Delaunay Triangulation with Transactions and Barriers

Michael L. Scott, Michael F. Spear, Luke Dalessandro, and Virendra J. Marathe

Department of Computer Science, University of Rochester
{*scott, spear, luked, vmarathe*}*@cs.rochester.edu*

Abstract

Transactional memory has been widely hailed as a simpler alternative to locks in multithreaded programs, but few nontrivial transactional programs are currently available. We describe an open-source implementation of Delaunay triangulation that uses transactions as one component of a larger parallelization strategy. The code is written in C++, for use with the RSTM software transactional memory library (also open source). It employs one of the fastest known sequential algorithms to triangulate geometrically partitioned regions in parallel; it then employs alternating, barrier-separated phases of transactional and partitioned work to stitch those regions together. Experiments on multiprocessor and multicore machines confirm excellent single-thread performance and good speedup with increasing thread count. Since execution time is dominated by geometrically partitioned computation, performance is largely insensitive to the overhead of transactions, but highly sensitive to any costs imposed on sharable data that are currently "privatized".

1. Introduction

With the recent end of thirty years of steady improvement in the speed of CMOS uniprocessors, all the major vendors are now building multicore chips. As a result, programming techniques and challenges once reserved to high-end scientific computing have become relevant across the computing spectrum: every software vendor will soon need to be writing multithreaded code. This prospect poses enormous challenges for the software industry, among them the traditional synchronization-based tradeoff between scalability and the complexity of fine-grain locking protocols. Transactional memory (TM) promises to soften this tradeoff. In a nutshell, TM allows programmers simply to mark their critical sections *atomic* and rely on lower levels of the system to run those sections in parallel with one another whenever possible.

Some two dozen TM systems have been described in the literature over the past few years, some implemented in hardware, some in software, and some in a combination of the two [11]. Unfortunately, the field has yet to develop standard benchmarks to capture application characteristics or to facilitate system comparisons. The closest we have at the moment are (multiple versions of) shared data structures (hash tables, red-black trees, linked lists) that are likely to be useful *within* transactional applications, but that do not convincingly capture the behavior of (still mostly hypothetical) "real world" applications.

Several groups are currently developing more ambitious applications, either from scratch or from existing lock-based code. One challenge is the lack of a standard language and API. To date, TM systems have been developed for (at least) C, C++, Java, C#, Haskell, OCaml, Python, and Common Lisp, and the multiple proposals for C, C++, Java, and C# employ mutually incompatible syntax. This diversity of notation is unlikely to disappear soon. Until standards emerge, benchmarks written for one system will require translation to be used on other systems. Even so, they can serve a useful role by codifying application-level algorithms and by increasing the degree to which experiments run on different systems can fairly be compared to one another.

This paper describes an implementation of Delaunay triangulation [5]. Given a set of points $\mathcal{P}$ in the plane, a *triangulation* partitions the convex hull of $\mathcal{P}$ into a set of triangles such that (1) the vertices of the triangles, taken together, are $\mathcal{P}$, and (2) no two triangles intersect except by sharing an edge. A *Delaunay* triangulation has the added property that no point lies in the interior of any triangle's circumcircle (the unique circle determined by its vertices). If not all points are colinear, a triangulation must exist. If no four points are cocircular, the Delaunay triangulation is unique.

A Delaunay triangulation maximizes the minimum angle across all triangles. It is a superset of the Euclidean minimum spanning tree of $\mathcal{P}$. It is also the geometric dual of Voronoi tessallation, used to quickly identify, for a given point in the plane, the closest member of $\mathcal{P}$. It is widely used in finite element analysis, where it promotes numerical stability, and in graphical rendering, where it promotes aesthetically pleasing shading of complex surfaces. In practice, Delaunay meshes are typically *refined* by introducing additional points where needed to eliminate remaining narrow triangles.

At the 2006 Workshop on Transactional Workloads, Kulkarni et al. proposed refinement of a (preexisting) Delaunay mesh as an ideal application of transactional memory [10]. The current paper complements that work by using transactions to

This work was supported in part by NSF grants CNS-0411127 and CNS-0615139, equipment support from Sun Microsystems Laboratories, and financial support from Intel and Microsoft.

construct the original triangulation. In future work we hope to extend our code to include the refinement process.

