Brief Announcement: Transactions and Privatization in Delaunay Triangulation*

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

Categories and Subject Descriptors:
D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*

General Terms: algorithms, experimentation, measurement, performance

Keywords: synchronization, transactional memory, benchmarks, privatization

1. INTRODUCTION

With the rise of multicore processors, much recent attention has focused on transactional memory (TM). Unfortunately, the field has yet to develop standard benchmarks to capture application characteristics or to facilitate system comparisons. This note describes one candidate benchmark: an implementation of Delaunay triangulation [4]. Source for this benchmark is packaged with Version 3 of the Rochester Software Transactional Memory (RSTM) open-source C++ library [1,9]. 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 ("privatized") work to stitch those regions together. Experiments on multiprocessor and multicore machines confirm good speedup and excellent single-thread performance. They also highlight the cost of extra indirection in the implementation of transactional data: since execution time is dominated by privatized phases, performance is largely insensitive to the overhead of transactions per se, but highly sensitive to any costs imposed on privatized data. Experience with the application-writing process provides strong anecdotal evidence that TM will eventually require language and compiler support.

2. OVERVIEW OF THE BENCHMARK

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. Delaunay triangulation 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 an existing Delaunay mesh as a potential application of transactional memory [8]. Our code addresses the complementary problem of constructing the initial triangulation; we do not yet consider refinement.

We begin by sorting points into geometric regions, one per worker thread. Using Dwyer's refinement [5] of Guibas and Stolfi's divide-and-conquer algorithm [6], each worker then triangulates its own region. Finally, we employ a mix of transactions and thread-local computation to "stitch" the regions together, updating previously chosen triangles when necessary to maintain the circumcircle property.

All told, our application comprises some 3200 lines of C++, in 24 source files. There are three transactional object types and three static occurrences of transactions. The first transactional type represents an edge between two points. The second contains, for a given point, a reference to some adjacent edge, from which others can be found by following neighbor links. The third is used to create links in the chains of a global hash set, used to hold created edges.

The first static transaction protects a call to the `edge` constructor. The second protects the body of a subroutine used when stitching regions together. The third is used to "reconsider" edges that may not satisfy the circumcircle property in light of subsequent region stitching. Together with called routines, these transactions comprise 72, 155, and 214 lines of code, respectively.

3. PERFORMANCE SUMMARY

We have measured the performance of the mesh application on both multiprocessor and multicore machines, using 2 different STM systems and both coarse and fine-grain locks. The original *RSTM* library [9] uses a level of indirection for atomic, nonblocking replacement of object versions. The *redo-lock* library [11] copies new versions back on top of the originals at commit time, avoiding the need for indirection when reading. The *CGL* (coarse-grain-lock) library forces "transactions" to compete for a single, global lock, yielding very low overhead in the uncontended case, but no concurrency. Finally, our *FGL* (fine-grain-lock) results use the CGL back end to avoid both overhead and indirection, and use `#ifdef`s to replace transactions with critical sections that acquire per-point locks.

*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.

PODC'07, August 12–15, 2007, Portland, Oregon, USA.
ACM 978-1-59593-616-5/07/0008.

Performance graphs can be found in our forthcoming paper in the benchmarks track of IISWC 2007 [10]. Briefly, on an 8-core, 32-thread Sun Niagara CMP, maximum speedup is obtained when triangulating about 200,000 points. Here the CGL back end takes 8.9s with one active thread (in which case it reduces to Dwyer's sequential algorithm), 2.4s with 8 active threads (speedup of 3.7), and 2.2s (speedup of 4.1) with 16 active threads. Both absolute times and speedups are better on our 16-processor SunFire SMP: the CGL back end achieves a speedup of 7.7 at 16 threads.

FGL and redo-lock times are similar to those of CGL. The original RSTM back end, however, is roughly 2× slower. This is a direct consequence of its use of indirection: the application is memory bound; private (geometrically partitioned) work consumes well over 90% of total run time; and indirection doubles the cost of accessing transactional objects, even in private code. (The memory-bound nature of the application also accounts for better scaling on the SMP machine, despite comparatively high penalties for coherence misses.) Our results provide perhaps the strongest evidence to date in support of indirection-free STM.

4. PROGRAMMING EXPERIENCE

The RSTM API is based on *smart pointers* [2] and accessor methods ("getters" and "setters"). These provide both initial-access and per-access "hooks" into the run-time system, and serve to catch a wide variety of access errors. Getters take an extra, "validator" argument that allows us to perform post-access consistency checks in redo-lock and other zero-indirection systems.

We have found the API to be a significant improvement over its predecessor, which was borrowed from DSTM [7]. In particular, smart pointers allow us, using generics, to create general-purpose functions that can be used in both transactional and nontransactional contexts. There are 11 such functions in the mesh application. Unfortunately, even the new API is still quite difficult to use [3].

The most obvious problem is simple awkwardness. Accessors are not a natural way to access fields in C++, though they could easily be made so with compiler support, as in C#. Validators, likewise, are pure syntactic clutter. Back ends that copy objects require methods to create, deactivate, and copy clones. Asking the user to write these exposes implementation details that ideally should also be hidden. Our use of generics for transactional/private code sharing is also cumbersome; a similar effect could be achieved with compiler-based function cloning. More problematically, our smart pointers, which come in four different varieties, introduce a level of complexity that seems out of place in a programming model intended to simplify concurrency.

Awkwardness aside, transactional programs that use our API must respect several significant restrictions. One cannot safely escape a transaction in any way other than falling off the end—no `goto`s, no `break`s, no `return`s. More significantly, transactional objects must generally have only trivial constructors and destructors, and only static methods: 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.

Privatization can be achieved via global consensus (as in the barriers of the mesh application) or by using a transaction to remove an object from a shared container. Such "privatizing transactions" must be explicitly labeled in our API, to avoid imposing overheads on non-privatizing code.

Finally, because we are able to instrument only explicitly identified transactional objects, nontransactional (nonshared) objects do not revert their values on abort. This means, among other things, that a transaction can safely read or write a nontransactional variable (assuming, in the latter case, it *always* writes before committing), *but not both.*

Like the sources of awkwardness above, almost all these limitations could be eliminated with appropriate compiler support. We conclude that library-based STM can be a valuable tool for experimentation with back-end implementation techniques, but that the end goal—simple, scalable thread coordination—will require language integration.

5. REFERENCES

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

[2] A. Alexandrescu. Smart Pointers. In *Modern C++ Design: Generic Programming and Design Patterns Applied*, C++ In-Depth Series, chapter 7. Addison Wesley Professional, 2001.

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

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

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

[6] 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.

[7] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software Transactional Memory for Dynamic-sized Data Structures. *22nd ACM Symp. on Principles of Distributed Computing*, July 2003.

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

[9] 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. *ACM SIGPLAN Workshop on Transactional Computing*, June 2006.

[10] M. L. Scott, M. F. Spear, L. Dalessandro, and V. J. Marathe. Delaunay Triangulation with Transactions and Barriers. *IEEE Intl. Symp. on Workload Characterization*, Benchmarks track, Sept. 2007.

[11] M. F. Spear, A. Shriraman, L. Dalessandro, S. Dwarkadas, and M. L. Scott. Nonblocking Transactions Without Indirection Using Alert-on-Update. *19th ACM Symp. on Parallelism in Algorithms and Architectures*, June 2007.