Engineering a Compact Parallel Delaunay Algorithm in 3D *

Daniel K. Blandford
Carnegie Mellon University
dkb1@cs.cmu.edu

Guy E. Blelloch
Carnegie Mellon University
blelloch@cs.cmu.edu

Clemens Kadow
Carnegie Mellon University
clemens.kadow@web.de

ABSTRACT

We describe an implementation of a compact parallel algorithm for 3D Delaunay tetrahedralization on a 64-processor shared-memory machine. Our algorithm uses a concurrent version of the Bowyer-Watson incremental insertion, and a thread-safe space-efficient structure for representing the mesh. Using the implementation we are able to generate significantly larger Delaunay meshes than have previously been generated—10 billion tetrahedra on a 64 processor SMP using 200GB of RAM.

The implementation makes use of a locality based relabeling of the vertices that serves three purposes—it is used as part of the space efficient representation, it improves the memory locality, and it reduces the overhead necessary for locks. The implementation also makes use of a caching technique to avoid excessive decoding of vertex information, a technique for backing out of insertions that collide, and a shared work queue for maintaining points that have yet to be inserted.

Categories and Subject Descriptors

J.0 [**Computer Applications**]: General

General Terms

Algorithms, Performance.

Keywords

Delaunay, meshing, parallel, space-efficient

1. INTRODUCTION

We present a parallel algorithm for 3D Delaunay tetrahedralization. The algorithm is based on previous work [9] involving a compact data structure for representing 2D and 3D meshes, with an accompanying sequential algorithm. In this paper we discuss the design issues involved in creating a parallel algorithm to run on a 64-processor shared-memory machine.

SCG'06, June 5–7, 2006, Sedona, Arizona, USA.

The sequential algorithm is based on incremental insertion using the well-known Bowyer-Watson kernel [12, 43]. During the course of the algorithm a Delaunay triangulation of the current pointset is maintained. An incremental step inserts a new vertex into the mesh by determining the tetrahedra that violate the Delaunay condition. We use the idea of Clarkson and Shor [18] and maintain an association between uninserted vertices and the tetrahedra containing those vertices. We keep a work queue of tetrahedra whose interiors contain points; threads draw tetrahedra at random from the queue for processing.

The data structure we use is space-efficient: it maintains the meshing data in a compressed format in main memory, uncompressing the data only when necessary to process queries and updates. This allows it to consume only 50 bytes per vertex (in 3D) for meshing data. Using this structure we were able to build a sequential meshing algorithm that requires much less space than conventional algorithms (such as the Pyramid algorithm of Shewchuk [36]).

This algorithm can be made parallel by allowing parallel insertions. Implementing this efficiently, however, is nontrivial. We discuss the modifications necessary to improve the memory locality of the algorithm (very important on an SMP machine), avoid concurrent accesses to the same region of the mesh, and maintain a work queue that will avoid contention between threads. We also show how to bootstrap the algorithm using Pyramid, and discuss why this is required.

As a preprocessing step, our algorithm relabels the input vertices using x-y-z cuts so that vertices that are close spatially have similar labels. This is necessary for the compression of the structure and can be used to improve memory locality as well. For this purpose, our algorithm partitions the vertices into groups with contiguous labels. The data for each vertex is stored in a compact hashtable [8] corresponding to its group. This makes it likely that data for the vertices in an insertion will share a small number of cache lines.

To prevent concurrent accesses to the same region of the mesh, we use OpenMP [1] test-locks (not wait-locks) attached to the hashtables for each vertex group. We also use test-locks to prevent concurrent accesses to the work queue. The work queue is divided into subqueues, each with its own lock. Section 3 describes our experimentation comparing different queueing disciplines for their effect on performance.

We bootstrap the algorithm by growing the mesh sequentially (using the Pyramid algorithm of Shewchuk [36]) until it is sufficiently large to avoid excessive contention between threads. We then run parallel point location to associate all

uninserted vertices with simplices in the mesh. This parallel point location is very fast, so this bootstrapping represents an improvement even for the sequential algorithm.

We present results from running the algorithm on several distributions of data, plus a real-world mesh based on octree decomposition from the Quake project [40]. We analyze the effect of high-degree vertices on the algorithm's performance.

We have used the algorithm to generate a mesh of over 10 billion tetrahedra (using 1.51 billion vertices randomly chosen from the unit cube). Constructing this mesh took 5512 seconds for 64 1.15-GHz EV67 processors on an HP GS 1280 SMP machine. For comparison, using one processor on 1/64 as many vertices took 3202 seconds. Delaunay tetrahedralization requires $O(n \log n)$ work on uniform data, so it is not appropriate to compare runtimes on different problem sizes to measure speedup. However, we can say that our algorithm can insert 64 times as many vertices using 64 processors using only 1.7 times as much time.

All data (including vertex coordinates, mesh connectivity data, and the work queue) fits within a memory footprint of 197GB of RAM. For comparison, a conventional 3D meshing structure would require 9 eight-byte pointers per tetrahedron (four vertices, four neighboring tetrahedra, and a data pointer), for a total of 720GB for the mesh connectivity data alone.

The algorithm as presented is only used to construct a Delaunay mesh over a given set of points; however, the generalization to Delaunay refinement described for the sequential version [9] would apply equally well in the parallel case.

