LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-694162

Transactional Memory for Algebraic Multigrid Smoothers

B. L. Bihari, U. M. Yang, M. Wong, B. R. de Supinski

June 3, 2016

12th International Workshop on OpenMP
Nara, Japan
October 5, 2016 through October 8, 2016

Disclaimer

Transactional Memory for Algebraic Multigrid Smoothers

Barna L. Bihari[1], Ulrike M. Yang[1], Michael Wong[2] and Bronis R. de Supinski [1]

[1] Lawrence Livermore National Laboratory
[2] Codeplay Software
{bihari1, yang11, desupinski1}@llnl.gov
{fraggamuffin}@gmail.com

Abstract. This paper extends our early investigations in which we compared transactional memory to traditional OpenMP synchronization mechanisms [7, 8]. We study similar issues for algebraic multigrid (AMG) smoothers in hypre [16], a mature and widely used production-quality linear solver library. We compare the transactional version of the Gauss-Seidel AMG smoother to an `omp critical` version and the default *hybrid* Gauss-Seidel smoother, as well as the l_1 variations of both Gauss-Seidel and Jacobi smoothers. Importantly, we present results for real-life 2-D and 3-D problems discretized by the finite element method that demonstrate the TM option outperforms the existing methods, often by orders of magnitude, in terms of residual behavior and run time.

1 Introduction

Transactional memory (TM) is widely recognized as an easy-to-use shared memory synchronization mechanism. However, the next version of the OpenMP specification does not currently seem likely to support it despite previous proposals to do so [9, 31], The lack of interest stems partly from limited availability of hardware support but perhaps even more so from the lack of demonstrations that it offers reasonable performance for production applications.

Most TM studies focus on the design of TM mechanisms and their optimization. Nearly all only consider benchmarks or kernels, particularly when applying TM to scientific computing [3, 6–9, 26, 29, 31, 33]. For example, our prior work used a small example code to explore TM performance when application semantics allow a degree of nondeterminism. In this work, we consider similar issues for a production code base with over two decades of development and widespread use: the *hypre* linear solver library [16]. Our results demonstrate that TM not only can simplify development but also provide significant performance benefits for mature applications. Overall, we show that TM outperforms alternative synchronization mechanisms by up to two orders of magnitude in *hypre*'s algebraic multigrid (AMG) smoother for 2-D and 3-D problems. These results indicate that the OpenMP arsenal of optimization techniques should include TM.

The paper is structured as follows. Section 2 covers related work, focused primarily on the current state of the art of TM. Section 3 provides a brief overview

of the AMG method and then details how we use TM to simplify its implementation and to improve its performance. Section 4 compares experimental results for five AMG smoothers, including two OpenMP synchronization options that we implement for this work. In Section 5 we conclude with a brief review of our results.

2 Transactional Memory State-of-the-Art

Many studies have explored TM programmability and performance compared to locks for a range of benchmarks and kernels including Delaunay triangulation [27], minimum spanning forest of sparse graphs [17], and Lee's routing algorithm [2], among others [15, 23, 18]. QuakeTM [13], Atomic Quake [34] (using a lock-based version), and SynQuake [20], which use TM to implement the Quake game server [1], provide the most significant application studies. These studies demonstrate that TM can improve performance as well as programmability for production multi-player games; our work provides similar proof for a production scientific computing application.

Other studies have investigated the usability of TM. For example, Rossbach et al. found that programs using fine-grain locking were more likely to contain errors than those using coarse grain locks or TM [25]. Pankratius and Adl-Tabatabai concluded that TM is not a panacea for parallel programming: it still requires good programmers although it has promise compared to fine-grain locking for large and complex parallel programming tasks [24]. While we are not specifically studying programmability, we have found that TM simplifies writing data-race free programs without sacrificing performance.

Substantial recent effort has explored mechanisms to add TM support to C++ [14]. This activity includes participation from HP, IBM, Intel, Oracle and RedHat and has led the C++ Standards Committee to form Study Group 5: Transactional Memory (SG5, for short) [32]. SG5 is now working with the C++ Standards Committee with the goal of creating an acceptable set of transactional language constructs for Standard C++. We have proposed OpenMP pragmas and semantics [30] that are closely related to a recent C++ SG5 proposal [19] and would simplify interoperability with a likely addition to the C++ Standard. This direction within the C++ community indicates that OpenMP should strongly consider adding TM support, as we advocate in this paper.