Our algorithm begins by sorting points into geometric regions, one for each available processor. It then employs a fast sequential solver to find, in parallel, the Delaunay triangulations of the points in each region. Specifically, we employ Dwyer's refinement [7] of Guibas and Stolfi's divide-and-conquer algorithm [8]. For uniformly distributed points, Dwyer's algorithm runs in $O(n \log \log n)$ time. Given triangulations of each region, we employ a mix of transactions and thread-local computation to "stitch" the regions together, updating previously chosen triangles when necessary to maintain the Delaunay circumcircle property. Related, hierarchical stitching algorithms for a message-passing model have been presented by Blelloch et al. [1], Hardwick [9], and Chen et al. [2], among others. All of these remain efficient when points are not uniformly distributed. Our code could be extended with a parallel median-finding algorithm (as in Chen et al.) to provide this property; in its current form it distributes points with a simple bucket sort.

The following section provides a brief overview of the Rochester Software Transactional Memory (RSTM) system and its API, sufficient to understand the mesh application code. Section 3 describes our parallel solver, focusing on its use of transactions and of barrier-separated transactional and geometrically-partitioned ("private") phases. Section 4 presents a brief summary of performance results on both a conventional multiprocessor and a more recent multicore machine.

2. The RSTM API

RSTM [19] is a transactional memory library for C++/pthreads programs. Its API and programming model [4], while too baroque for naive users, are simple and flexible enough to express sophisticated applications and to support experimentation with a wide variety of underlying implementations. Applications written to the RSTM API can be compiled and linked with several different implementations, most of which have several variants. Three of these appear in the results of Section 4: The original *RSTM* back end [12] is a nonblocking software system that uses a level of indirection to install new versions of modified objects atomically at commit time.[1] The *redo-lock* back end is a blocking system that resembles Dice et al.'s TL2 [6]. It copies new versions back on top of the originals at commit time, avoiding the need for indirection when reading. The *CGL* (coarse-grain-lock) back end is intended for comparison purposes: its transactions compete for a single, global lock, yielding very low overhead in the uncontended case, but no concurrency. Several other systems employ a mix of hardware and software support; they currently run only on our hardware simulator [15]–[17].

1. In a nonblocking system, no reachable state of the system as a whole precludes forward progress by any given thread; among other things, a thread need never wait for a preempted peer.

2.1. Transactional Objects

Every thread in an RSTM program begins by calling the function `stm_init`. Transactions are delimited by the macros `BEGIN_TRANSACTION` and `END_TRANSACTION`. These introduce a nested scope; they must be syntactically paired. When two concurrent transactions conflict (they access the same object, and at least one of them tries to write it), one is allowed to proceed and the other waits or retries, under control of a software *contention manager*. By default, RSTM employs the "Polka" contention manager [14], which gives preference to transactions in which substantial work has already been invested.

The programming model distinguishes between transactional and nontransactional objects. When a transaction retries, transactional objects revert to the values they had at the beginning of the transaction. Nontransactional objects do not. Any class of shared objects `T` whose instances are to be protected by transactions must derive from generic class `stm::Object<T>`, which provides metadata and (hidden) methods for conflict detection. `T` must also provide a `clone` method, used to initialize new versions of an object and, for redo-lock, a `redo` method, used to copy an updated version back to the original. If any action is required when discarding an outdated version, the user can provide an optional `deactivate` method.

To catch common mistakes, and to provide hooks for hidden initial-access and every-access methods, transactional objects must be accessed only through generic "smart pointers" provided by the API. These come in four varieties: `sh_ptr<T>`, `rd_ptr<T>`, `wr_ptr<T>`, and `un_ptr<T>`. Pointer fields within transactional objects must be `sh_ptr`s. They can safely be copied and tested for equality, both inside and outside transactions, but cannot be dereferenced. The other three pointer types can be initialized from `sh_ptr`s. Within a transaction, a `wr_ptr` allows both reading and writing; a `rd_ptr` allows only reading. An `un_ptr` allows both reading and writing, but avoids transactional overhead; its use constitutes a promise from the programmer that the object in question has been "privatized", and is visible to only one thread.