The main contributions of this work are as follows. We show that a 3D incremental insertion algorithm can be parallelized without assigning processors to separate regions of the mesh. We develop a concurrent thread-safe version of our compressed mesh representation. We describe techniques for improving the locality of access of our insertions (particularly important on an SMP machine). We demonstrate a tradeoff between queueing disciplines for the work queue of our insertion algorithm. Finally, we provide results for the largest 3D Delaunay mesh that has been generated, as far as we know.

1.1 Related Work

Parallel Delaunay. There has been significant previous work on parallel Delaunay algorithms, using three main approaches to avoid conflicts between threads.

The first approach is that of divide-and-conquer: The mesh is (recursively) partitioned in two regions, with each partition built by a separate processor. The border between the regions must be constructed separately. Aggarwal [3] described a 2D algorithm which constructed the border by joining the regions after they were built. Goodrich et al. [23] described a parallel 3D convex hull algorithm (the 3D convex hull problem is equivalent to the 2D Delaunay problem). Chen et al. [14] described a 2D algorithm which assigned certain points to both regions; this resulted in some duplicate work but meant that joining the regions involved only discarding duplicate triangles. Hardwick [26], Blelloch et al. [11], and Lee et al. [31] described algorithms that project the 2D points to a paraboloid in 3D, compute the lower convex hull, and use that to derive a border before building the regions. Amato et al. [4] and Chan et al. [13] gave parallel divide-and-conquer algorithms for the 4D convex hull problem, which is equivalent to the 3D Delaunay problem. Many of these algorithms are strictly theoretical (they do not have experimental results).

The second technique for parallel Delaunay meshes involves incremental insertion (using the Bowyer-Watson kernel [12, 43]). Most algorithms for incremental insertion avoid collisions by assigning a region of the mesh to each processor. Operations involving multiple regions of the mesh are handled by message-passing between processors. For this technique it is necessary to perform load-balancing between regions while still ensuring that each region's border is small. In 2D this was done by Okusanya and Peraire [33] and Chrisochoides and Sukup [17].

The region-per-processor technique was also used by Chrisochoides and Nave [15, 16] to produce a 3D parallel algorithm for a message-passing architecture. That work focused on minimizing the latency from interprocessor communication, a problem which we can avoid since our concern is with a shared-memory machine. Also, our work is on a greater scale: our largest mesh is 5000 times larger than theirs.

Kohout et al. [30, 29] describe a 2D incremental insertion algorithm which does not assign a region to each processor; instead, all processors draw from a global queue, similar to our own work. They report a speedup of up to 5.84 on eight processors. Their algorithm uses a DAG data structure for point location (whereas our algorithm associates points with tetrahedra to save memory). Kohout et al. also give a good survey of related work.

None of these consider space-efficiency of their representations. Our previous work [9] describes a 3D compact mesh representation supporting queries in $O(d(v_1, v_2))$ time, and updates in expected amortized $O(d(v_1, v_2))$ time, where $d(v_1, v_2)$ is the number of vertices having edges to both v_1 and v_2. That work also presents experimentation showing that the structure is time-efficient compared to the Pyramid algorithm of Shewchuk [36]. In more recent work [8] we describe a 3D compact mesh representation permitting queries in $O(1)$ time and updates in $O(1)$ expected amortized time; the associated constant on the space usage is significantly larger, though (we estimate by a factor of 2.6), so in this work we use the more compact representation of [9].

Compressed Meshes. There has been considerable work involving compressed meshes [20, 25, 39, 34, 35, 38, 28, 27, 22]. In three dimensions these methods can compress the topology of a mesh to less than a byte per tetrahedron [38]—about 6 bytes/vertex (not including vertex coordinates). These techniques, however, are designed for storing meshes on disk or for reducing transmission time, not for representing a mesh in main memory. They therefore do not support dynamic queries or updates to the mesh while in compressed form.

Another option for handling larger meshes is to maintain the mesh in external memory. To avoid thrashing, this requires designing algorithms for which the access to the mesh is carefully orchestrated. Several such external memory algorithms have been designed [24, 21, 19, 32, 42, 6, 41, 5]. Of particular note is the bucketed randomized insertion order scheme of Amenta et al. [5], which improves the memory locality of an out-of-core tetrahedralization algorithm by altering the insertion order of the vertices. This insertion order might combine with our own work to form an improved

out-of-core algorithm using compressed data structures with very strong memory locality. We discuss this further in Section 4.

2. THE SEQUENTIAL ALGORITHM

The sequential version of our algorithm is described in detail in our previous paper [9]; we will summarize it here. The compressed data structure is described in Section 2.1.

Reordering for Locality. For several purposes, involving both compression quality and locality of memory access, we found it important to ensure that vertices that were close spatially (*e.g.*, those likely to share edges in the mesh) had similar labels. To ensure this, as a preprocessing step we relabeled the vertices using x-y-z cuts.

Given a set of points, our algorithm first finds which of the x, y, and z axes has the greatest diameter. It finds the approximate median of that diameter and partitions the points using that median. The points on one side are labeled first, then the points on the other side. This is done recursively (and in parallel) to produce a labeling in which points that are near each other have similar labels.

