaces the elements in that cavity with new elements, some of which may not satisfy the quality guarantees themselves. It can be shown that the algorithm always terminates and produces a guaranteed quality mesh, regardless of the order in which bad elements are processed.

Delaunay mesh generation can be parallelized in a natural way because elements that are far away in the mesh do not interfere with each other as they are being processed. We present experimental results showing that in practice, there is indeed a lot of parallelism that is exposed in this way. However, exploiting this parallelism in practice can be complicated. Compile-time analysis and parallelization are infeasible because of the input-dependent nature of the algorithm. One alternative is optimistic parallelization. We show how logical transactions can be identified in a natural way in this code, argue that current transactional memory implementations are inadequate for this application, and suggest an alternative conception of transactional memory that addresses these problems.

1. Introduction

Most partial differential equations cannot be solved exactly, so it is necessary to use numerical techniques such as the finite-element and finite-difference methods to solve them approximately. The finite-element method transforms the calculus problem of solving the partial differential equation into the algebraic problem of solving systems of linear equations. A key step in this process is *mesh generation*.

In general, mesh generation refers to the problem of discretizing a continuous domain by placing points in a plane (in the two-dimensional case) or in a space (in higher dimensions), and forming a mesh over those points. Mesh generation algorithms are also useful in graphics, where they are used to tessellate curved surfaces so they may be represented as polygons, which can be rendered more easily. In this paper, we consider the problem of mesh generation for two-dimensional domains, although most of the ideas discussed here generalize to three dimensions.

In graphics as well as in finite element analysis, *mesh quality* is an important consideration. The problem may require that the mesh meet certain quality guarantees, so not every tessellation of the domain is adequate. These guarantees may include bounds on the size of the largest angle in any triangle or on the size of the largest triangle.

One technique for producing guaranteed quality meshes is *Delaunay mesh generation*. Delaunay mesh generation replaces elements that do not satisfy the constraints with elements that do, producing new, refined meshes by inserting new points into the mesh. The basic algorithm was described by Chew [4] and extended by Ruppert [13]. They provide the appropriate refinement procedure, as well as mathematical guarantees regarding the quality of the mesh upon the termination of the procedure. Shewchuck's Triangle program [15] is an efficient implementation of the Delaunay mesh generation algorithm.

In this paper, we argue that the correct approach to parallelizing the Delaunay mesh generation refinement is to use optimistic parallelization. We also discuss the pros

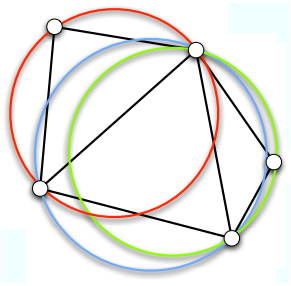

Figure 1. This mesh obeys the Delaunay property. Note that the circumcircle for each of the triangles does not contain other points in the mesh.

and cons of using transactional memory for implementing optimistic parallelization for this application.

The remainder of this paper is structured as follows. Section 2 describes the Delaunay mesh generation algorithm and provides pseudocode listings for the major parts of the sequential algorithm. Section 3 discusses the parallelization opportunities inherent in the algorithm and shows how optimistic parallelization is essential for this application. In Section 4, we discuss how existing transactional memory techniques can be applied to this algorithm, and discuss their shortcomings, together with possible solutions, in Section 5. We describe ongoing work in Section 6.

2. Delaunay Mesh Generation

A *Delaunay mesh* is a mesh over a set of points which satisfies the *Delaunay property* [4]. This property, also called the *empty circle property*, states that the circumcircle of any element (triangle) in the mesh (i.e. the circle which circumscribes the three vertices of the triangle) should not contain any other point in the mesh. An example of such a mesh is shown in Figure 1. In the absence of four co-circular points, a given set of points on a surface has only one triangulation that satisfies the Delaunay property.

In practice, the Delaunay property alone is not sufficient, and it is necessary to impose various quality constraints governing element shape and size. To meet these constraints, Delaunay mesh generation algorithms use an *iterative refinement procedure* that fixes elements that do not satisfy quality constraints.