To allow hidden methods to be called at appropriate times, fields of transactional objects must be read and written only through *accessor* methods. These are generated automatically by field-declaring macros. In the mesh application, for example, the `edge` class appears as in Figure 1. Given this declaration, we can update fields as follows:

```
p->set_tentative(true);
p->set_neighbors(0, cw, other_edge);
```

With the original RSTM back end, they are read as follows:

```
status = p->get_tentative();
nbor = p->get_neighbors(0, cw);
```

To ensure correct behavior in the presence of in-place updates, the redo-lock back end requires an additional "validator"

```
class edge : public stm::Object<edge> {
public:
  GENERATE_ARRAY(point*,points,2)
    // point* points[2];
  GENERATE_2DARRAY(sh_ptr<edge>,neighbors,2,2)
    // sh_ptr<edge> neighbors[2][2];
    // indexed by edge end and direction
  GENERATE_FIELD(bool,tentative)
    // Is this edge potentially non-Delaunay?
}
```

Figure 1. Declaration of transactional `edge` class.

argument:

```
status = p->get_tentative(p.v());
nbor = p->get_neighbors(0, cw, p.v());
```

Validators are also required for some forms of privatization (Section 2.3). They are ignored in the mesh application when using the RSTM back end.

2.2. Programming Model Restrictions

To avoid violating assumptions in one or more of the back-end systems, programmers must respect several additional (unenforced) restrictions on the programming model.

- Transactional types must have trivial constructors and destructors, and only static methods. Specifically, a constructor must not, under any circumstances, throw an exception or conflict with another transaction (which might cause it to abort). A destructor must not do anything that has to happen at `delete` time—the memory manager delays space reclamation to avoid errors in concurrent transactions. And since fields must be accessed through smart pointers, `this` cannot safely be used.
- Control must leave a transaction only by falling off the end—never via `return`, `break`, or `goto`. A `throw` from inside a transaction causes the transaction to abort, and then continues to propagate.
- An `un_ptr` cannot be used once its object has become visible to other threads. If an object is "privatized" repeatedly, an `un_ptr` to it cannot be retained across phases.

2.3. Privatization

Many natural uses of transactions embody what Larus and Rajwar call *privatization* [11, Section 2.1.2]: program logic that ensures an object is accessible to only one thread during some particular program phase. In the mesh application, privatization is achieved via global consensus, using barriers. During a "private phase", we partition the point space geometrically, and each thread operates only on edges guaranteed to lie entirely within its region. In other programs, a transaction may privatize an object by removing it from a shared data structure. Such *privatizing transactions* introduce implementation complexities not considered here [18].

2.4. Useful Idioms

In several cases, utility routines in the mesh application are called in both transactional and private phases. Examples include routines to create a new edge, destroy an existing edge, or reconsider the inclusion of an edge that may not be Delaunay. Each of these has smart pointer parameters that will be `un_ptrs` in private code and `rd_ptrs` or `wr_ptrs` in transactional code. We use generics (templates) to generate both versions from a common code base. In a few cases we use a `txnal<`*pointer_type*`>()` generic predicate to choose between alternative code paths. We also provide generic type constructors that allow the user to declare pointer types analogous to (i.e., transactional or nontransactional, as appropriate) some other pointer type. For example, if `edgeRp` is a template parameter that may be either `rd_ptr<edge>` or `un_ptr<edge>`, then `edgeRp::w_analogue<foo>::type` will be `wr_ptr<foo>` or `un_ptr<foo>`, respectively.

In one natural realization of the mesh algorithm, `point` objects contain x and y coordinates and a smart pointer reference to some adjacent edge. While the `first_edge` field is occasionally read or written, such references are overwhelmingly outnumbered by simple reads of the (immutable) coordinates. To avoid transactional overhead on every such immutable reference, we manually split the `point` class into mutable and immutable portions and use address arithmetic to find one given the other (such *structure splitting* [3] is a well-known optimization in high-end compilers). References to coordinates can now proceed with zero overhead.