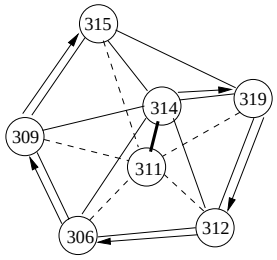
In previous work dealing with graph compression [7] we showed that, as long as each cut made in a mesh of size n intersects $O(n^c)$ edges for $c < 1$, and for bounded-degree vertices, the labeling produced leads to compression to $O(n)$ bits. For our Delaunay meshing application, we cannot guarantee ahead of time that each cut will intersect $O(n^c)$ edges (since the edges do not yet exist when we perform the relabeling). However, we find that using x-y-z cuts works well in practice.

If not all vertices are known before the algorithm begins, our algorithm can assign a sparse labeling to the initial vertices. When a new vertex is added, it is assigned a label that is close to the labels of its neighbors. In previous work [9] we presented results for a Delaunay refinement algorithm that made use of this technique. This algorithm could be made parallel in a straightforward fashion. We note that if a large number of vertices are inserted in one region of the mesh, it may be necessary to stop insertion and perform a global relabeling.

Sequential Insertion. We employ the well-known Bowyer-Watson kernel [12, 43] to incrementally generate the mesh. The algorithm maintains a Delaunay triangulation of the current pointset at all times. An incremental step inserts a new vertex into the mesh by determining the tetrahedra that violate the Delaunay condition. Those tetrahedra form the Delaunay *cavity*. The faces that bound the cavity are called the *horizon*. The mesh is modified by removing the tetrahedra in the cavity and connecting the new vertex to the horizon.

We use the idea of Clarkson and Shor [18] and maintain an association of every point p not yet inserted into the mesh with the tetrahedron t_p that contains p. The search for the cavity of p starts at t_p. With each tetrahedron we keep data indicating which uninserted points are contained in it. We maintain a work queue of tetrahedra which contain points.

At each step, the algorithm draws a tetrahedron from the the work queue. The algorithm checks that the tetrahedron is still in the mesh (that is, that an update has not deleted that tetrahedron since it was added to the queue). If so, the algorithm extracts a point p from the tetrahedron and performs the insertion. It uses the *bulldozing* idea described in [10] to reassociate points from the cavity with new tetrahedra. Any new tetrahedra that contain points are added to the work queue.



5	319	312	306	309	315

5	5	−2	−8	−5	1

Figure 1: The neighborhood and corresponding difference code data for the edge $314 \rightarrow 311$. The first entry, 5, is the degree of the vertex. Other entries are the offsets of the neighbors from 314.

2.1 Concurrent Compressed Data Structure

Here we summarize the data structure we use to represent our 3D meshes. The structure is adapted from that of our previous work [9]. In addition to supporting operations for adding a tetrahedron with associated data, deleting a tetrahedron, and finding a tetrahedron given a face, it supports operations for locking a vertex and unlocking all locked vertices. A vertex needs to be locked before any incident tetrahedra can be added or deleted.

Our structure represents the mesh by storing the *link* for a set of edges (1-simplices), such that every face is represented by at least one edge in this set. The link of an edge is the oriented cycle of vertices that connect to both endpoints of the edge (see Figure 1 for an example). The link is stored in a variable-bit-length hashtable [8] that maps the two vertices of the edge to the list of vertices in the link, and the data for the tetrahedra. For our incremental Delaunay algorithm the data is a uninserted vertex that lies within the tetrahedra. The link is compressed by difference encoding the vertex labels relative to one of the vertices on the edge and using variable bit-length codes for each difference [44].

Uninserted Points. A tetrahedron may contain more than one uninserted point. We represent these points using a linked list. We keep an array `next`$[0..N-1]$ such that, if point p is contained within a tetrahedron, then `next`$[p]$ is the index of another point within the same tetrahedron (or -1 if there is no such point). The first point in the list is stored with the tetrahedron in the mesh data structure.

Memory Locality. In an environment in which multiple threads are accessing a data structure, it is important to ensure that memory accesses involved in a query go to a small set of cache lines. Hashtables have notoriously poor memory locality; to address this, we divide the vertices into vertex groups of size G. (We used $G = 16$ in our experiments.) Each vertex group is allocated with its own hashtable; all data associated with the hashtable is kept in the same contiguous block. (If the hashtable requires resizing, the ad-

ditional memory must be allocated elsewhere.) Edges are stored in the hashtable corresponding to their first vertex. We take special care to avoid thread contention for memory pages by allocating all data for a vertex group in contiguous memory.

Along with the hashtable data we keep a data lock, shared by the G vertices of the vertex group; a thread must acquire the lock in order to read from or write to the hashtable. The space of the lock is amortized across the vertices. This can cause "false sharing" of locks since locking one vertex will lock all vertices in the group, but because of the locality of the labeling this is not a problem in our experiments.

Caching. To improve the efficiency of lookups our implementation uses a caching system. When a query or update is made, the compressed codes associated with the appropriate edge are decoded. The information is represented in uncompressed form as a linked list with one listnode per vertex in the link of the edge. The lists are kept in a cache. Update operations may affect the lists while they are in the cache. As part of an update, the application may delete simplices, producing holes in the mesh; however, we maintain the invariant that edge links that are written out of the cache must be full cycles. Thus the cache is only flushed after a new vertex insertion is complete.

We maintain the invariant that each thread has the lock on any lists that are in its cache. A thread releases all its locks at once, when the cache is flushed after an insertion.

2.2 Parallel Algorithm

The parallel version of the algorithm is the same as the sequential version except that we maintain a set of parallel threads, each of which draws work from the queue. To avoid overlapping reads and writes between threads we use data locks in two ways: on the mesh and on the work queue. All data locks are "test-locks" rather than "wait-locks": if a thread fails to acquire a lock, it aborts the operation rather than waiting for the lock to become free.

