LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-671137

On the Algorithmic Aspects of Using OpenMP Synchronization Mechanisms II: User-Guided Speculative Locks

B. L. Bihari, H. Bae, J. Cownie, M. Klemm, C. Terboven, L. Diachin

May 21, 2015

International Workshop on OpenMP (IWOMP)
Aachen, Germany
September 30, 2015 through October 2, 2015

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

On the Algorithmic Aspects of Using OpenMP* Synchronization Mechanisms II: User-Guided Speculative Locks

Barna L. Bihari[1], Hansang Bae[2], James Cownie[2], Michael Klemm[2], Christian Terboven[3], and Lori Diachin[1]

[1] Lawrence Livermore National Laboratory
[2] Intel Corporation
[3] RWTH Aachen
{bihari1, diachin2}@llnl.gov
{hansang.bae, james.h.cownie, michael.klemm}@intel.com
terboven@itc.rwth-aachen.de

Abstract. In this paper we continue our investigations started in [4] into the effects of using different synchronization mechanisms in OpenMP-threaded iterative mesh optimization algorithms. We port our test code to the Intel® Xeon® processor (former codename "Haswell") by employing a user-guided locking API for OpenMP [1] that provides a general and unified user interface and runtime framework. Since the Intel® Transactional Synchronization Extensions (TSX) provide two different options for speculation—Hardware Lock Elision (HLE) and Restricted Transactional Memory (RTM)—we compare a total of four different run modes: (i) HLE, (ii) RTM, (iii) OpenMP `critical`, and (iv) "unsynchronized". As we did in [4], we find that either speculative execution always outperforms the other two modes in terms of their convergence characteristics. Even with their higher overhead, the TSX options are very competitive when it comes to runtime performance measured with the "time-to-convergence" criterion introduced in [4].

1 Introduction

While transactional memory (TM) has been around for over two decades [7], and despite the many different implementations in both software and hardware that have been offered over the years, it has yet to become a mainstream mechanism of thread synchronization. This is partly due to the fact that any reasonable TM runtime performance strongly depends on specialized hardware, typically achieved through modifications to the L1 and/or L2 caches. So far, two major vendors, IBM* and Intel, provide hardware transactional memory (HTM) on their Blue Gene*/Q (BG/Q) [6] and Intel® Xeon® processors [8], respectively. Despite the availability of HTM platforms, support for TM has not yet made it into the OpenMP API specification [10] (though a hinted lock proposal seems likely to be adopted into OpenMP 4.1). As a consequence, for those intending to

write identical code for a range of systems, the elegance and promised efficiency of TM is compromised by its lack of portability between different platforms.

Over three years ago IBM offered the first production quality TM comprising BG/Q hardware and a TM runtime that delivered features such as multiple roll-back capability and smart heuristics in its software stack, but without burdening the user with and exposing all the details of the underlying algorithms. To use TM, the user only needed to insert directives, call runtime libraries, and tune a few environmental variables.

Intel platforms, on the other hand, offer a more versatile and potentially more efficient TM capability — the (Intel® Transactional Synchronization Extensions (TSX) — with two options: Hardware Lock Elision (HLE) and Restricted Transactional Memory (RTM). However, the system does not come with a complete runtime library that would be readily usable with RTM: the programmer has to provide a non-transactional fall-back path which would have code to eventually acquire the lock that was elided. For most non-expert developers, some of whom maintain or modify legacy code, this is a major drawback, as each transaction (e.g., at least once in each `for` loop with race conditions) would require several lines of coding to be able to use the new hardware features. A more ambitious customer may develop a home-grown library of the different transactional backup paths, but at the cost of some non-trivial time and effort.

An attractive solution to this "missing software dilemma" was introduced in [1] where a library of user-guided locks was proposed as an extension of the well-known OpenMP* API. This API enables the use of HLE and RTM, and also includes other lock hints which can be used as either real alternatives to TSX, or to tune lock-synchronized code by using different lock implementations in the absence of TM. At the user level, the complete package is now comparable to IBM's system, with the added benefit that it is implemented in the open-source LLVM*/Intel runtime, allowing potential modifications by an expert user.