This refinement procedure is best understood as a *worklist algorithm*. In a worklist algorithm, units of work are placed on a list. When necessary, a unit is removed from the list and processed. Any additional work units produced during this step are placed back onto the worklist. In the case of Delaunay mesh generation, the units of work are elements that do not meet the quality constraints ("bad" elements or triangles). The refinement procedure terminates when the worklist is empty.

```
1: Mesh m = /* read in mesh */
2: WorkQueue wq;
3: wq.enqueue(mesh.badTriangles());
4: while (!wq.empty()) {
5:    Element e = wq.dequeue();
6:    if (e no longer in mesh) continue;
7:    Cavity c = new Cavity(e);
8:    c.expand();
9:    c.retriangulate();
10:   mesh.update(c);
11:   wq.enqueue(c.badTriangles());
12:}
```

Figure 2. Pseudocode of the mesh generation algorithm

Figure 2 shows the pseudocode for the mesh generation procedure. The key steps of this procedure are as follows.

1. Find all the bad elements in the mesh and place them into a *workqueue*, which is an implementation of the worklist [line 3]. Then repeat the following steps until the queue of bad triangles is empty [line 4].
2. Pick an element from the queue [line 5], such as the shaded element in Figure 3(a). The processing of other bad elements may have removed this element from the mesh. If so, there is no work to be done [line 6].
3. Find the circumcenter of the element. This is the new point that will be added to the mesh [line 7]. In Figure 3(a), this is the black point.
4. With respect to this new point, several existing elements will no longer satisfy the Delaunay property (i.e. the new point lies within their circumcircles). Determine the set of elements that are affected by the new point. The set of elements is called a *cavity*, and the process of finding these elements is called *cavity expansion* [line 8]. In Figure 3(b), the shaded grey elements represent the cavity.
5. Calculate a new set of elements which fills the cavity while incorporating the new point. This is the *retriangulation* step [line 9].
6. Replace the cavity with the new elements (i.e. remove the old elements from the mesh, and add in the newly calculated elements) [line 10]. See Figure 3(c).
7. Because the newly created elements are not guaranteed to meet the quality constraints, any new elements

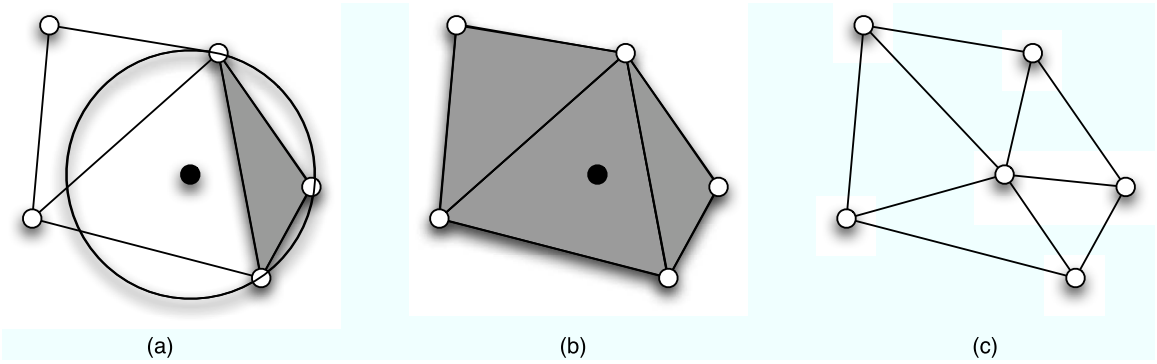

Figure 3. Example of processing a bad element. In (a), we see a "bad triangle" (shaded grey), as well as the new point that we would like to add to the mesh (placed at the center of the triangle's circumcircle). In (b), the grey triangles represent the cavity (elements whose circumcircles contain the new point). In (c), we see the new elements, which fill the cavity but contain the new point. The new mesh still satisfies the Delaunay property.

that are "bad" must be added to the workqueue [line 11].