Because nontransactional (nonshared) objects do not revert their values on abort, care must be taken when communicating information across transaction boundaries. In particular, a transaction can safely read or write a nontransactional variable (assuming, in the latter case, it *always* writes before committing), but not both.

3. Parallel Delaunay Triangulation

All told, the mesh application comprises approximately 3200 lines of code and 24 source files. It is included in the RSTM distribution [19].

3.1. Algorithm

Function `main` parses command line arguments, creates a set of random points (or reads them from standard input), logs the application start time, forks and joins a set of worker threads, and prints the final run time. Workers run function `do_work` in `worker.cc`. This function has three major phases:

1) (2 internal barriers) Workers partition the point space into vertical bands (regions) by giving each worker a contiguous, equal-size slice of the x coordinate range. As noted in Section 1, a more sophisticated algorithm would be required to accommodate nonuniform point distributions.

2) Each worker triangulates the points in its own region, using Dwyer's sequential algorithm. This phase consumes most of the execution time. It falls short of ideal speedup because of memory bandwidth limitations.
3) (8 internal barriers) Each worker stitches its region together with the region to the right. The last worker (and any other whose own or neighboring region is empty) bows out of the computation. This phase has several sub-phases, illustrated in Figure 2:
   a) (one internal barrier) Each worker connects its right-most point to the neighbor's left-most point. This initial cross edge is guaranteed not to intersect any intra-region edge, but it may not be Delaunay, and it may (in the case of singleton regions) share a point with some other initial cross edge. To avoid races, each worker uses a transaction to create its first cross edge.
   b) Working down from the cross edge, each worker uses function `baste` to create additional edges between its region and its right hand neighbor's, so long as it never touches a point that is geometrically closer to a neighbor's seam. Both the newly created edges and the edges traversed on the regions' convex hulls may be non-Delaunay.
   c) Continuing down the seam, each worker uses function `synchronized_baste` to create additional cross edges. Because responsibility for points is no longer geometrically determined, this phase uses transactions to mediate conflicts.
d, e) Mirroring sub-phases 3b and 3c, workers return to the initial cross edge and work upward, first on known-to-be-private edges, then on transaction-mediated edges. Through all these sub-phases, workers keep a record of "tentative" (potentially non-Delaunay) edges. With the completion of this sub-phase, we have a complete triangulation, but not one known to be Delaunay.
   f) Each worker w iterates over its list of tentative edges. If edge $\overline{ab}$ is a diagonal of a convex quadrilateral, all four points of which are closest to w's seam, w considers "flipping" the edge as shown in Figure 3. For each flip $\langle \overline{ab} \rightarrow \overline{cd} \rangle$, w adds $\overline{ac}$, $\overline{ad}$, $\overline{bc}$, and $\overline{bd}$ to its list of tentative edges. If $\overline{ab}$ lies on the global convex hull, no further action is required. If any of a, b, c, or d lies closer to some other worker's seam, w defers consideration of a possible flip to the next sub-phase.
   g) Workers iterate over deferred tentative edges (and any other edges rendered tentative by the flipping of a tentative neighbor), using transactions to protect potential flips. If a worker runs out of work it steals from the tentative edge lists of other workers.

3.2. Data Structures

Three types are derived from `stm::Object`; instances of these are the only data whose values revert on transaction abort:

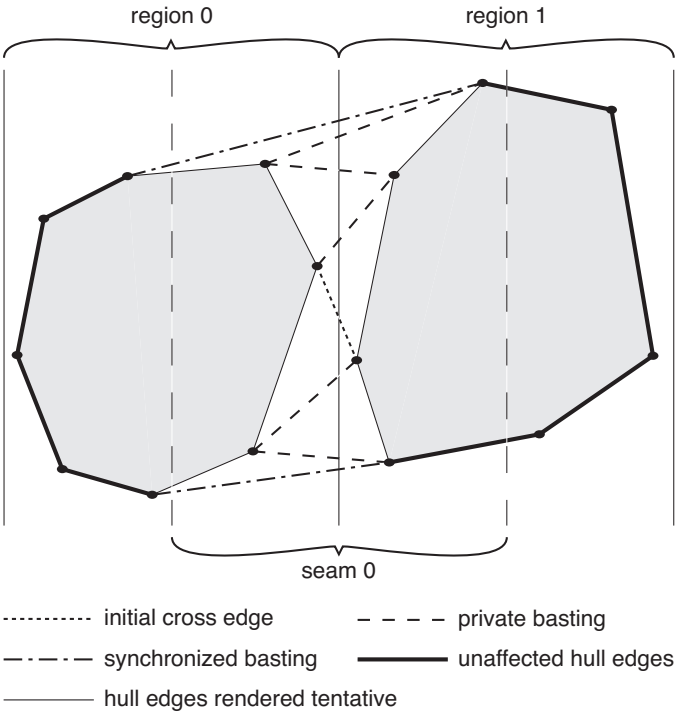

Figure 2. Stitching together Delaunay regions. Worker thread k is responsible for triangulating region k and, subsequently, stitching seam k.

`edge`: This is the benchmark's central type. As suggested by Guibas and Stolfi [8], it contains pointers to the edge's two endpoints and to the four neighboring edges found by rotating clockwise and counter-clockwise about those endpoints.

`edge_ref`: As noted in Section 2.4, the information about a point is split across two structures. The heavily-used part is immutable. The other is a reference to one adjacent edge, from which others can be found by following neighbor links. An `edge_ref` is this mutable part.

`LLNode`: The benchmark accumulates its triangulation in a set of edges, represented by an array of hash tables, one per geometric region. Type `LLNode` is used to build hash chains.

A Standard Template Library `queue` is used to hold tentative edges identified in steps 3b through 3e above. Edges that cannot be reconsidered locally in step 3f are moved to a nonblocking M&S queue [13], from which work can safely be stolen by other threads.

Geometric calculations are more-or-less straightforward, if tedious. Several operations are realized as methods of a `side` data structure designed for crawling along a convex hull. Separate versions of this structure are used for nontransactional (`baste`) and transactional (`synchronized_baste`) phases of the algorithm.

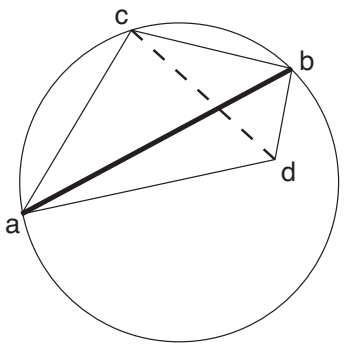

Figure 3. Reconsidering an edge. Edge $\overline{ab}$ should be *flipped* (replaced with edge $\overline{cd}$) if $adbc$ is a convex quadrilateral and circumcircle bca contains point d (or, symmetrically, circumcircle adb contains point c).

3.3. Static Transactions

Only 3 transaction blocks appear in the source code. The first, used in step 3a, protects a call to the static method that plays the role of `edge` constructor. This in turn calls 11 additional (non-accessor, non-library) routines, directly or indirectly, for a total (not including headers) of 72 lines of code. The second transaction, used in steps 3c and 3e, protects a 40-line block of code. The call tree under this includes 17 nontrivial subroutines, the bodies of which comprise an additional 115 lines of code. The third transaction, used in step 3g, protects a call to function `reconsider`. Together with the bodies of its 16 callees, it comprises 214 lines of code. 11 separate functions are templated for instantiation of transactional and nontransactional versions.

4. Sample Performance Results

Figure 4 summarizes performance of the mesh application on representative multiprocessor and multicore machines. We show curves for the RSTM and Redo-lock back ends described at the beginning of Section 2, together with coarse-grain (CGL) and fine-grain (FGL) locks. The CGL results use an alternative back end that adds no metadata or indirection to transactional objects, and implements `BEGIN_TRANSACTION` and `END_TRANSACTION` as acquisition and release, respectively, of a single, global, test-and-set lock. The FGL results use the same back end, to avoid metadata and indirection, and use `#ifdefs` to replace transactions with critical sections that acquire per-point locks. A critical section that touches multiple points identifies them speculatively, sorts them into canonical order, acquires their locks, and then double-checks to make sure the set of points is still valid. If not, it backs out and retries while basting, much as transactions do automatically. When reconsidering edges it simply backs out: a conflict indicates that another thread has already handled the edge.