Given this background and the availability of another major vendor's HTM system (i.e., Intel TSX), we continue our experimentation with threaded mesh optimization where potential race conditions exist. Both the hardware and the software are very different from the IBM environment used in our previous work [4], thus our intent was to experimentally test and ideally verify that the principles detailed in [4] still hold. That is, in a shared-memory environment does a threaded algorithm benefit from using TM to dynamically update shared variables that have dependencies? We find the answer through extensive experimentation on the same code used in [4] but run in the Intel environment. The results are quite encouraging: we find similar trends to those of [4] and the same conclusions remain true.

The rest of this paper is organized as follows. Section 2 presents an overview of Intel TSX as well as the user-guided lock library as it relates to our experimentation, while Section 3 gives a brief review of the algorithm and the role we expect TM to play in it. We present our experimental results in Section 4, where we compare three different flavors of synchronizations, as well as a completely unsynchronized version of the code. We also show TM statistics and runtime

performance using the new metric introduced in [4]. We conclude with Section 5 with a brief review of our results.

2 User-guided Locking API with TSX

For completeness, we briefly review Intel TSX and show how it can be used in an OpenMP application that is compiled and linked with the Intel (or LLVM) OpenMP runtime.

2.1 Intel Transactional Synchronization Extensions

Intel Transactional Synchronization Extensions are new capabilities introduced in the current Intel Xeon processor series, which was formerly known as "Haswell". The extensions provide support for transactional execution while using cache-coherence protocols to detect memory access conflicts. On a transactional abort, the architectural state of the processor is restored to that at the start of the transaction (all transactional memory writes are discarded, and register state is restored). At transaction commit all transactional writes become atomically visible to other cores.

By using Intel TSX it is possible to execute multiple dynamic instances of a critical region simultaneously, with the required mutual exclusion enforced by the hardware when conflicting memory accesses occur between these instruction streams. This allows code written with a single coarse lock to behave as if it were implemented through fine-grain reader-writer locks at cache-line granularity.

Intel TSX provides two different interfaces to speculation: Hardware Lock Elision (HLE) and Restricted Transactional Memory (RTM). In both cases, speculation is implemented in the coherence protocol of the processor caches. Because the cache protocol keeps track of the states of individual cache lines, the hardware can use this information to detect conflicting memory accesses and to abort the speculative execution in the cores executing the threads that suffered the conflicts.

HLE is a backwards binary-compatible interface that can be added to an existing lock by tagging the lock and unlock instructions with instruction prefixes that are ignored on processors without TSX support. It requests that the processor executes the protected critical region speculatively. If the speculation fails the processor rolls back, takes the lock "for real", and executes the critical section non-speculatively. HLE preserves all of the semantics of existing locks, such that the lock value read inside the critical section appears to be locked, as it would if there were no speculation.

RTM adds new instructions to put the processor into the speculative execution state and to commit the speculative state. With RTM the user has to provide a non-speculative execution path, since no lock is visible to the hardware. The backup path is triggered whenever the transaction fails and has been aborted by the hardware. The code invoked for this case may choose to retry

speculation, but must ultimately provide a way of executing the critical region non-speculatively.

Since some operations (inter alia any ring transition that enters the kernel) cannot be executed speculatively so will abort, blindly converting all critical regions of a code to use speculative locks will likely be counterproductive. If a large amount of speculative work has been performed before the abort, discarding it becomes prohibitively expensive. Consider a lock used to serialize output, for instance, in which effort may be expended to format data before the write system call is made. If this is done within speculative execution the formatting work will all be lost and will have to be repeated in the non-speculative backup. Therefore, it is hard for the runtime system to determine which locks should be speculated without information from the programmer (who knows what code will be executed in the critical region). This observation motivated our previous proposal for hinted locks [1].

2.2 Using the User-guided Locking API

The Intel/LLVM OpenMP runtime library contains lock implementations that use HLE and RTM. Programmers can select the lock implementation either globally through an environment variable or selectively through an OpenMP extension [1].

The OpenMP extension was proposed to overcome one of the disadvantages of the lock API available in the current OpenMP API specification. The key feature of the new API is to give application writers the capability to choose a hint on a per-lock basis and to pass information to the runtime about the estimated degree of lock contention. As a result, users gain fine-grained control of the OpenMP runtime and can optimize their applications by adjusting the lock implementation for each individual lock if desired.