In addition to our prior work [6–9, 26, 31], others have proposed adding transactional memory support to OpenMP [22, 4]. These efforts have concluded that TM is well suited to a directive-based approach since transactions are naturally represented as sequential code blocks. As already discussed, our work has found that TM can provide competitive performance for toy benchmarks that represent scientific computing patterns found in mesh-based algorithms [6–9]. Our current work shows that production applications can benefit even more.

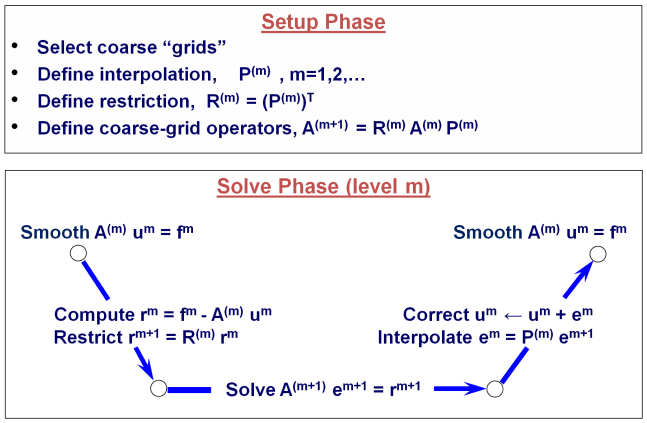

Fig. 1. AMG building blocks.

3 Applying Transactional Memory to the AMG Smoother

3.1 Brief Review of Algebraic Multigrid Methods

Algebraic multigrid (AMG) methods [28] are well-suited for large-scale scientific applications because they are algorithmically scalable: they solve a sparse linear system $Au = f$ with n unknowns with $O(n)$ computations. They obtain this optimality by reducing error using two separate operations: smoothing and coarse grid correction between successively coarser levels. Coarse grid correction involves restriction and prolongation or interpolation operators between levels. The restriction is generally defined as the transpose of the prolongation.

Smoothers must reduce errors in the directions of eigenvectors. These "smooth errors"' can be characterized with $Ae \approx 0$. For an effective AMG method the prolongation operator $P^{(m)}$ that interpolates the approximate error e^{m+1} from the $m+1$-st level to the mth level must be defined so that the smooth errors on the mth level are approximately in the range of $P^{(m)}$. Simple point-wise smoothers, such as Jacobi or Gauss-Seidel, or their combinations, reduce smooth errors associated with large eigenvalues rapidly. Reducing errors associated with small eigenvalues can be more time consuming. Algebraic multigrid (AMG) does not require an explicit grid. Instead, coarse grid selection and the generation of interpolation and restriction operators only depend on the matrix coefficients.

AMG consists of two phases: setup and solve, as shown in Fig. 1. The primary computational kernels in the setup phase are the selection of the variables for the coarser grids, the definition of the interpolation ($P^{(m)}$) and restriction ($R^{(m)}$) operators, and the creation of the coarse grid matrix operator $A^{(m+1)}$ for $m = 0, 1, ..., L$, where $L+1$ is the number of levels. The variables for the $(m+1)$st level as well as the entries in $P^{(m)}$ and $R^{(m)}$ are determined by making use of the coefficients of $A^{(m)}$. These algorithms can be quite complicated.

In the solve phase, a smoother is applied on each level $m = 0, ..., k-1$, and then the residual r^m is transferred to the next coarser grid, where the process

continues. On the coarsest level, the linear system $A^{(k)}e^k = r^k$ is solved by Gaussian elimination. The error e^k is then interpolated to the next finer grid, followed by relaxation, which continues to the finest grid. Figure 1 describes the m-th level of the solve phase. The process of starting on the fine grid, restricting to the coarse grid, and interpolating back to fine grid again is called a V-cycle.

The solve phase primarily consists of a matrix-vector multiplication (MatVec) and the smoother. The classical smoother used for algebraic multigrid is Gauss-Seidel, which is highly sequential. Therefore AMG often uses a parallel variant, called hybrid Gauss-Seidel (*HGS*), which can be viewed as an inexact block-diagonal (Jacobi) smoother with Gauss-Seidel sweeps inside each process. In other words, we use a sequential Gauss-Seidel algorithm locally on each process, with delayed updates across processes. One HGS sweep is similar to a MatVec.