As a thread explores the cavity for a point p, it secures the lock on each vertex it encounters. (Recall that the vertices have been relabeled so that vertices with similar labels are close together; it is likely that many of the vertices for a cavity will share the same few locks.) If a thread encounters a vertex that is locked by another thread, it aborts the insertion: it releases all of its locks and returns the tetrahedron to the work queue. Otherwise, once the thread has secured the locks on all of the vertices of the cavity, it performs the insertion as normal and releases the locks when finished.

The work queue is also secured by locks to prevent concurrent access. In the parallel version for p processors the work queue contains $10p$ subqueues. (We experimented with several queue configurations—see Section 3 for details.) Each subqueue has its own separate lock; when a thread accesses the work queue, it probes the subqueues at random until it acquires the lock on one. The thread operates on the queue (adding a number of tetrahedra to be processed, or randomly extracting a tetrahedron for processing) and then releases the lock.

In rare cases it may be necessary for a thread to allocate more memory using calls to `malloc`. (For example, this is needed if a hashtable overflows.) To do this a thread must wait until it acquires a global lock. This is the only time in our algorithm when a thread waits to acquire a lock.

Contention. When the mesh is very small compared to the number of threads operating on it, there is danger of contention: multiple threads may all compete for the same few vertices, such that for a long time, no thread is able to acquire enough vertex locks to perform an insertion in a certain area of the mesh. This may result in a few very large tetrahedra remaining untouched, with many uninserted vertices on them, while other areas of the mesh are tetrahedralized to a fine resolution.

An easy solution to the contention problem is to hold some threads back at the start of the algorithm. Experimentally we find that restricting the density of threads to one per 2^{14} vertices in the mesh is sufficient to eliminate contention almost entirely. Unfortunately, this causes other types of slowdown: for the initial 2^{14} vertex insertions, only one thread is active in the mesh.

To see why this is a problem, recall that our point-location scheme involves keeping an association between each tetrahedron and the uninserted vertices contained in it. After each insertion, our algorithm must perform planeside tests for the uninserted vertices that lay in the deleted tetrahedra. If k of the n vertices have been inserted, then there are an expected $\Omega(n/k)$ vertices per insertion that require planeside tests. (In particular, the first insertion performed requires $\Theta(n)$ planeside tests for the uninserted vertices.) Performing all of these tests with one thread is inefficient.

Bootstrapping via Pyramid. To run our algorithm in parallel, we need to build the mesh sufficiently large that all threads can use it at once. To do this we make use of a separate tetrahedralization algorithm—the serial Pyramid algorithm of Shewchuk. That algorithm is different from ours in that it does not associate uninserted vertices with tetrahedra; instead, to insert a vertex v, it walks through the mesh using plane-side tests to locate the tetrahedron that should contain v. If the mesh has size k, then the expected path length to the target vertex is $O(k^{\frac{1}{3}})$; the use of multiple-starting-point heuristics reduces this cost to $O(k^{\frac{1}{4}})$.

Our bootstrapping algorithm works as follows. Given n vertices and p processors, we first relabel the vertices using x-y-z cuts, as in the standard algorithm. We then sample k vertices for insertion via Pyramid. (We could perform the sampling at random; however, since we have already relabeled the points using x-y-z cuts, we instead sample at evenly spaced intervals. This produces a better point distribution.) Once the Pyramid mesh data structure is built, we perform point location on the remaining vertices to associate them with tetrahedra in the mesh.

Each processor performs point location on a contiguous block of vertices. Since this does not involve modifying the mesh it produces no conflicts between threads. Shewchuk's point location routine allows us to begin the walk from any tetrahedron in the mesh. Since the vertices have high spatial locality (due to our reordering via x-y-z cuts), we begin the walk for each vertex v from the tetrahedron that contained vertex $v-1$. The cost for this point location is thus quite low.

When all the vertices have been mapped to tetrahedra, the Pyramid mesh structure is deallocated. The work queue is allocated, and the tetrahedra are inserted into it. (The space used for Pyramid is reused for the work queue, so that it does not add to the total space cost of the algorithm.) Our parallel insertion algorithm then begins as normal.

There is a tradeoff between insertion of vertices using Shewchuk's mesh-walking code and our bulldozing code. If there are n total points, and k points have been inserted into the mesh, then inserting a vertex using our code requires $\Theta(n/k)$ work (spent using planeside tests to reassociate the points in the cavity with new tetrahedra). The cost of the same insertion using Pyramid is $\Theta(k^{\frac{1}{4}})$ serial time, which is equivalent to $\Theta(pk^{\frac{1}{4}})$ work.

To optimize performance we must select a k such that these costs are balanced. Solving the expression $\frac{n}{k} = pk^{\frac{1}{4}}$ yields $k = (\frac{n}{p})^{\frac{4}{5}}$. For our experimental setup, however, we always use $n = 2^{24.5}p$: the same k should be valid throughout. Experimentally we find that k should be between 2^{19} and 2^{20} for best performance. With 64 processors, our initial mesh needs roughly 2^{20} vertices to avoid contention. Accordingly we use bootstrapping of 2^{20} vertices for all of our tests.