The proposed API is available in the latest release of Intel's open-source OpenMP runtime [9], and can be enabled at build time. Two additional lock initialization functions are provided to pass extra information to the runtime:

```
void kmp_init_lock_hinted(omp_lock_t*, kmp_lock_hint_t)
void kmp_init_nest_lock_hinted(omp_nest_lock_t*, kmp_lock_hint_t)
```

User-selectable hints (of type `kmp_lock_hint_t`) are defined as an enumeration:

```
typedef enum kmp_lock_hint_t {
  kmp_lock_hint_none,
  kmp_lock_hint_uncontended,      // Optimize for an uncontended lock
  kmp_lock_hint_contended,        // Optimize for a contended lock
  kmp_lock_hint_nonspeculative,   // Do not use hardware speculation
  kmp_lock_hint_speculative,      // Do use HLE hardware speculation
  kmp_lock_hint_adaptive,         // Adaptively use RTM speculation
} kmp_lock_hint_t;
```

```
void example() {
  omp_lock_t lock;
  kmp_init_lock_hinted(&lock, kmp_lock_hint_speculative);

  omp_set_lock(&lock);
  // critical section
  omp_unset_lock(&lock);
}
```

Fig. 1. Using user-guided locks in the Intel OpenMP runtime.

Programmers can use these hints together with the additional initialization routines to tag a lock with a hint. The example code in Figure 1 shows how to use the new API to speculatively execute a critical region that is protected through the OpenMP lock API. After the lock has been initialized and the hint has been bound to the lock, the standard lock routines of OpenMP acquire and release the lock as usual. This simplifies the task of changing the code to use a specific type of lock; only a single initialization of a lock needs to be located and not all the places where the lock is claimed and released. The runtime system uses the hint to change the lock implementation for this particular lock, and, in this case, to select speculation for it.

3 Application of TSX to the Test Code

We will now briefly introduce the algorithm under investigation and then show how Intel TSX can be applied to its efficient thread synchronization.

3.1 A Brief Review of the Algorithm

In our experiments we use the same simple C++ mesh smoothing algorithm of [4] that takes the same initially distorted mesh as input and produces a final converged mesh as output. The main `for` loop accomplishing the averaging operation is symbolically represented by the equation:

$$\mathbf{x}_i^{(n+1)} = \frac{1}{N_i} \sum_{j=1}^{N_i} \mathbf{x}_j^{(m)} \tag{1}$$

where $\mathbf{x}$ is a 2- or 3-D vector, n is the current (old) iteration, $n+1$ is the latest (new) iteration, N_i is the number of connected vertices for grid point i, and m can refer to either n or $n+1$ depending on whether or not that point has already been updated or not. Figure 2 shows the relevant code section, as modified to run with our user-guided locking API on the Intel machines.

In order to avoid write-after-read (WAR) race conditions in Figure 2, the code needs to protect the entire section that includes both "Step 1" and "Step 2", and not just the update operations of "Step 2". The associated `{}` brackets symbolize and the `omp_set_lock` and `omp_unset_lock` routines actually delimit

```
      omp_lock_t lock;
      kmp_init_lock_hinted(&lock, kmp_lock_hint_speculative);
#pragma omp parallel for
      for(int i=0; i < numFreeVerts; i++) {
        int vertexID = freeVertexIDs[i];

// get attached vertex ids:
        std::vector<Point2D*> &myAttachedVertices =
          attachedVertices[vertexID];

        Point2D newX={0.0,0.0};
        size_t numAttachedVertices = myAttachedVertices.size();

        omp_set_lock(&lock)        // start of critical region
        {
// Step 1:  take average of neighbors
          for(int j=0;j<numAttachedVertices;j++){
            newX.x += myAttachedVertices[j]->x;
            newX.y += myAttachedVertices[j]->y;
          }  //  end for (over neighbor vertices)
          newX.x = newX.x/numAttachedVertices;
          newX.y = newX.y/numAttachedVertices;

// Step 2:  update current coordinates:
          x[vertexID].x = newX.x/numAttachedVertices;
          x[vertexID].y = newX.y/numAttachedVertices;
        }
        omp_unset_lock(&lock)     // end of critical region
      }       //  end for (over all vertices)
```

Fig. 2. Simple Laplacian mesh smoothing algorithm.

the entire *transaction*. Note that this is essentially the same simple construct as that of the IBM BG/Q TM runtime, and, apart from the lock initialization, the code of Figure 2 is identical to that of [4].

The OpenMP language offers the `critical` construct for the purposes of simple coarse-grain locking, which would be the obvious choice. However, because OpenMP does not yet support hints to change the behavior of `critical` from regular lock acquire/release to speculation, we emulate this functionality by using the OpenMP lock API and run in two additional modes, called *speculative* or *HLE*, as well as *adaptive* or *RTM* (see Section 2.2).