All experiments triangulate a set of 200,000 uniformly distributed points. This is close to the data set size at which the Niagara (multicore) machine achieves its maximum speedup, and a little beyond the point at at which the SunFire (multiprocessor) does best. With fewer points the serial fraction of the application is proportionally larger; with more points cache locality deteriorates, making memory bandwidth the dominant factor in run time. Numbers shown are the average over three runs, with different seeds for the pseudorandom number generator used to create the point set. Speedups (effective processors) are relative to an optimized, single-threaded implementation of Dwyer's sequential algorithm.

Perhaps the most striking behavior in the graphs is the $\sim 2\times$ overhead of the RSTM back end. This is a direct consequence of its use of indirection to ensure nonblocking progress: the application is memory bound, and indirection doubles the cost of accessing transactional objects, even in private code. The extensive private computation in our Delaunay benchmark provides a powerful argument for zero-indirection TM systems.

Unsurprisingly, CGL has the best single-thread times, though it doesn't beat FGL and redo-lock by much. More surprisingly, CGL continues to equal or tie the performance of FGL and redo-lock at higher thread counts. We had originally expected serialization on a single lock to become a bottleneck as concurrency increased. The fact that it does not would seem to imply that transactional memory offers no advantage over coarse-grain locks in this particular program—neither performance nor convenience. The problem with this conclusion is that we see it only in hindsight. While CGL and TM are equally easy to use, they are not equally easy to refine. To choose CGL over TM at the outset is to gamble that bottlenecks will not arise. If they do, the conversion to fine-grain locks may take considerable time and introduce subtle errors.

The SunFire machine continues to obtain speedups out to the full complement of 16 processors, at which point stitching (the part of the execution that actually contains transactions) consumes just over 8% of total run time. The Niagara machine tops out at 8 threads—the same as the number of cores. While each core has 4 thread contexts, they don't help at this data set size, due to lack of memory bandwidth (they do help with smaller data sets). Here stitching consumes less than 1.2% of total run time, presumably due to the lower relative cost of multicore coherence misses.

The time consumed by stitching is highly repeatable for a given set of points, but varies across point sets (random number seeds) and numbers of worker threads. Together, these determine the pattern of transaction conflicts. At 200,000 points, the total amount of work required to stitch the graph together isn't enough to hide the variations. (The graph becomes smoother at larger point counts.) Nonetheless, we can see that the RSTM back end incurs a higher relative penalty on the SunFire machine as the number of threads increases, again due, we believe, to the higher relative cost of coherence misses, of which RSTM has about twice as many as the other back ends.