From the last step, it might appear that the algorithm could potentially not terminate. However, it is guaranteed that no elements can be created whose sides are smaller than the initial segments in the mesh. Since newly created elements are smaller than the original elements, the refinement process must terminate [4]. Note that this holds regardless of the order in which bad elements are processed.

We now describe how the key steps of the refinement procedure (*expansion*, *retriangulation* and *update*) work. A graphical representation of these steps for a single element is shown in Figure 3.

Expansion: In this step, we determine both the cavity (the elements affected by the new point) as well as the border elements of the cavity (the elements immediately surrounding the cavity). Because a cavity is a connected region of the mesh, this step can be accomplished with a simple breadth-first search, starting from the initial bad element, and adding affected elements to the cavity. The border of the cavity is defined by the elements we encounter that are not affected by the new point (which also serves as the termination criterion for that direction of search).

Retriangulation: At this stage, we determine the new set of elements that will replace the affected elements. The new elements should fill the same space as the original cavity, but include the new point. This is accomplished by determining the "boundary" of the cavity (i.e. the outer edges of the cavity), and creating new elements whose vertices are the new point and the vertices of a segment on the border.

Update: Because both the cavity and the new elements share the same boundary, and hence have the same border elements, replacing the cavity with the new elements is straightforward.

Encroachment: There is one special case, called *encroachment*. If the cavity contains a boundary segment of the overall mesh (i.e. the tightest circumcircle of the segment contains the new point), we say that the cavity *encroaches* upon the boundary. We first build a cavity around the boundary segment by placing a new point at its midpoint, and perform the complete refinement process. After this is done, we return to the original bad element and process it again.

3. Parallelization

Two facts are noteworthy about the mesh generation algorithm described in Section 2.

- Each cavity is a connected region of the mesh, and is small compared to the overall mesh.
- There is no specific order in which bad elements need to be processed.

3.1 Parallel Mesh Generation

The first fact leads naturally to a parallel algorithm. Because the cavities are localized, we note that two bad elements that are far enough apart on the mesh will have cavities that do not interfere with one another. Furthermore, the entire refinement process (expansion, retriangulation and graph updating) for each element is completely independent. Thus, the two elements can be processed in parallel, an approach which obviously extends to more than two elements (see Figure 4).

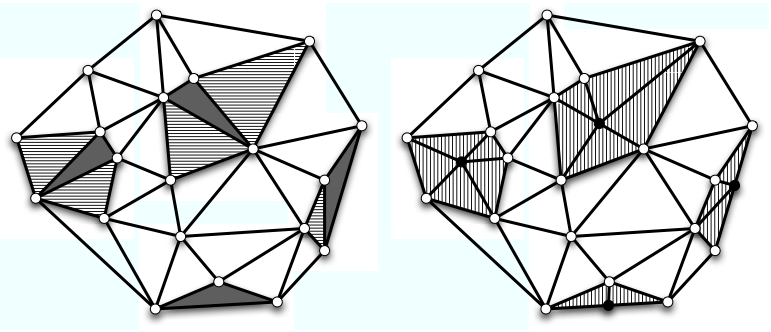

Figure 4. An example of processing several elements in parallel. The left mesh is the original mesh, while the right mesh represents the refinement. In the left mesh, the *dark grey* triangles represent the "bad" elements, while the *horizontally shaded* are the other elements in the cavity. In the right mesh, the the *black* points are the newly added points and *vertically shaded* triangles are the newly created elements.