Note that if we restrict the critical region in Figure 2 to the coordinate update in "Step 2" only, no conflicts exist since each thread only updates the points that it owns. However, the neighboring coordinates might change during "Step 1" so the result could depend significantly on whether old or new data is being used during the update. In other words, we have a write-after-read (WAR) conflict for which we cannot use `#pragma omp atomic`. Thus, we will not compare *HLE* or *RTM* to *atomic* as we did in some of our prior experiments in [5] and [11].

3.2 The Role of TSX

As seen in [4], the BG/Q TM system made a substantial difference to the quality of the solution we obtained because of the way the updates were done dynamically. With Intel's TSX system, we also expect to see a difference, especially since now we actually have two options to properly synchronize the critical code section of Figure 2.

HLE When HLE is requested, the system tries to elide the lock and execute the critical region without requiring any communication through the lock. However, the TSX-enabled hardware is able to detect conflicting operations in hardware. Should such a conflict occur, the system will execute the protected code section non-transactionally, that is, without elision and with an acquire/release cycle for the lock. If such a *retry* happens (often referred to as a "rollback") the code will essentially be serialized and forward progress is ensured, albeit with lower efficiency. In terms of the usual TM terminology, only one rollback is allowed with HLE. Once that happens, HLE becomes equivalent to `omp critical`.

RTM With RTM, the lock protecting the transactional region is also elided at first. Upon a conflict—detected by the hardware as with HLE—the transaction *aborts* and reverts to the fallback instruction address provided by the programmer. This provides a great deal of flexibility to handle the abort, as well as the implied responsibility to provide a code sequence that guarantees forward progress. In our work, we took advantage of the OpenMP extensions of [1] in order to benefit from the hardware-assisted RTM. As mentioned in Section 2.2, and further elaborated on in [1], the basic system-provided RTM software interface was built into the "adaptive" option. The adaptive lock has some additional features, such as the environmental variable `KMP_ADAPTIVE_LOCK_PROPS = 'M,N'` which allows `M` number of maximum retries, and a "maximum badness" of `N`. The latter parameter is used to guess or speculate whether or not to even try RTM since transactions that have previously failed may only have a slight chance of succeeding in the future and may not be worth the higher overhead of new speculative attempts, hence the name "adaptive". The adaptive lock also collects some statistics related to the successes and different failure modes of the speculation. Because of all these additional features, the "adaptive" lock is naturally more expensive than its "speculative" counterpart, but it can potentially yield a higher quality result as multiple rollbacks typically mean multiple updates, therefore newer and more accurate neighbor coordinates for the the current mesh point.

As was the case with BG/Q [4], the way the hardware detects memory conflicts and how the software handles them will significantly affect the overall behavior of algorithm itself. In our experiments, for example, there were many instances of multiple retries on the same location, where RTM turned out to be quite useful, while for others HLE with its low overhead was sufficient to handle a low conflict-probability scenario. Indeed, the choice for handling of the critical section will have a profound influence on the algorithm and the solution quality

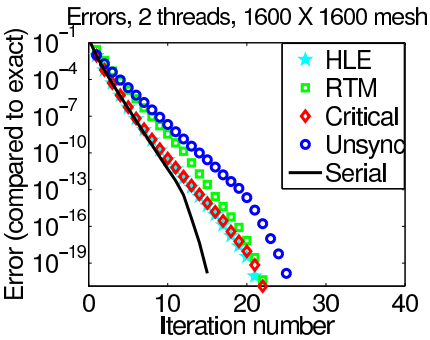

Fig. 3. Convergence on 2 threads.

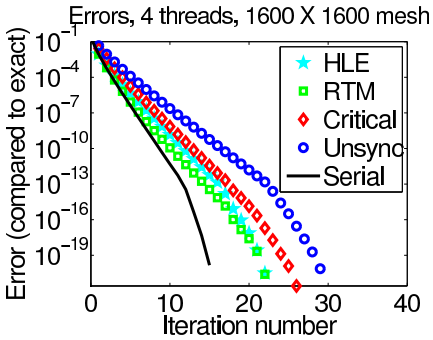

Fig. 4. Convergence on 4 threads.

itself. The right decision of which synchronization method to use will likely be made by weighing quality against cost.