For our experiments, we use the parallel AMG code BoomerAMG as a pre-conditioner to a GMRES solver, both contained in the *hypre* software library [16]. We use HMIS coarsening [11] with extended+i interpolation [10]. Sparse matrices in BoomerAMG are stored in the ParCSR matrix data structure, in which the matrix A is partitioned by rows into matrices A_k, $k = 0, \ldots, p-1$, where p is the number of MPI processes or OpenMP threads. A_k is stored locally as two matrices in sequential CSR (compressed sparse row) format, D_k and O_k. D_k contains all entries in A_k for which column indices point to rows stored on process k. O_k contains the remaining entries, which have column indices that point to rows stored on other processes. Matrix-vector multiplication Ax involves computing $A_k x = D_k x^D + O_k x^O$ on each process, where x^D is the portion of x stored locally and x^O is the portion that needs to be sent by other processes. Both the MPI- and OpenMP-based parallelism for Gauss-Seidel relaxation is accomplished in the same "hybrid" fashion: on node- or thread-boundaries the previous iterate's information is passed and used, while within each node or thread a full Gauss-Seidel iteration is performed [12].

3.2 TM-Assisted Error-Smoothing in AMG

We now describe how we use TM to simplify the implementation of multi-grid smoothing and how our approach can improve its convergence. Multigrid smoothing is symbolically represented by the equation:

$$u_i^{(n+1)} = \frac{1}{A_{ii}} \sum_{j=1}^{N_i} A_{ij} u_j^{(l)} \tag{1}$$

where u is a scalar, A_{ij} represent the non-zero components of row i in matrix A, n is the current (old) iteration (so $n+1$ is the next), N_i is the number of entries in row i, and l is either n or $n+1$, depending on whether that entry has already been updated. Since $u^{(l)}$ can imply dependences within the straightforward loop-based calculation of $u^{(n+1)}$, threading the computation over index i is non-trivial. However, we can apply the TM-based threading approach that we previously used for mesh smoothing operations [7, 8],

```
.........
#pragma omp parallel for private(i,ii,jj,res) HYPRE_SMP_SCHEDULE
  for (i = 0; i < n; i++)
  {             // start of for-loop threaded over rows
    if (cf_marker[i] == relax_points &&
        A_diag_data[A_diag_i[i]]  != zero)
    {           // start of if-statement
      res = f_data[i];
      for (jj = A_offd_i[i]; jj < A_offd_i[i+1]; jj++)
      {
        ii = A_offd_j[jj];
        res -= A_offd_data[jj] * Vext_data[ii];
      }
#pragma tm_atomic
      {         // start of transaction
// Step 1:  Take weighted-average.
      for (jj = A_diag_i[i]+1; jj < A_diag_i[i+1]; jj++)
        {       // start of averaging for-loop
          ii = A_diag_j[jj];
          res -= A_diag_data[jj] * u_data[ii];
        }       // end of averaging for-loop
// Step 2:  Update current u.
        u_data[i] = res / A_diag_data[A_diag_i[i]];
      }         // end of transaction
    }           // end of if-statement
  }             // end of for-loop threaded over rows
.........
```

Fig. 2. Transactional version of actual code section from *Hypre*.

Figure 2 shows the relevant *hypre* code section. We add exactly *one* OpenMP directive. In essence, this becomes a simplified version of *HGS* where the inter-thread Jacobi update is eliminated, or rather, replaced by TM. We have write-after-read (WAR) race conditions as Figure 2 shows. Because the value of the other elements of `u_data` might change during "Step 1", the resulting average could depend significantly on whether the update in "Step 2" uses old or new data. Therefore, the transaction must protect the entire code section that includes both steps, and not just the update operations of `u_data` in "Step 2". The latter update, by itself, is embarrasingly parallel. In other words, we have a write-after-read (WAR) conflict for which we cannot use `#pragma omp atomic`.

4 Experimental Results

Our experiments evaluate the convergence rate and run time of the algorithm in Section 3.2 for two finite element discretizations in 2-D and 3-D [5]. All results use the modification of the BoomerAMG branch of *hypre*, HMIS coarsening [11] with extended+i interpolation [10] and AMG-preconditioned GMRES as the solver. Several existing smoother options provide state-of-the-art comparisons [16].

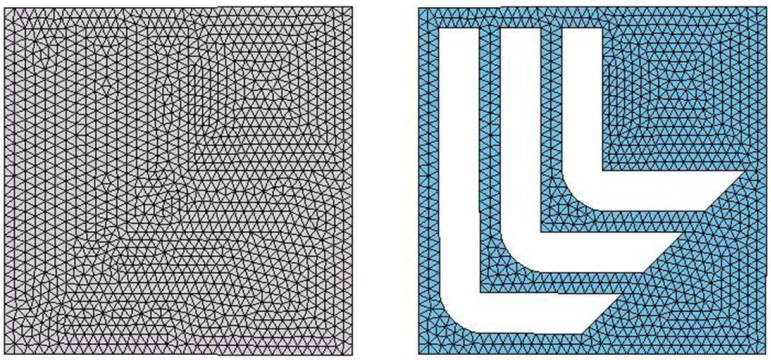

Fig. 3. Original unstructured mesh and cut-outs for multi-material 2-D LLNL mesh.