Cleanup. As the algorithm nears termination, it may occur that only one region of the mesh still contains uninserted vertices. In this case, the algorithm may encounter contention. To prevent this, threads leave the mesh as the number of remaining uninserted vertices decreases: thread k leaves the mesh when fewer than $2048k$ uninserted vertices remain. Since the last insertions are quite rapid (as they involve almost no planeside tests), this does not cause significant slowdown. Other techniques, such as exponential backoff on contention, might also have worked here.

3. EXPERIMENTATION

Experimental Setup. The system used for our experiments was `rachel.psc.edu` [2], a pair of HP GS 1280 SMP machines with 64 1.15-Ghz EV67 processors each. The operating system was Tru64 Unix. We used the OpenMP [1] library to provide parallel functionality. Our code was written in C and C++; we compiled using the command `cxx -O -fast -arch ev7 -tune ev7 -omp`.

There were 4 Gbytes of RAM available per processor. Given our space usage (discussed below) this was sufficient to build a mesh of about $2^{24.5}$ (about 23 million) vertices per processor.

We used the exact arithmetic predicates of Shewchuk [37] for all geometric tests. Additionally we used the beta version of Shewchuk's Pyramid code [36] to bootstrap our main parallel algorithm, as described above.

Main Results. We ran our algorithm on points with the uniform distribution using between 1 and 64 processors. In all cases we used $2^{24.5}$ (about 23 million) points per processor; thus, if our algorithm featured perfect speedup, all runs would take the same amount of time. We used a fixed amount of bootstrapping (2^{20} vertices) for each run. In the one-processor case our algorithm took 3202 seconds, for an average of 7410 vertices/second. In the 64-processor case our algorithm averaged 275490 vertices/second. The vertex insertion rate increased by a factor of 37.18 in the 64-processor case. However, note that (for uniform random data) this algorithm requires $O(n \log n)$ work, so it is not correct to compare two runs of different sizes for speedup. The work done per processor in the 64-processor case is a factor of $\frac{30.5}{24.5}$ more than the work per processor in the one-processor case. After accounting for this, we estimate the actual speedup of our algorithm to be 46.28 on 64 processors.

We decompose the runtime of our algorithm into several factors (see Table 1). The total runtime listed includes all steps of the algorithm from x-y-z reordering to termination. The next time measurement given includes only the parallel loop (that is, without the bootstrapping, reordering, or initialization phases). For convenience of analysis we divide the parallel loop into *stages* $s_0 \ldots s_{15}$, each of which involves inserting 1/16 of the total pointset. We give the insertion rate during s_0, s_{10}, and s_{15} as examples of how the cost of an insertion changes over time. Note that, at the start of phase s_0, 2^{20} vertices have already been inserted by bootstrapping. The 64−processor case begins with many more uninserted vertices per tetrahedron than the 1−processor case, so the algorithm must perform more point location work per insertion. This accounts for the dramatic slowdown during s_0.

Finally, we give three measures of contention. A lock failure is classed as an *initialization failure* if the thread fails to obtain the lock on one of the vertices in the initial tetrahedron, or a *dig failure* if the thread fails to obtain the lock on a some other vertex while computing the cavity for the insertion. If the failure occurs immediately after a previous failure, it is instead classed as a *repeat failure*. We give the average number of each type of failure per processor. The large number of initialization failures is due to our use of the FIFO queueing discipline, as discussed below.

The 64-processor run inserted $1,518,041,200$ points, producing $10,274,246,916$ tetrahedra. As far as we know this is the largest tetrahedral Delaunay mesh that has been generated.

Queueing Disciplines. In our algorithm there is a central work queue from which all threads draw tetrahedra for processing. To avoid concurrency issues, the queue is divided into a number of subqueues; when a thread wishes to access the queue, it chooses randomly from the subqueues until it finds one that is not in use. Here we discuss the issues involved in design of the work queue.

We considered three possible queueing disciplines for our work queue. The first we considered was the standard FIFO queueing discipline. A concern with the FIFO discipline is that, on completion of an insertion, our threads may add to the queue a large number of tetrahedra that all share the same vertex (the newly inserted point). If two or more threads attempt to handle the tetrahedra resulting from a single push, then most (or all) of the threads will encounter locked vertices and abandon the job.

A second discipline we considered was the random queue (RAND): tetrahedra are added to the tail of the queue but extracted at random from any point within the queue. This ensured that our threads' access patterns were random. Unfortunately we found experimentally that it led to larger queue sizes than the FIFO queue: large numbers of "garbage" tetrahedra (those that no longer existed in the mesh) collected in the queue and were not removed until near the end of the algorithm.

The third option we considered was the "queue-random" discipline (QR), a compromise between the first two disciplines. A thread would initially attempt to draw a tetrahedron from the front of the queue; if work on that tetrahedron

processors:	1	2	4	8	16	32	64
Total runtime	3202s	3769s	4352s	4435s	4686s	5090s	5512s
Parallel loop	2995s	3553s	4063s	4159s	4416s	4725s	5064s
s_0, vtxs/p/sec	7130	5388	4221	3990	3454	3170	2853
s_{10}, vtxs/p/sec	7768	6809	6183	6043	5779	5439	5101
s_{15}, vtxs/p/sec	7678	7152	6596	6510	6263	5935	5712
Init Fails/p	0	3.8M	5.4M	6.2M	6.5M	6.7M	6.7M
Dig Fails/p	0	40K	65K	80K	90K	99K	106K
Rep Fails/p	0	1.1M	2.2M	2.9M	3.1M	3.3M	3.2M

Table 1: Performance measurements per processor for our algorithm. We inserted $2^{24.5}$ (about 23 million) vertices per processor.