3.2 Compile-time Parallelization

Compile-time parallelization approaches perform dependence analysis to determine a partial order of program operations, and schedule operations for parallel execution if there are no dependences between them. We do not know of any compile-time parallelization technique that will succeed in finding parallelism in this problem. Since each iteration of the *while* loop of Figure 2 reads and writes the mesh data structure, any compile-time analysis technique that treats the entire mesh as a monolithic unit will assert that there are dependences from every iteration to all succeeding iterations. Since the mesh is read at the beginning of each iteration and updated at the end of that iteration, there is little useful overlap of computations between iterations. A more sophisticated, fine-grained analysis might try to use techniques like shape analysis [8, 14] to discern if the reads and writes to the mesh data structure in different iterations are disjoint. However, such an analysis requires determining whether the cavities of two bad triangles are disjoint, but this depends on the mesh, and thus is not a question that can be determined at compile time. Note also that any compile-time parallelization of the code in Figure 2 will still process bad triangles in the same order as the sequential code would, which is unnecessarily restrictive.

3.3 Optimistic Parallelization

Since static parallelization will not work, we turn instead to optimistic parallelization. At this stage, we leverage the second insight regarding the sequential algorithm: the elements can be processed in any order. Therefore, we do not need to adhere to a specific schedule of processing bad triangles, but can instead expand cavities whenever we can ensure that they can run in parallel with other concurrent expansions. To implement this sort of parallelization, we can perform dynamic checks to detect interference during cavity expansion. For example, we can lock mesh elements during cavity expansion; if some element needed for a cavity expansion is already locked by another cavity expansion, there is interference and one of the cavity expansions must be rolled back. If no interference is detected, we make the appropriate changes to the mesh. In this way, we are able to exploit the inherent parallelism in the mesh generation algorithm even without knowledge of which elements can be processed in parallel, but at the risk of doing useless work in computations that get rolled back.

3.4 Experimental Results

Optimistic parallelization is useful only if the risk of rollbacks is small. *A priori*, it is unclear whether or not optimistic parallelization is useful for Delaunay mesh generation. In addition, the amount of parallelism is very data-dependent and depends on the size of the mesh, the number of bad triangles, etc. The probability of conflict between two concurrent cavity expansions depends not only on these factors but on the scheduling policy for parallel activities.

Antonopoulos *et al.* [2] have investigated how many cavities could be expanded in parallel in a mesh of one million triangles (see Figure 5). They focused on coarse-grain parallelization on a distributed-memory computer, which required mesh partitioning and distribution. They found that across the entire problem, there were more than 256 cavities that could be expanded in parallel until almost the end of execution, and, halfway through execution, there were between 350 and 800 thousand cavities that could be expanded in parallel. These results

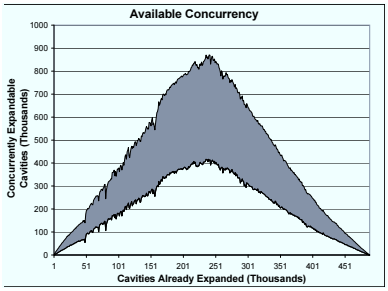

Figure 5. Feasibility study from [2], examining the number of concurrently expandable cavities during the execution of the Delaunay mesh generation algorithm. The x-axis represents time, while the y-axis shows the number of expandable cavities. The upper and lower borders of the shaded area represent upper and lower estimates for expandable cavities.

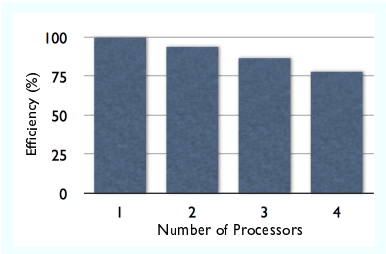

Figure 6. Parallel efficiency results for a lock-based implementation of parallel Delaunay mesh generation.

show that there is adequate potential for parallelism in this problem for mesh sizes of practical interest.

The study of Antonopoulos *et al.* provides upper and lower bounds for potential parallelism because the probability of conflicts between cavity expansions depends on the policy for enqueuing and dequeuing bad triangles on the worklist of Figure 2; some orderings will lead to more parallelism than others. Furthermore, the practically achievable parallelism is constrained by the issues involved in updating the shared structures (i.e. the worklist and the mesh) concurrently. Thus, although the Antonopoulos result shows that the problem exhibits significant parallelism theoretically, there are many factors which may affect the realistic parallelism available in the problem.