We stop calculations after a preset iteration count and use the residuals to measure quality instead of allowing the run to converge to a tolerance value. We do not use iteration count for the latter since our metric provides much more accurate timings per unit reduction in residuals (i.e., "quality") comparison. We used the built-in *hypre* timers associated with the solve phase of Figure 1 . We run our experiments on IBM Blue Gene/Q systems using its hardware transactional memory (HTM) support with `TM_ENABLE_INTERRUPT_ON_CONFLICT = YES` and `TM_MAX_NUM_ROLLBACKS=10`.

4.1 Problem Descriptions

Both test problems solve the scalar diffusion problem described by (see also [5]):

$$-\nabla\dot{(}a(x,y,z)\nabla u) = f \tag{2}$$

It is discretized on unstructured meshes using the MFEM finite element package [21], which results in matrices that are not diagonally dominant. Both cases use homogeneous Dirichlet boundary conditions. In addition to our current detailed tests, we can also qualitatively compare with results from a prior study [5].

2-D LLNL is a two-dimensional problem on a unit-square discretized into triangular elements with four material subdomains that form the LLNL logo (Figure 3). The coefficient $a(x, y)$ is 1 in the three Ls and 10^{-3} in the outer domain.

3-D Sphere is a three-dimensional sphere discretized with trilinear hexahedral finite elements and two material subdomains that are placed in arbitrary locations. Their material coefficients $a(x, y, z)$ are 1 and 10^3 (see Figure 4).

4.2 Convergence

We measure convergence as the l_2 norm of the residuals after each GMRES iteration. In addition to our TM algorithm, we run both problems using hybrid Gauss-Seidel (*HGS*), l_1 Gauss-Seidel and l_1 Jacobi, denoted by *L1-GS* and

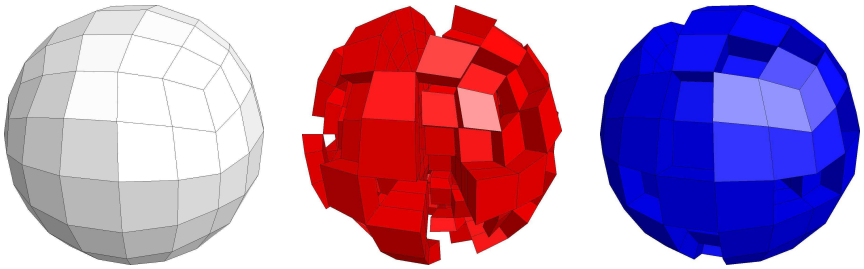

Fig. 4. Coarse version of 3-D sphere mesh and its two material subdomains in color.

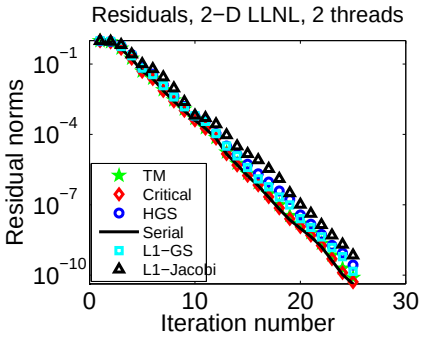

Fig. 5. Convergence on 2 threads, LLNL.

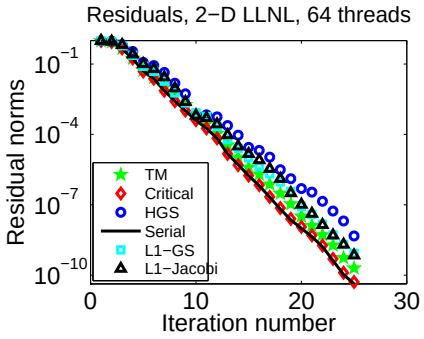

Fig. 6. Convergence on 64 threads, LLNL.

L1-Jacobi, respectively. *HGS* is the default option in *hypre* used for problems parallelized both via MPI and OpenMP, while the latter two appeared to remedy some shortcomings of *HGS* observed in prior work [5]. They represent a fair comparison as they are often used as AMG smoothers in conjunction with GMRES to solve non-symmetric problems. We also run a version of the algorithm in Section 3.2 that replaces `#pragma tm_atomic` with the OpenMP standard `#pragma omp critical` (called *critical*), which is the only OpenMP synchronization mechanism that is comparable to TM.