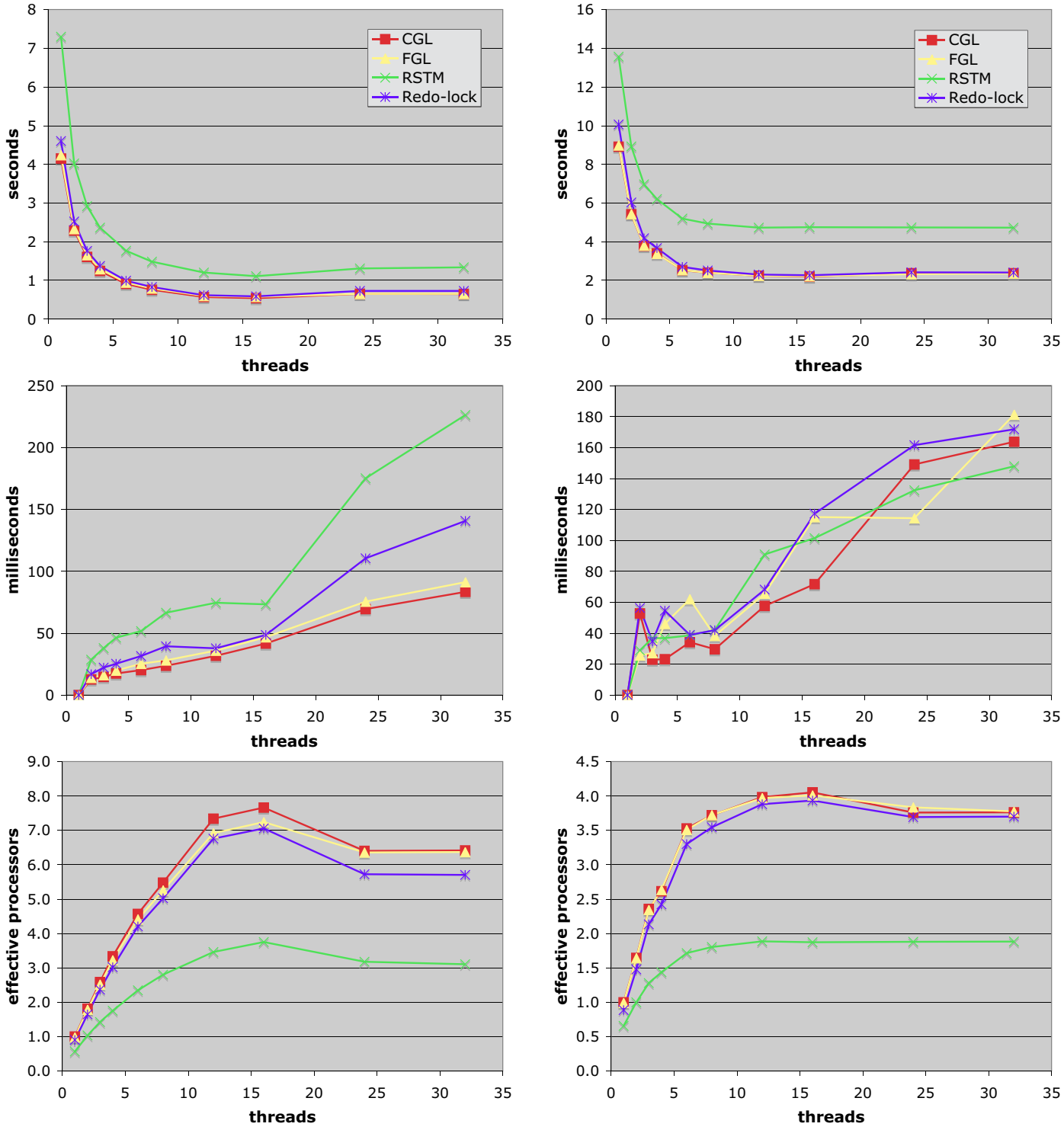

Figure 4. Total time (top), stitching time (middle), and speedup (bottom) for 200K-point triangulation. Left column: 16-processor, 1.2 GHz SunFire 6800. Right column: 8-core (32-thread), 1 GHz Sun T1000 (Niagara).

5. Conclusions

We have presented a parallel implementation of Delaunay triangulation. Among transactional benchmarks, our code is unusual in its extensive use of privatization and its mix of transactions and barriers. While somewhat memory-bound, it devotes relatively little time to synchronization, and scales well on the ∼16-processor machines on which it has been tested.

Our code was written to run above the RSTM C++ library package, available for open source download [19]. We draw three main conclusions from our work. First, it makes sense to mix transactions and barriers, and in particular to use transactions to "stitch together" spatially partitioned work performed between barrier episodes. Second, for such an application, a TM system must minimize—or better yet, eliminate—extraneous overhead on access to privatized data. This requirement argues strongly for "zero-indirection" TM systems. Third, while the RSTM API suffices to build non-trivial applications, it remains too complex to give to naive users [4]. Several groups have proposed compiler support as a way to improve the performance of transactions. We conjecture that compiler—and language—support will be even more important as a way to improve the programming model.

Acknowledgments

Our thanks to colleagues, past and present, who have contributed to the RSTM runtime and its variants: Athul Acharya, Sandhya Dwarkadas, David Eisenstat, Chris Heriot, Hemayet Hossain, Corey Proscia, Aaron Rolett, Bill Scherer, Arrvindh Shriraman, Vinod Sivasankaran, and Andrew Sveikauskas.