Discipline	Init Fails	Rep Fails	Maximum Queue Size	Runtime (main loop)
FIFO (2p)	113M	234M	181M	4620s
FIFO (10p)	51M	23M	160M	4321s
QR (2p)	68M	19K	257M	4165s
QR (10p)	37M	11K	200M	4234s
RAND (2p)	32K	107	595M	4249s
RAND (10p)	32K	47	589M	4300s

Table 2: Impact of various queueing disciplines on our algorithm using 2 or 10 subqueues per processor. Our tests used $2^{27.5}$ (about 190M) vertices and 8 processors.

failed due to contention, the thread would next draw from a random point within the queue.

In addition to experimenting with various queueing disciplines, we performed experiments with varying the numbers of subqueues in the work queue. The FIFO queueing discipline problem occurred when multiple threads accessed the same subqueue within a short amount of time; by increasing the number of subqueues we hoped to make this event less likely. For p processors we experimented with using $2p$ and $10p$ subqueues.

The results of our experiments are shown in Table 2. We ignore failed attempts to lock the work queues, as these are rare and do not have a large cost. When attempting to lock the mesh, we classify lock failures as *initial failures* or as *repeat failures* depending on whether the thread had encountered a lock failure just prior to the current one. The increase in failures for the FIFO queueing discipline is quite dramatic, and the increase in queue size for the random disciplines equally so. However, the corresponding increase in runtime was fairly small since most of the failures occurred before significant work was performed. Thus, we chose to minimize the space used by the work queue: for our experiments we use the FIFO queueing discipline with $10p$ subqueues.

Space usage. Our algorithm allocates space for several purposes. The vertex coordinates use 24 bytes per vertex (three eight-byte floating-point values). The array `next`$[p]$, used to link together vertices in the same tetrahedron, uses 4 bytes per vertex. For the work queue we allocate two entries per vertex, each of which holds three vertices a, b, c of a tetrahedron containing uninserted vertices. (To find the fourth vertex, the algorithm performs a lookup on (a, b, c).) The work queue uses 24 bytes per vertex.

The mesh structure divides vertices into groups of $G = 16$; for each group it allocates a structure of 1160 bytes, or 72.5 bytes per vertex. This includes 160 6−byte blocks of memory for storing encodings of the links of edges, 16 7−byte blocks of memory mapping edges to the encodings of their links, a bit vector to handle allocation of the memory, and a pointer to additional memory if necessary. It also includes an OMP data lock.

When the hashtable for a group overflows, additional memory is allocated from the heap. The algorithm multiplies the size of the hashtable by 1.5, then performs a rehash of all data in the group. To save memory, the hashtable reuses the 160 6−byte blocks from the main structure, so that for the first rehash only 80 6−byte blocks are allocated from the heap. (The hashtable numbers the old blocks $0 \ldots 159$ and the new blocks $160 \ldots 239$.)

We chose settings such that an overflow occurs on 13.8% to 15.4% of groups in the tests for Table 1. The average cost of overflows is 4.30–4.77 additional bytes per vertex.

The algorithm allocates some fixed-size structures as well (caches and pools of linked list nodes), but the memory for these is negligible. The total space cost for our algorithm, then, is less than 130 bytes per vertex. Thus our 10−billion-tetrahedron computation used 197GB of RAM.

Point Distributions. We tested our algorithm on several different distributions of data, including uniform, Gaussian, kuzmin, and line-singularity distributions. Details on these distributions can be found in [11]. For each distribution we ran $2^{24.5}$ vertices on one processor and $2^{27.5}$ vertices on eight processors. We computed the total runtime required and the number of additional bytes of memory per vertex allocated. (The numbers given are in addition to the 124.5 bytes per vertex required in all cases.) Results are shown in Table 3.

We also ran tests on real-world data: a set of grid points based on an octree decomposition generated by the Quake project [40]. We used eight processors to compute a mesh over 2^{27} points. The problem of computing a tetrahedral mesh over grid points proved difficult, as our algorithm was not designed to handle the perfectly flat tetrahedra that result when four vertices lie at the vertices of a square. To handle this we introduced small random perturbations: we added a small random value to each coordinate of each vertex.

Even after doing this, though, we encountered some difficulty with the tetrahedralization. Our insertion algorithm begins with a single tetrahedron on four artificial "boundary" vertices $v_1 \ldots v_4$, chosen such that the tetrahedron con-

Distribution	Time	Additional Bytes/Vtx
uniform	3136s	4.30
normal	3164s	4.67
kuzmin	3182s	4.56
line	3147s	2.80

Distribution	Time	Additional Bytes/Vtx
uniform	4296s	4.69
normal	4301s	4.79
kuzmin	4478s	4.80
line	4301s	3.67

Table 3: Space used and time required by our algorithm for $2^{24.5}$ vertices and one processor (left) and for $2^{27.5}$ vertices and eight processors (right).