2-D LLNL: We stop this calculation at 25 iterations. The serial solution, which should be optimal in some sense, serves as our reference in all plots. Figure 5 shows that differences in convergence of the six different smoothers already emerge with 2 threads. As expected, *L1-Jacobi* is the slowest to converge, while *TM* and *critical* are almost indistinguishable from *serial*, with *L1-GS* and *HGS* being somewhere in between. 64 threads results in a larger spread between the different methods and *HGS* becomes the slowest to converge, with *L1-GS* approaching *L1-Jacobi*, which, of course, is invariant to thread count (Figure 6). As initial evidence that synchronization matters as the thread count increases, *TM* and *critical* are the closest to *serial*, the latter being the overall fastest converging. We observe two orders of magnitude difference in the residuals of *HGS* at 25 iterations and *serial* with 64 threads, which is consistent with prior results [5]. The convergence plots for other thread counts are in between the 2 and 64 ones.

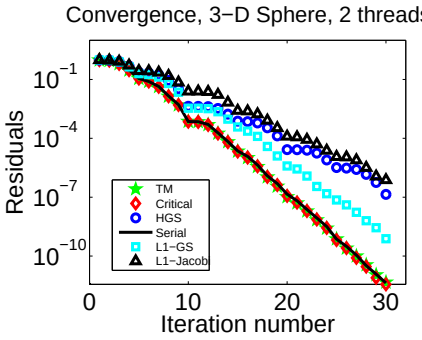

Fig. 7. Convergence, 2 threads, sphere.

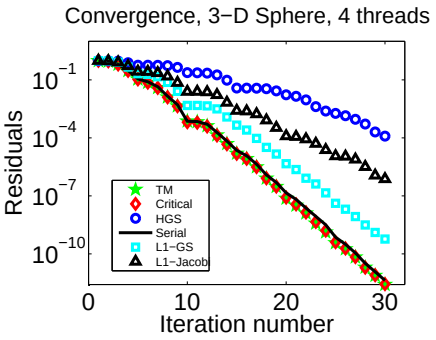

Fig. 8. Convergence, 4 threads, sphere.

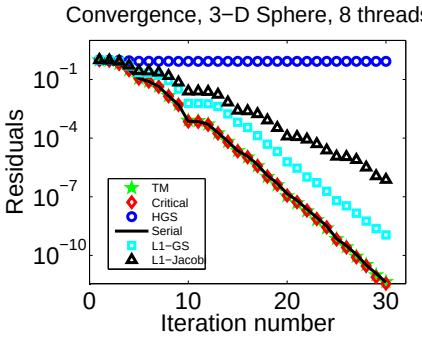

Fig. 9. Convergence on 8 threads, sphere.

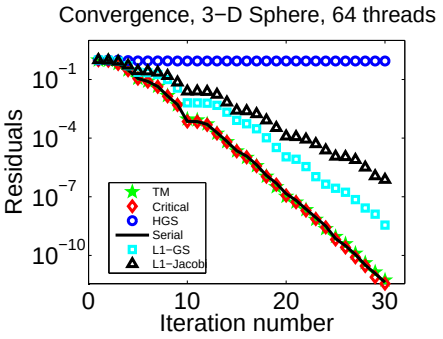

Fig. 10. Convergence, 64 threads, sphere.

3-D Sphere: We stop the calculation at 30 iterations. This problem exhibits a much more dramatic change with increasing thread count so we show plots for 2, 4, 8 and 64 threads. While on one thread all of *TM*, *critical*, and *HGS* are identical to *serial* (not shown), 2 threads already results in a substantial difference between *HGS* and those other options. Its convergence deteriorates by orders of magnitude, approaching that of *L1-Jacobi* (Figure 7). *HGS* convergence continues to deteriorate as we increase the thread count to 4 (Figure 8) and it stops converging altogether for 8 threads (Figure 9); the 64-thread case is similar (Figure 10).

For all thread counts the convergence of *L1-GS* remains in between *serial* and *L1-Jacobi*, with small deteriorations with increasing number of threads, as Figures 7- 10 show. The performance of *TM* and *critical* remain the same in all cases: very close to that of the serial version, showing the value of synchronization in these iterative algorithms. The most surprizing result is the significant improvement of *HGS* from the addition of TM, which, in effect, made the difference between non-convergence and convergence.