4 Experimental Results

We now present some computational experiments using the algorithm and test mesh described above. As in [4], the original Cartesian mesh cells were 1×1 non-dimensional units in size, which are then disturbed by a random factor in the range of $(-0.5, 0.5)$, the largest distortion that still guarantees no initial mesh-line cross-overs. During the runs we vary the number of threads (1, 2, 4, 8, 16, 32, and 64) and use the three coarse-grain synchronization modes *HLE*, *RTM*, and *critical* as well as *unsync*.

For the evaluation, we use Intel® C++ Compiler 16.0.0.056 Beta running on an Intel® Xeon® E5-2698v3 (2.3GHz, 64GB DDR3 at 2133 MHz, TSX enabled). Our system runs Red Hat Enterprise Linux Server release 6.6 (Santiago), with kernel version kernel-2.6.32-504.16.2.

We now analyze the convergence, and some TM characteristics of the results.

4.1 Convergence

Convergence is measured in terms of the l_1 norm of the difference between the current iterate and the exact solution which is known. The *error* $e^{(n)}$ at time n is defined by:

$$e^{(n)} = \frac{1}{M} \sum_{i=1}^{M} |\mathbf{x}_i^{(exact)} - \mathbf{x}_i^{(n)}| \tag{2}$$

where (n) denotes the current iteration counter, M is the number of interior (non-boundary) vertices, and $\mathbf{x}_i$ is the coordinate vector of point i. The $(exact)$ supersript denotes the solution of the mesh optimization problem, which is known for our simple problem.

We use the serial solution (which is the same as executing a single thread) as our reference solution in all plots. Figure 3 shows that convergence rates with 2

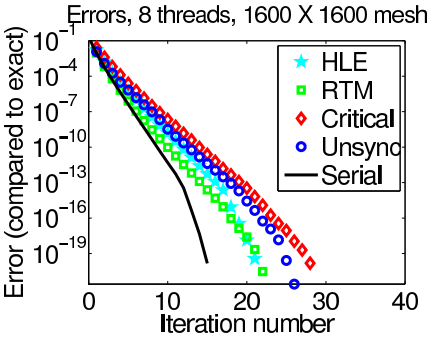

Fig. 5. Convergence on 8 threads.

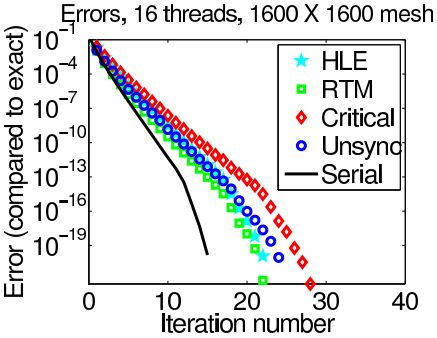

Fig. 6. Convergence on 16 threads.

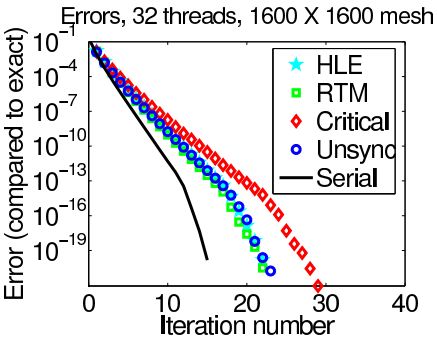

Fig. 7. Convergence on 32 threads.

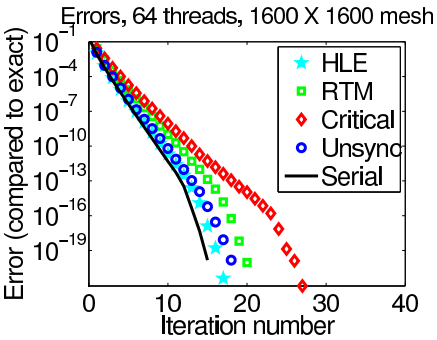

Fig. 8. Convergence on 64 threads.

threads exhibit little difference between *HLE* and *critical*, which are close to each other, while *unsync* is the slowest to converge, *RTM* being in between. However, Figures 4, 5, and 6 show *RTM* overtaking the others, with *critical* performing the worst on 8 and 16 threads. As explained in [4], the serial version will have the least error as the points will always get the most recent update possible. As we increase the thread count to 32 (Figure 7), *HLE* gets closer to *RTM*, and at 64 threads (Figure 8) *HLE* actually performs the best.