Random Perturbation	Boundary Size	Contention (Rep Fails)	Time (sec)
(uniform random)	9K	16M	3132
0—1	99K	20M	3054
0—.5	138K	62M	3230
0—.2	203K	316M	3895
0—.1	274K	836M	3954
0—.05	370K	2207M	7397
0—.02	525K	7976M	16544
0—.01	(too much contention, aborted)		

Table 4: Performance of our algorithm on 2^{27} fully random points (from the unit cube) versus 2^{27} points derived from the Quake project [40]. The points provided have a very large boundary, resulting in contention for the lock on the four bounding vertices. Adding randomness to the point locations makes less of the boundary "visible" to the boundary vertices, making the problem more tractable.

tains all of the points to be inserted. As the points are inserted, the vertices $v_1 \ldots v_4$ connect to the boundary of the mesh. For the random distributions we tested this did not pose a problem: the boundary of the mesh was no more than a few thousand vertices at most. For our octree-decomposition data, however, the boundary of the mesh was much larger. The degree of the vertices $v_1 \ldots v_4$ grew large enough that there was significant contention between processors attempting to perform insertions near the boundary of the mesh. This decreased the performance of our algorithm considerably.

We were able to solve the problem by adding more randomness to the points. The smallest distance between any two points in our octree-decomposition data was 6 units; we added a random value between 0 and 1 to every vertex coordinate. By doing this we decreased the boundary of the mesh to a reasonable size. Results are shown in Table 4.

One interesting feature of our octree-decomposition data was its labeling. For random data, as a preprocessing step our algorithm relabels the points using x-y-z cuts (as described in Section 2). For our octree-decomposition data we found this step was unnecessary: the points came preordered, with a labeling that produced compression superior to what our relabeling algorithm provided. (For uniformly-random points with our reordering, the mesh data for 2^{27} vertices required 1.13G blocks for edge data. For the 2^{27} octree points with our reordering, mesh data required 1.09G blocks for edge data. For the octree points without reordering, mesh data required 0.97G blocks, which left the hashtables somewhat underfull.) Accordingly, the results in Table 4 use the labeling provided by the data.

4. FUTURE WORK

Out-Of-Core Algorithms. Our algorithm and data structure could be extended to work in an out-of-core setting: because the vertices are relabeled for locality, most of the memory accesses for an insertion should be very close together. Keeping the representation compressed means that more of it could fit in RAM at once. Unfortunately, our algorithm as described performs insertions in an almost-random order. This could be improved by using a BRIO ("biased randomized insertion order") [5] to provide some locality between insertions. The work queues of our algorithm would need to be replaced with a series of $O(\log n)$ groups of work queues, one for each level of the BRIO.

High-Degree Meshes. We have shown that the algorithm behaves well on several distributions as long as the maximum degree of a vertex is bounded. When the mesh contains vertices of high degree (as for the octree-decomposition data, as discussed in Section 3), the competing threads suffer from contention for the high-degree vertices. Experimentally it seems that our code is tolerant of four vertices with total degree $138K$ (out of a total of $128M$ vertices shared among 8 processors), but performance suffers when the degree grows larger.

Part of this problem is from our coarse-grained locking mechanism: we divide our data structure into hashtables, and force threads to lock each hashtable they access. We allocate one hashtable per $G = 16$ vertices, and store the data for each edge in a hashtable corresponding to one of its vertices. We require that a thread acquire the lock on all of the vertices adjoining the cavity before performing an update. For correctness it is only necessary to lock the *edges* adjoining the cavity, not the vertices. A more conservative locking mechanism might be able to exploit this to tolerate high-degree vertices. However, to do this it would be necessary to distribute the edges of a high-degree vertex evenly among many hashtables, which might sacrifice the good memory-locality properties of the representation.

Also, even with this improvement, there exist 3D meshes in which *all* vertices have high degree. It is not clear how any parallel incremental-insertion algorithm could handle such meshes efficiently.

5. REFERENCES

[1] OpenMP. `http://www.openmp.org/`.

[2] `rachel.psc.edu`. `http://www.psc.edu/machines/marvel/rachel.html/`.

[3] A. Aggarwal, B. Chazelle, L. Guibas, C. O'Dunlaig, and C. Yap. Parallel computational geometry. *Algorithmica*, 3:293–327, 1998.

[4] N. M. Amato, M. T. Goodrich, and E. A. Ramos. Parallel algorithms for higher-dimensional convex hulls. In *IEEE Symposium on Foundations of Computer Science*, pages 683–694, 1994.

[5] N. Amenta, S. Choi, and G. Rote. Incremental constructions con BRIO. In *Proc. ACM Symposium on Computational Geometry*, pages 211–219, June 2003.

[6] L. Arge. External memory data structures. In *Proc. European Symposium on Algorithms*, pages 1–29, 2001.

[7] D. Blandford, G. Blelloch, and I. Kash. Compact representations of separable graphs. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, 2003.

[8] D. K. Blandford and G. E. Blelloch. Dictionaries using variable-length keys and data, with applications. In *Symposium on Discrete Algorithms*, 2005.

[9] D. K. Blandford, G. E. Blelloch, D. E. Cardoze, and C. Kadow. Compact representations of simplicial meshes in two and three dimensions. In *12th International Meshing Roundtable*, 2003.

[10] G. Blelloch, H. Burch, K. Crary, R. Harper, G. Miller, and N. Walkington. Persistent triangulations. *Journal of Functional Programming (JFP)*, 11(5), Sept. 2001.

[11] G. Blelloch, J. Hardwick, G. L. Miller, and D. Talmor. Design and implementation of a practical parallel Delaunay algorithm. *Algorithmica*, 24(3/4):243–269, 1999.