4.3 Transactional Memory Statistics

The rest of this section focuses on the 3-D sphere problem since it is the larger and therefore more challenging one. An important issue for TM algorithms is how

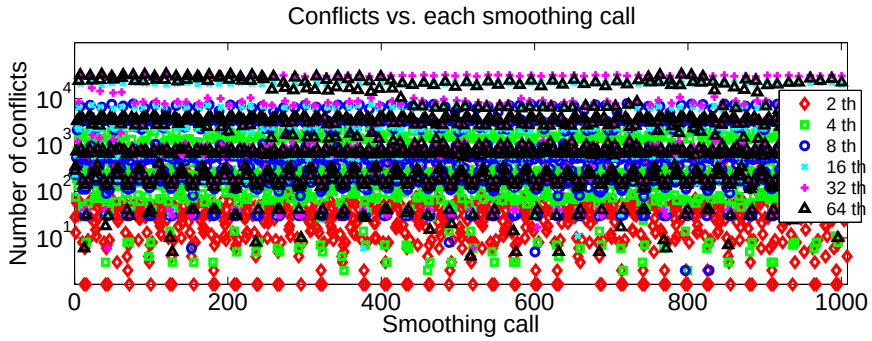

Fig. 11. Rollbacks per call to the smoother for sphere, on 2 through 64 threads.

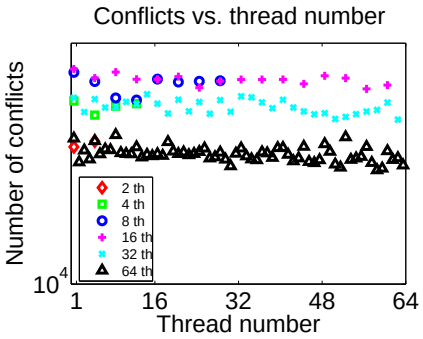

Fig. 12. Rollbacks per thread number for sphere, on 2 through 64 threads.

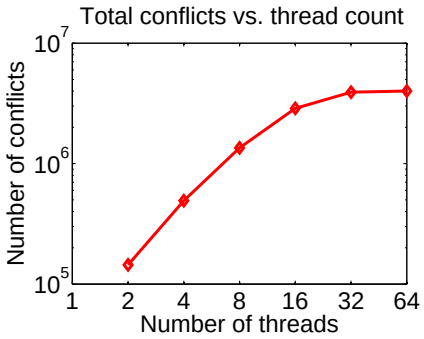

Fig. 13. Total number of rollbacks per thread count for sphere.

many times the TM subsystem rolled back transactions. At the end of each call to the smoother we use the `tm_print_all_stats()` utility to output the TM-report. Figure 11 shows that these numbers exhibit an uneven but regular pattern for each thread count. Recall from Section 3.1 that smoothing is invoked on each level of matrix resolution, and, therefore, both the number of non-zero entries that are averaged and the number of entries that are updated are different on each level. Thus, the number of transactions and the number of rollbacks varies with the level m. While Figure 11 is extremely busy, one can still distinguish a cyclic pattern, for each thread count that corresponds to AMG's cyclic nature described in Section 3.1.

Figure 12 shows that the per-thread breakdown of the total number of conflicts has an even patter between threads although the number of rollbacks is highly dependent on the order of the original finite element mesh and it can vary greatly between different threads. The total number of rollbacks versus thread count curve shows a monotonic increase that tapers off at 32 threads (Figure 13).

4.4 Timed Performance

Strong Scaling: Given the limited memory capacity of a BG/Q node and our emphasis on OpenMP threading, we show results for strong scaling only.

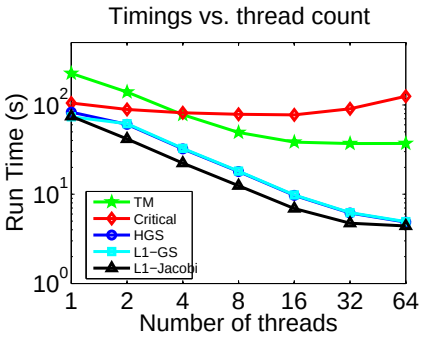

Fig. 14. Strong scaling for sphere, on 1 through 64 threads.

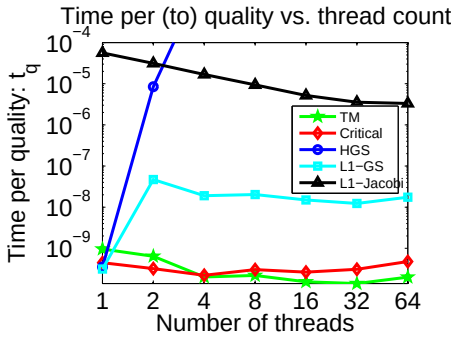

Fig. 15. Time per quality for sphere, on 1 through 64 threads.