References

[1] G. E. Blelloch, G. L. Miller, and D. Talmor. Developing a Practical Projection-Based Parallel Delaunay Algorithm. In *Proc. of the 12th ACM Symp. on Computational Geometry*, Philadelphia, PA, May 1996.

[2] M.-B. Chen, T.-R. Chuang, and J.-J. Wu. Parallel Divide-and-conquer Scheme for 2D Delaunay Triangulation. *Concurrency and Computation—Practice and Experience*, 18:1595-1612, 2006.

[3] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-Conscious Structure Definition. In *Proc. of the SIGPLAN 1999 Conf. on Programming Language Design and Implementation*, Atlanta, GA, May 1999.

[4] L. Dalessandro, V. J. Marathe, M. F. Spear, and M. L. Scott. Capabilities and Limitations of Library-Based Software Transactional Memory in C++. In *Proc of the 2nd ACM SIGPLAN Workshop on Transactional Computing*, Portland, OR, Aug. 2007.

[5] B. Delaunay. Sur la Sphère Vide. *Bulletin of the USSR Academy of Sciences, Classe des Sciences Mathématiques et Naturelles*, 7:793-800, 1934.

[6] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proc. of the 20th Intl. Symp. on Distributed Computing*, Stockholm, Sweden, Sept. 2006.

[7] R. A. Dwyer. A Faster Divide and Conquer Algorithm for Constructing Delaunay Triangulation. *Algorithmica*, 2:137-151, 1987.

[8] L. Guibas and J. Stolfi. Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams. *ACM Trans. on Graphics*, 4(2):74-123, Apr. 1985.

[9] J. C. Hardwick. Implementation and Evaluation of an Efficient Parallel Delaunay Triangulation Algorithm. In *Proc. of the 9th Annual ACM Symp. on Parallelism in Algorithms and Architectures*, Newport, RI, June 1997.

[10] M. Kulkarni, L. P. Chew, and K. Pingali. Using Transactions in Delaunay Mesh Generation. In *Workshop on Transactional Memory Workloads*, Ottawa, ON, Canada, June 2006.

[11] J. R. Larus and R. Rajwar. *Transactional Memory*, Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2007.

[12] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the Overhead of Software Transactional Memory. In *Proc of the 1st ACM SIGPLAN Workshop on Transactional Computing*, Ottawa, ON, Canada, June 2006. Expanded version available as TR 893, Dept. of Computer Science, Univ. of Rochester, Mar. 2006.

[13] M. M. Michael and M. L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proc. of the 15th ACM Symp. on Principles of Distributed Computing*, Philadelphia, PA, May 1996.

[14] W. N. Scherer III and M. L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. In *Proc. of the 24th ACM Symp. on Principles of Distributed Computing*, Las Vegas, NV, July 2005.

[15] A. Shriraman, M. F. Spear, H. Hossain, S. Dwarkadas, and M. L. Scott. An Integrated Hardware-Software Approach to Flexible Transactional Memory. In *Proc. of the 34th Intl. Symp. on Computer Architecture*, San Diego, CA, June 2007. Earlier but expanded version available as TR 910, Dept. of Computer Science, Univ. of Rochester, Dec. 2006.

[16] M. F. Spear, A. Shriraman, H. Hossain, S. Dwarkadas, and M. L. Scott. Alert-on-Update: A Communication Aid for Shared Memory Multiprocessors (poster paper). In *Proc. of the 12th ACM Symp. on Principles and Practice of Parallel Programming*, San Jose, CA, Mar. 2007.

[17] M. F. Spear, A. Shriraman, L. Dalessandro, S. Dwarkadas, and M. L. Scott. Nonblocking Transactions Without Indirection Using Alert-on-Update. In *Proc. of the 19th Annual ACM Symp. on Parallelism in Algorithms and Architectures*, San Diego, CA, June 2007.

[18] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Privatization Techniques for Software Transactional Memory. TR 915, Dept. of Computer Science, Univ. of Rochester, Feb. 2007.

[19] The Rochester Software Transactional Memory Runtime. 2006. www.cs.rochester.edu/research/synchronization/rstm/.