To understand these issues, we built a prototype shared-memory implementation. Coordination among concurrent activities was provided by locking. As the cavity was expanded, the triangles needed by the cavity were successively locked. If a triangle needed by a cavity was already locked by some other cavity, a conflict was recorded and cavity expansion was aborted. A two-phase locking

```
1: Mesh m;
2: WorkQueue wq;
3: void process() {
4:    while (!wq.empty()) {
5:       Element e = wq.dequeue(); //atomic
6:         if (e no longer in mesh) continue;
7:       startTransaction();
8:          Cavity c = new Cavity(e);
9:          c.expand();
10:         c.retriangulate();
11:         mesh.update(c);
12:      endTransaction();
13:      wq.enqueue(c.badTriangles()); //atomic
14:   }
15:}
16:void main() {
17:   m = /* read in Mesh */
18:   wq.enqueue(m.badElements());
19:   for (/* size of threadpool */) {
20:      spawn_thread(process);
21:   }
22:}
```

Figure 7. Pseudocode for parallel mesh generation

scheme was used to avoid cascading rollbacks. We found that even this prototype implementation gives fairly good results, achieving a parallel efficiency (parallel speedup divided by number of processors) of approximately 75% on four processors as shown in Figure 6.

These results suggest that optimistic parallelization of Delaunay mesh generation should work well in practice.

4. Transactional Memory

As is well-known, the use of locks to implement parallel algorithms can be cumbersome since the programmer has to focus on proper lock placement, ensure appropriate roll-backs in case of conflicts, etc. In contrast, transactional memory [1, 5, 6, 7, 12] promises a simple solution to parallelization, relieving the programmer of many of these burdens. In this section, we show how the sequential algorithm can be transformed to run in parallel, using transactions to provide proper synchronization between concurrent activities. Pseudocode for this algorithm is shown in Figure 7.

Most studies of transactional memory are concerned with converting parallel code with locks into parallel code that uses transactions. Our research project is concerned more with identifying opportunities for optimistic parallelization in sequential code. The use of the transactional model only provides us with a synchronization mecha-

Parameter	Average value
Instructions	730K
Stores	60K
Loads	88K
L1 accesses	80861
L1 misses	7523

Table 1. Performance characteristics of a single transaction in Delaunay mesh generation

nism but does not specify how the program itself should be parallelized.

In our approach, there is a thread pool in which there are several threads, each of which draws work from the workqueue of Figure 2. Because we want to ensure that concurrent cavity expansions do not interfere, we see that each iteration of the loop in Figure 2 naturally maps to a single transaction. The interference detection provided by transactional memories can detect when two cavities overlap, and hence serve as the trigger for rolling back expansion. The buffering and rollback mechanisms inherent in transactional memory implementations allow for rollbacks without any additional programmer input.

There are a few issues that arise during this type of parallelization. First, we must perform all modifications to the workqueue (both choosing which elements to process as well as enqueueing newly created bad elements) outside the transactions. This is because modifications to shared structures made by a transaction are not visible to other transactions until that transaction commits. Thus, to avoid multiple transactions choosing the same element, we move these operations outside the transaction, and synchronize them using standard mechanisms such as monitors or locks. The second point is that all reads and writes that touch shared structures (which are any that read/create elements or update the mesh) must use transactional reads and writes. This may require rewriting the data structures in terms of transactional operations.

An important note is that, unlike many of the benchmarks that transactional memories are applied to, the transactions in this problem are long running and access relatively large amounts of memory. Table 1 provides some data about the average performance characteristics of a single transaction in this problem when executed on an Itanium 2.

We see that the transaction executes for far longer than most microbenchmarks. Also, the number of memory operations is significantly higher than the number of operations involved in the microbenchmarks of, *e.g.*, [5]. Note also that although the number of L1 cache misses is not necessarily representative of working set size, it indicates that the actual working set is likely to be much larger than the 250 lines of L1 cache on our target system, meaning that hardware transactional memories may overflow their transactional caches.