Figure 14 shows relatively linear scaling for *TM* on up to 16 threads, for *L1-Jacobi* on up 32 threads, and for *L1-GS* and *HGS* (virtually on top of each other) on all thread counts; *critical*, on the other hand, does not scale at all, an expected result. However, the same figure also shows that, on a per-run (30 iterations) basis, *critical* is faster than *TM* on 1- or 2 threads, and *L1-Jacobi*, *L1-GS* and *HGS* are up to an order-of-magnitude faster than *TM* on all thread counts. However, these raw timings do not consider quality of solution.

Time-to-quality: We slightly modify our performance measure introduced in [8] to define "run time per quality" $t_q^{(n)}$, (assuming $q^{(n)}$ is the inverse of the l_2-norm of the residual $r^{(n)}$) at each iteration n:

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

The results in Figure 15 for this performance measure show several orders of magnitude difference between the various smoothing methods. As already indicated by Figures 9-10, *HGS* does not converge past four threads, thus its $t_q^{(n)} \to \infty$, despite its near perfect scaling shown in Figure 14. On the other hand, *critical* has strong performers in Figure 15 even though it does not scale at all. The best performance is offered by *TM* on 8 through 64 threads, with its 32-thread $t_q^{(n)}$ being the overall best performer for this problem.

As with the simpler mesh optimization problem [7, 8], even though some methods may be more expensive on a per-iteration basis, if they converge faster they may end up being more efficient overall since fewer iterations are needed to achieve convergence, and the CPU-time to solution can actually be shorter.

5 Concluding Remarks

Building upon our prior work [7, 8, 5], we studied two different OpenMP synchronization constructs in the context of iterative AMG smoothers with emphasis on transactional memory as a promising mechanism to resolve write-after-read

memory conflicts. On each thread count we conducted detailed studies of the behavior of residuals, TM statistics, strong scaling, as well the overall "price/performance" of each method considered. Using our figure of merit [8], we concluded that TM outperformed the alternatives currently offered in BoomerAMG and *hypre* often by orders of magnitude. In all of our tests, OpenMP synchronization made a significant difference in reducing the residuals for a given CPU time. Surprisingly, OpenMP critical performed well under this metric.

In our future work, we will use TM for other GS-flavored methods, such as l_1 GS and l_1 symmetric GS. These implementation will allow explorations for other classes of solvers, such as conjugate gradient.

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

References

1. A. Abdelkhalek and A. Bilas. Parallelization and performance of interactive multiplayer game servers. In *IPDPS*, 2004.
2. M. Ansari, C. Kotselidis, K. Jarvis, M. Lujan, Kirkham C., and Watson. Lee-TM: A nontrivial benchmark for transactional memory. In *ICA3PP*, 2008.
3. H. Bae, J. Cownie, M. Klemm, and C. Terboven. A User-Guided Locking API for the OpenMP Application Program Interface. In *IWOMP*, pages 173–186, Salvador, Brazil, September 2014.
4. W. Baek, C.C. Minh, M. Trautmann, C. Kozyrakis, and K. Olukotun. The OpenTM Transactional Application Programming Interface. In *PACT*, pages 376–387, 2007.
5. A.H. Baker, R.D. Falgout, Tz.V. Kolev, and U.M. Yang. Multigrid Smoothers for Ultraparallel Computing . *SIAM J. Sci. Comput.*, 33:2864–2887, 2011.
6. B. L. Bihari. Applicability of transactional memory to modern codes. In *ICNAAM*, pages 1764–1767, Rodos, Greece, 2010. APS.
7. B. L. Bihari, H. Bae, J. Cownie, M. Klemm, C. Terboven, and L. Diachin. On the Algorithmic Aspects of Using OpenMP Synchronization Mechanisms II: User-Guided Speculative Locks. In *IWOMP*, pages 133–148, Aachen, Germany, September 2015.
8. B. L. Bihari, M. Wong, B. R. de Supinski, and L. Diachin. On the Algorithmic Aspects of Using OpenMP Synchronization Mechanisms: The Effects of Transactional Memory. In *IWOMP*, pages 115–129, Salvador, Brazil, September 2014.
9. 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*, pages 44–58, Rome, Italy, June 2012.
10. Hans De Sterck, Robert D. Falgout, Joshua W. Nolting, and Ulrike Meier Yang. Distance-two interpolation for parallel algebraic multigrid. *Numerical Linear Algebra With Applications*, 15:115–139, April 2008.
11. Hans De Sterck, Ulrike Meier Yang, and Jeffrey J. Heys. Reducing complexity in parallel algebraic multigrid preconditioners. *SIAM Journal on Matrix Analysis and Applications*, 27:1019–1039, 2006.
12. Robert D. Falgout, Jim E. Jones, and Ulrike Meier Yang. Pursuing Scalability for *hypre*'s Conceptual Interfaces. *ACM Transactions on Mathematical Software*, 31:326–350, September 2005.
13. V. Gajinov, F. Zyulkyarov, O.S. Unsal, A. Cristal, E. Ayguade, T. Harris, and M. Valero. QuakeTM: Parallelizing a complex sequential application using transactional memory. In *ICS*, pages 126–135, 2009.