[12] A. Bowyer. Computing Dirichlet tessellations. *The Computer Journal*, 24:162–166, 1981.

[13] Chan, Snoeyink, and Yap. Primal dividing and dual pruning: Output-sensitive construction of four-dimensional polytopes and three-dimensional voronoi diagrams. *GEOMETRY: Discrete & Computational Geometry*, 18, 1997.

[14] M. Chen, T. Chuang, and J. Wu. Efficient parallel implementations of 2D Delaunay triangulation with high performance Fortran. In *Proceedings of 10th SIAM Conference on Parallel Processing for Scientific Computing*, 2001.

[15] N. Chrisochoides and D. Nave. Simultaneous mesh generation and partitioning for Delaunay meshes. In *Proceedings of the Eighth International Meshing Roundtable*, pages 55–66, 1999.

[16] N. Chrisochoides and D. Nave. Parallel Delaunay mesh generation kernel. *International Journal for Numerical Methods in Engineering*, 58:161–176, 2003.

[17] N. Chrisochoides and F. Sukup. Task parallel implementation of the Bowyer-Watson algorithm. In *Proceedings of the Fifth International Conference on Numerical Grid Generation in Computational Fluid Dynamic and Related Fields*, 1996.

[18] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete & Computational Geometry*, 4(1):387–421, 1989.

[19] A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E. Ramos. Randomized external-memory algorithms for some geometric problems. In *Proc. ACM Symposium on Computational Geometry*, pages 259–268, June 1998.

[20] M. Deering. Geometry compression. In *Proc. SIGGRAPH*, pages 13–20, 1995.

[21] F. K. H. A. Dehne, D. Hutchinson, A. Maheshwari, and W. Dittrich. Reducing I/O complexity by simulating coarse grained parallel algorithms. In *Proc. IPPS/SPDP*, pages 14–20, 1999.

[22] P.-M. Gandoin and O. Devillers. Progressive and lossless compression of arbitrary simplicial complexes. In *Proc. SIGGRAPH*, 2002.

[23] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proceedings of the 34th Annual Symposium on Foundations of Computer Science*, pages 714–723, 1993.

[24] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proc. IEEE Symposium on Foundations of Computer Science*, pages 714–723, Nov. 1993.

[25] S. Gumhold and W. Strasser. Real time compression of triangle mesh connectivity. In *Proc. SIGGRAPH*, pages 133–140, 1998.

[26] J. Hardwick. Implementation and evaluation of an efficient parallel Delaunay triangulation algorithm. In *Proceedings of Ninth Annual Symposium on Parallel Algorithm and Architectures*, 1997.

[27] M. Isenburg and J. Snoeyink. Face fixer: Compressing polygon meshes with properties. In *Proc. SIGGRAPH*, pages 263–270, 2000.

[28] Z. Karni and C. Gotsman. Spectral compression of mesh geometry. In *Proc. SIGGRAPH*, pages 279–286, 2000.

[29] J. Kohout, I. Kolingerov, and J. Zara. Practically oriented parallel Delaunay triangulation in E2 for computers with shared memory. *Computers and Graphics*, 28:703–718, 2004.

[30] I. Kolingerov and J. Kohuou. Optimistic parallel Delaunay triangulation. *The Visual Computer*, 18(8):511–5, 2002.

[31] S. Lee, C. Park, and C. Park. An improved parallel algorithm for Delaunay triangulation on distributed memory parallel computers. *Parallel Processing Letters*, 11:341–352, 2001.

[32] S. McMains, J. M. Hellerstein, and C. H. Squin. Out-of-core build of a topological data structure from polygon soup. In *Proc. Symposium on Solid Modeling and Applications*, pages 171–182, June 2001.

[33] T. Okusanya and J. Peraire. 3D parallel unstructured mesh generation. *AMD*, 220:109–115, 1997.

[34] R. Pajarola, J. Rossignac, and A. Szymczak. Implant sprays: Compression of progressive tetrahedral mesh connectivity. In *Proc. Visualization 99*, pages 299–306, 1999.

[35] J. Rossignac. Edgebreaker: Connectivity compression for triangle meshes. *IEEE Transactions on Visualization and Computer Graphics*, 5(1):47–61, 1999.

[36] J. R. Shewchuk. Pyramid mesh generator software. (`http://www.cs.berkeley.edu/~jrs/`). Personal Communication.

[37] J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete and Computational Geometry*, 18(3):305–368, 1997.

[38] A. Szymczaka and J. Rossignac. Grow & Fold: compressing the connectivity of tetrahedral meshes. *Computer-Aided Design*, 32:527–537, 2000.

[39] G. Taubin and J. Rossignac. Geometric compression through topological surgery. *ACM Transactions on Graphics*, 17(2):84–115, 1998.

[40] T. Tu and D. R. O'Hallaron. A computational database system for generating unstructured hexahedral meshes with billions of elements. In *Proceedings of SC2004*, 2004.

[41] T. Tu, D. R. O'Hallaron, and J. C. Lopez. ETREE - a database-oriented method for generating large octree meshes. In *Proc. International Meshing Roundtable*, pages 127–138, Sept. 2002.

[42] J. S. Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, June 2001.

[43] D. F. Watson. Computing the n-dimensional Delaunay tesselation with application to Voronoi polytopes. *The Computer Journal*, 24:167–172, 1981.

[44] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes*. Morgan Kaufman, 1999.