These characteristics make simple hardware transactional memories [7] unsuitable for our purposes. However, software transactional memories, such as [5, 6], or more advanced hardware approaches [1, 12] may suffice, although their efficiency in the context of long-running transactions must still be studied.

5. Transactional Memory Limitations

While transactions are an intuitive approach to the optimistic parallelization demands of the mesh generation algorithm, current implementations of transactional memory (both software and hardware) exhibit a few problems that may adversely affect the performance of the parallel Delaunay mesh generation algorithm. These two issues are *scheduling of conflicting transactions* and *conservative interference detection*.

5.1 Scheduling of Conflicting Transactions

Consider two concurrent cavity expansions whose cavities overlap. If these transactions attempt to commit simultaneously, it is possible for both to be rolled back [5]. This is because most realistic implementations of transactional memory do not commit the modified state of a transaction in a single step, so it is possible that both transactions attempt to update shared state in inconsistent ways. If both transactions are immediately re-executed, they are likely to conflict again. This type of livelock is a common issue with transactional memories. The traditional solution to this problem is some form of random back-off, to allow some forward progress. From an efficiency standpoint, this is not ideal; a better solution would be to abort one of the transactions and to use the now-idle resources to execute a different transaction. This solution avoids livelock while not incurring the overhead of back-off.

However, as we noted above, the isolation of concurrently executing transactions means we must select the elements to process outside of the transactions (otherwise we risk two transactions choosing to work on the same element). Unfortunately, this means that the default approach is to continue attempting to process a single element until the transaction successfully commits, precluding our use of a more efficient scheduling policy as described above.

One solution is to use a transactional memory system which provides user-specified abort handlers, such as LogTM [10]. Thus, on abort, rather than simply rolling back the transaction, one could re-enqueue the current element and dequeue a new element before restarting the transaction, thus ensuring that the next execution of the transaction will operate on a different element than the previous execution.

Open Nesting

A more general solution may be provided by the *open nesting* approach as discussed in [11]. Nested transactions that are performed in an "open" manner within a parent transaction are allowed to modify shared memory without waiting for the enclosing transaction to commit. Thus, it is possible for a transaction to make changes that are immediately visible to other transactions, rather than waiting until commit.

This would allow the operations on the worklist to appear within the transaction. We can execute the worklist dequeue to obtain an element to process as an open transaction. Because the open nested transaction modifies the worklist before the outer transaction commits, concurrent transactions will see the results of previous dequeues and hence choose distinct elements to work on.

Because shared memory is modified directly, it is necessary to provide "undo" actions to reverse the effects of an open nested transaction in the case of an abort. In this case, a worklist dequeue is undone with an enqueue. On abort, a transaction will re-enqueue the element it is working on, providing the desired rollback behavior as described above.

While this seems like a natural solution, there are many constraints that must be placed on open nested transactions for them to behave properly within the transactional model. Care must be taken that modifications can always be undone and that modifications will not affect the isolation and atomicity of the overall transaction. Current implementations of open nesting, such as ATOMOS [3], do not provide the necessary guarantees to ensure that open nesting is performed safely, choosing instead to relax the isolation guarantees provided by transactions. Thus, the burden falls upon the programmer to utilize open nesting correctly.

We feel that as currently realized, open nesting is overly complex; using it correctly requires the programmer to reason about concurrent effects and potential race conditions. Further investigation is warranted to design an open nesting system that provides the functionality we desire while maintaining the programmability of normal transactions.

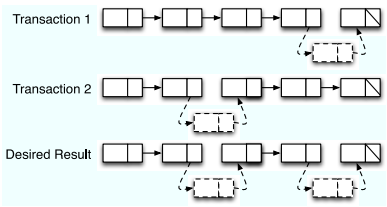

Figure 8. Concurrent insertions into a list

5.2 Conservative Interference Detection

The second drawback of current transactional memories is that they are too conservative; they may detect interference and trigger unnecessary rollbacks. This is an odd charge to level against transactional memory, since one of the motivations for the transactional model is the conservative synchronization enforced by many locking schemes, an issue that transactional memories attempt to avoid.