14. Transactional Memory Specification Drafting Group. Transactional language constructs for C++. https://sites.google.com/site/tmforcplusplus/, May 2014.
15. R. Guerraoui, M. Kapalka, and J. Vitek. STMBench7: A benchmark for software transactional memory. In *EuroSys*, pages 315–324, 2007.
16. *hypre*: High performance preconditioners. http://www.llnl.gov/CASC/hypre/.
17. S. Kang and D.A. Bader. An Efficient Transactional Memory Algorithm for Computing Minimum Spanning Forest of Sparse Graphs. In *PPoPP*, pages 15–24, 2009.
18. G. Kestor, S. Stipic, O. Unsal, A. Cristal, and M. Valero. RMS-TM: A transactional memory benchmark for recognition, mining and synthesis applications. In *Proc. 4th ACM SIGPLAN Workshop on Transactional Computing TRANSACT*, 2009.
19. V. Luchangco and M. Wong. Transactional Memory Support for C++. http://www.openstd.org/jtc1/sc22/wg21/docs/papers/2014/n3919.pdf, Feb 2014.
20. D. Lupei, B. Simion, Bogdan, D. Pinto, M. Misler, M. Burcea, W. Krick, and C. C. Amza. Transactional Memory Support for Scalable and Transparent Parallelization of Multiplayer Games. In *EuroSys*, pages 41–54, 2010.
21. MFEM: Modular parallel finite element methods library. `http://mfem.googlecode.com`.
22. M. Milovanovic, R. Ferrer, A. Unsal, O.and Cristal, E. Ayguade, J. Labarta, and M. Valero. Transactional Memory and OpenMP. In *IWOMP*, pages 37–53, 2007.
23. C.C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *IISWC*, pages 315–324, 2008.
24. V. Pankratius and A. Adl-Tabatabai. A study of transactional memory vs. locks in practice. In *SPAA*, pages 43–52, 2011.
25. C.J. Rossbach, O.S. Hofmann, and W. Witchel. Is Transactional Programming Actually Easier? In *PPoPP*, pages 47–56, 2010.
26. M. Schindewolf, J. Gyllenhaal, B.L. Bihari, A. Wang, M. Schulz, and W. Karl. What Scientific Applications Can Benefit from Hardware Transacional Memory? . In *SC12*, 2012.
27. M.L. Scott, M.F. Spear, L. Dalessandro, and V.J. Marathe. Delaunay Triangulation with Transactions and Barriers. In *IISWC*, 2007.
28. K. Stüben. An introduction to algebraic multigrid. In U. Trottenberg, C. Oosterlee, and A. Schüller, editors, *Multigrid*, pages 413–528. 2001.
29. A. Wang, M. Gaudet, P. Wu, M. Ohmacht, J.N. Amaral, C. Barton, R. Silvera, and M. MIchael. Evaluation of Blue Gene/Q Hardware Support for Transactional Memories . In *PACT*, 2012.
30. M. Wong, E. Ayguade, J. Gottschlich, V. Luchangco, B. R. de Supinski, and B. L. Bihari. Towards Transactional Memory for OpenMP. In *IWOMP*, Salvador, Brazil, September 2014.
31. M. Wong, B. L. Bihari, B. R. de Supinski, P. Wu, M. Michael, Y. Liu, and W. Chen. A case for including transactions in OpenMP. In *IWOMP*, pages 149–160, Tsukuba, Japan, June 2010.
32. M. Wong and Gottschlich. SG5: Software Transactional Memory (TM) Status Report. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3422.pdf, Sept 2012.
33. R. Yoo, C. Hughes, K. Lai, and R. Rajwar. Performance Evaluation of Intel Transactional Synhcornization Extensions for High-Performance Computing. In *SC13*, 2013.
34. F. Zyulkyarov, V. Gajinov, O.S. Unsal, A. Cristal, E. Ayguade, T. Harris, and M. Valero. Atomic Quake: Using Transactional Memory in an nteractive Multiplayer Game Server. In *PPoPP*, pages 25–34, 2009.