4.2 Transactional Memory Statistics

Results for threaded code with memory conflicts, with or without TM, are always timing-sensitive. Therefore, we typically repeat the same outer loop dozens or hundreds of times — as we did in e.g. [2, 5, 3] — and run it well past convergence. We thus obtain some statistical averages as well as the iteration-dependent variation of the TM statistics. The statistics for the *RTM* case are obtained from the `__kmp_print_speculative_stats()` utility provided by the user-guided lock runtime. At each iteration we observe the number of retries ("soft failures") that have occurred, the grand total at the end of the run, as well as several other diagnostic numbers such as "non-speculative acquires" (i.e. the number of times the code was serialized) and "hard failures"; the difference of these two numbers is the number of times speculation was not even tried by the "adaptive" lock.

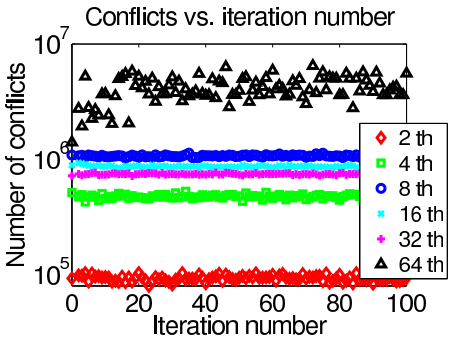

Fig. 9. RTM retries per iteration on 1600×1600 mesh on 2 through 64 threads.

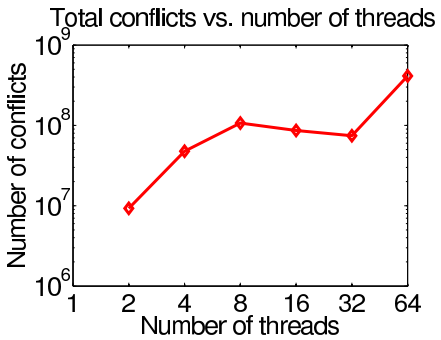

Fig. 10. Total number of RTM retries on 1600×1600 mesh on 2 through 64 threads.

Plotting the number of retries in Figure 9, we see a relatively large (one order of magnitude) spread in the number of conflicts from iteration to iteration on 64 threads, but small oscillations on low to moderate thread counts (2 to 32 threads). This may be due to the fact that the Haswell node this was run on has 32 cores, and at 64 threads hyperthreading may have created some anomalies. The iteration-to-iteration volatility is certainly much smaller than that observed on the BG/Q (see [4]).

Figure 10 shows the total number of conflicts over 100 iterations for all thread counts. We observe a monotonic increase in the number of total conflicts as the number of threads increases to 8, with some drop at 16 and 32 threads, and another increase at 64 threads.

4.3 Performance Measurement

Finally, we use the new performance measure of [4] to define "run time per quality" $t_q^{(n)}$ (loosely speaking "time to convergence"), assuming $q^{(n)}$ is the inverse of the error $e^{(n)}$ defined by equation (2):

$$t_q^{(n)} = \frac{t^{(n)}}{q^{(n)}} = t^{(n)} e^{(n)}. \tag{3}$$

We use this measure to gauge whether one or both of the TSX options provides some benefit when their well-known overhead is balanced against their superior convergence behavior. For this study, we again run the code in four different modes (*HLE*, *RTM*, *critical*, and *unsync*), and to iteration $n = 17$. The inclusion of *unsync* in the study is for information purposes only since strictly speaking it is non-conforming OpenMP code due to the data races.

We find that both *HLE* and *RTM* outperform *critical* by one or two orders of magnitude on all except 2 threads (Figure 11). Moreover, *RTM* yields significantly better time-per-quality than *unsync* on 2, 4 and 8 threads, and is the same on 16 and 32 threads. On the other hand, *HLE* outperforms *unsync* on 2, 4, and 64 threads. In essence, one of the TSX options meets or beats *unsync* at *every* thread count, with *HLE* being the overall best at 64 threads. Indeed,