Consider the example of insertion into a sorted linked list, as seen in Figure 8. Here we see two transactions attempting to insert new nodes into the list at different points. Ideally, these two insertions can proceed in parallel, as they will not interfere with one another. However, the first transaction, in traversing the list to reach the insertion point, has read locations that the second transaction will attempt to modify. A standard transactional memory will detect interference between these two transactions and eventually determine that the only safe course of action is serialization, despite the "high-level" independence of the two transactions.

At a high level, the problem is that there are certain invariants on the abstract data type that must be preserved for correct execution of the program. Ensuring that there are no conflicting reads and writes to the concrete data structures that implement the abstract data type is sufficient but not necessary to ensure that the high level invariants are respected. In the transactional code shown in Figure 7, two transactions interfere if and only if their cavities overlap. However, the transactional memory merely enforces that they do not perform incompatible reads and writes to memory locations of the concrete data structure, which might result in false positives which detect spurious interference.

Let us consider how these false positives could arise in the mesh generation problem. At the abstract level, we can treat the mesh as a graph. Each node of the graph represents an element in the mesh, and adjacency in the

graph models adjacency in the mesh (so each element has at most three neighbors). In this high level view of the structure, we see that we want to detect interference (and hence trigger a roll back) only if two transactions access the same nodes in the graph (*e.g.*, one reads an element which is then removed from the graph by another transaction).

However, transactional memory does not detect interference at this high level. Rather, it focuses on conflicts that occur when accessing specific memory locations – a fundamentally low-level view of interference. Because transactional memory operates at a low level, the amount of interference detected is highly dependent on the concrete implementation of the mesh data structure. However, regardless of the concrete implementation, it is not possible to avoid false positives.

Adjacency list: The simplest implementation of a graph is as an adjacency list. The adjacency list maintains a map from each node in the graph to a list of its neighbors. If this structure is implemented using a standard library such as STL, it will be based around balanced-trees (the default STL map). Thus, any additions or removals of nodes from the graph will result in large portions of the data structures being accessed and modified (due to procedures such as tree rebalancing). Furthermore, because the edges are stored in an adjacency list, steps such as *expand* in the algorithm will require reading large portions of this data structure (to find neighbors). The upshot is that even if two transactions do not access the same elements of the mesh, they will access large parts of the mesh data structure, and are likely to cause low level interference, leading to a high number of false positives.

Local edge information: One technique that could be used to prevent some low level interference is moving the adjacency information from a global adjacency list down to the nodes themselves. Thus, each node maintains a list of its neighbors, but this information is not globally visible. Instead, the only globally visible portion of the graph is a "membership set" which keeps track of which nodes are in the graph. Thus, operations that require finding neighbors (such as *expand*) no longer require reading from a global structure. Although this implementation still suffers from false positives, they are less likely for two reasons. One is that two transactions that access different elements will not interfere in an adjacency list (since there isn't one), instead only potentially interfering in the membership set. The second is that elements are only added and removed from the mesh at the end of a transaction, meaning that the potential interference exposed by modifying the data structure is only "active" for a short period of time.

Hash set: A final optimization that may reduce the number of false positives is to implement the membership set with a hash set instead of an STL set (which uses a balanced tree). This will reduce interference as transactions will no longer access the structure extensively for operations such as tree rebalancing. However, even the use of a hash set does not eliminate the potential for false positives. Note that if two distinct elements hash to the same bucket, transactional memory will still detect interference even if the two transactions access disjoint sets of elements and therefore should not interfere. This problem is exacerbated because, in general, a transaction adds and removes several elements from the mesh (as it removes all the elements in the cavity and inserts all the newly created elements), increasing the likelihood of low level interference.

High Level Transactional Memory

As we see, regardless of the concrete implementation of the mesh data structure, transactional memory still has the potential to detect irrelevant interference and hence trigger unnecessary roll backs.

We believe that what may be necessary is a transactional memory implementation that is aware of high level abstractions. Thus, to this implementation, it would be apparent that the insertion and removal of distinct elements from the mesh should not trigger interference, even if at the low level a typical transactional memory implementation would have detected interference. Implementing such a transactional memory may require significant work, as the transactional memory should ensure that the operations on the structure that do not trigger interference still complete correctly when executed concurrently.

6. Conclusion

We presented Delaunay mesh generation, an algorithm for producing guaranteed-quality meshes. We showed how this algorithm can be easily parallelized by processing multiple elements from the set concurrently. However, this approach requires that the cavities created during processing do not interfere with each other. We demonstrated how optimistic parallelism, and specifically the transactional model, provides a straightforward, easily implemented way to guarantee that the parallel execution is correct.

We also discussed shortcomings in current transactional memory implementations that may hinder parallel efficiency: inefficient handling of conflicting transactions

as well as overly conservative interference detection. We feel that addressing these issues is an important avenue of exploration in transactional memory research.

We believe there are many other algorithms that have the properties similar to those of the Delaunay mesh generation algorithm — a worklist structure with infrequent dependences between work units, which cannot be determined statically. One example, also from the same general domain, is *advancing front mesh generation* [9]. In this scheme, the mesh is "grown" inwards from the boundaries by selecting segments on the boundary and adding a new point on the interior to create a new element. Effectively, this requires placing all the segments of the boundary into a queue, and as segments are processed, updating the queue with the new boundary segments. Advancing front mesh generation can be parallelized in much the same way as Delaunay mesh generation, and the use of a similar transactional approach holds promise.

References

[1] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, Washington, DC, USA, 2005. IEEE Computer Society.

[2] Christos D. Antonopoulos, Xiaoning Ding, Andrey Chernikov, Filip Blagojevic, Dimitrios S. Nikolopoulos, and Nikos Chrisochoides. Multigrain parallel delaunay mesh generation: challenges and opportunities for multithreaded architectures. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 367–376, New York, NY, USA, 2005. ACM Press.

[3] Brian D. Carlstrom, Austen McDonald, Hassan Chafi, JaeWoong Chung, Chi Cao Minh, Christos Kozyrakis, and Kunle Olukotun. The ATOMOS transactional programming language. In *PLDI '06: Proceedings of the Conference on Programming Language Design and Implementation*, 2006.

[4] L. Paul Chew. Guaranteed-quality mesh generation for curved surfaces. In *SCG '93: Proceedings of the ninth annual symposium on Computational geometry*, pages 274–280, New York, NY, USA, 1993. ACM Press.

[5] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402, New York, NY, USA, 2003. ACM Press.

[6] Maurice Herlihy, Victor Luchangco, Mark Moir, and III William N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, New York, NY, USA, 2003. ACM Press.

[7] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, New York, NY, USA, 1993. ACM Press.

[8] S. Horwitz, P. Pfieffer, and T. Reps. Dependence analysis for pointer variables. In *Proceedings of the SIGPLAN '89 Conference on Program Language Design and Implementation*, Portland, OR, June 1989.

[9] P. Ivanyi. *Finite Element Mesh Generation*. Saxe-Coburg Publishers, 2004.

[10] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. Logtm: Log-based transactional memory. In *HPCA '06: Proceedings of the 12th International Symposium on High Performance Computer Architecture*, 2006.

[11] J. Eliot B. Moss and Antony L. Hosking. Nested transactional memory: Model and preliminary architectural sketches. In *SCOOL '05: Sychronization and Concurrency in Object-Oriented Languages*, 2005.

[12] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. *SIGARCH Comput. Archit. News*, 33(2):494–505, 2005.

[13] Jim Ruppert. A new and simple algorithm for quality 2-dimensional mesh generation. In *SODA '93: Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, pages 83–92, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics.

[14] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3), May 2002.

[15] Jonathan Richard Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, May 1996.