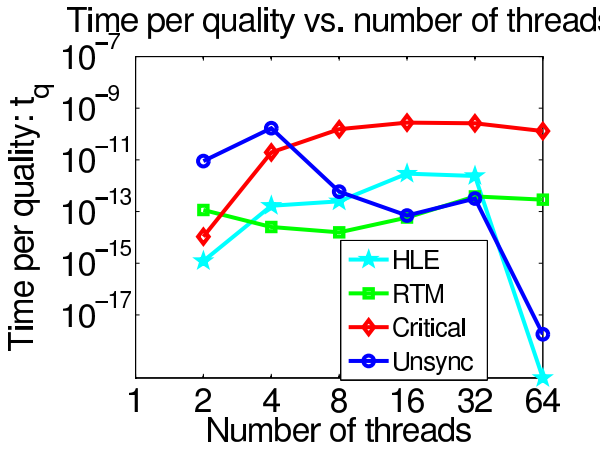

Fig. 11. Time per quality on a 1600×1600 mesh on up to 64 threads.

at 64 threads there are an order of magnitude more retries than on all other thread counts (see Figures 9 and 10). These retries come with all their overhead costs, providing at least a partial explanation for the difference between *HLE* and *RTM* at 64 threads. As alluded to earlier, hyperthreading may be another cause of this outlier behavior.

5 Conclusion

We continued studying different OpenMP synchronization constructs using the iterative Laplacian mesh-smoothing algorithm of [4] and the user-guided locking API of [1] on the Intel® Xeon® processor. We again concentrated on evaluating the specialized TSX hardware offered by this platform as well as the open source library developed as a user interface to it. Using the new figure of merit introduced in [4], we concluded that not only both TSX options compared favorably to OpenMP `critical`—an expected result—but also that they outperformed even a completely unsynchronized version of the code. In all of our tests the best performance is provided by one of the two speculative execution options.

Acknowledgments

The authors thank Trent E. D'Hooge of Livermore Computing for his assistance with our inquiries and in accommodating our runs on the local compute nodes.

Intel and Xeon are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

* Other names and brands are the property of their respective owners.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully

evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to `http://www.intel.com/performance`.

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

References

1. H. Bae, J. Cownie, M. Klemm, and C. Terboven. A User-Guided Locking API for the OpenMP Application Program Interface. In *IWOMP 2014 Conference Proceedings*, pages 173–186, Salvador, Brazil, September 2014. LNCS 8766.
2. B. L. Bihari. Applicability of transactional memory to modern codes. In *International Conference on Numerical Analysis and Applied Mathematics 2010 (ICNAAM 2010) Conference Proceedings*, pages 1764–1767, Rodos, Greece, 2010. APS.
3. B. L. Bihari. Transactional Memory for Unstructured Mesh Simulations. *J. Sci. Comput.*, 54:311–332, 2012.
4. B. L. Bihari, M. Wong, B. R. de Supinski, and L. Diachin. On the Algorithmic Aspects of Using OpenMP Synchronization Mechanisms II: User-Guided Speculative Locks. In *IWOMP 2014 Conference Proceedings*, pages 115–129, Salvador, Brazil, September 2014. LNCS 8766.
5. B. L. Bihari, M. Wong, A. Wang, B. R. de Supinski, and W. Chen. A Case for Including Transactions in OpenMP II: Hardware Transactional Memory. In *IWOMP 2012 Conference Proceedings*, pages 44–58, Rome, Italy, June 2012. LNCS 7312.
6. R.A. Haring, M. Ohmacht, T.W. Fox, M.K. Gschwind, D.L. Satterfield, K. Sugavanam, P.W. Coteus, P. Heidelberger, M.A. Blumrich, R.W. Wisniewski, A. Gara, G.L.-T. Chiu, P.A. Boyle, N.H. Christ, and C. Kim. The IBM Blue Gene/Q Compute Chip. *IEEE Micro*, 32(2):48–60, 2013.
7. M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. *SIGARCH Comput. Archit. News*, 51(2):289–300, 1993.
8. Intel Corporation. Intel® Architecture Instruction Set Extensions Programming Reference, 2012. Document number 319433-014.
9. Intel Corporation. Intel® OpenMP* Runtime Library, 2015. `http://www.openmprtl.org/`.
10. OpenMP Architecture Review Board. OpenMP Application Program Interface, Version 4.0, 2013. `http://www.openmp.org/`.
11. M. Schindewolf, J. Gyllenhaal, B.L. Bihari, A. Wang, M. Schulz, and W. Karl. What Scientific Applications Can Benefit from Hardware Transacional Memory? . In *Int. Conf. for High Perf. Computing, Networking, Storage and Analysis (SC12)*, 2012.

Prepared by LLNL under Contract DE-AC52-07NA27344.