Software Transactional Memory, OpenMP and Pthread Implementations of the Conjugate Gradients Method – A Preliminary Evaluation

Vincent Heuveline[2], Sven Janko[1], Wolfgang Karl[1], Björn Rocker[3], and Martin Schindewolf[1]

[1] Karlsruhe Institute of Technology (KIT), Chair for Computer Architecture and Parallel Processing, Haid-und-Neu-Straße 7, 76131 Karlsruhe, Germany
{sven.janko,karl,schindewolf}@kit.edu

[2] Karlsruhe Institute of Technology (KIT), Engineering Mathematics and Computing Lab (EMCL), Fritz-Erler-Str. 23, 76133 Karlsruhe, Germany
vincent.heuveline@kit.edu

[3] Robert Bosch GmbH, Corporate Sector Research and Advance Engineering, Robert-Bosch-Platz 1, 70839 Gerlingen-Schillerhöhe, Germany
bjoern.rocker@de.bosch.com

Abstract. This paper shows the runtime and cache-efficiency of parallel implementations of the Conjugate Gradients Method based on the three paradigms Software Transactional Memory (STM), OpenMP and Pthreads. While the two last named concepts are used to manage parallelization as well as synchronization, STM was designed to handle only the latter. In our work we disclose that an improved cache-efficiency does not necessarily lead to a better execution time because the execution time is dominated by the thread wait time at the barriers.

Keywords: Software Transactional Memory, OpenMP, Pthreads, Conjugate Gradients Method, Case Study.

1 Introduction and Motivation

Parallelization is state of the art in scientific computing for a long time, but also comes with the need to synchronize parallel threads of execution. Efficient synchronization is the key towards maximum performance on (shared memory) multicore architectures. Traditional synchronization primitives in OpenMP (e.g., `omp critical`) and Pthreads (e.g., locks) achieve synchronization through enforcing mutual exclusion. Threads may experience long delays when waiting for a lock to become available. In the last decade Transactional Memory (TM) has been proposed for synchronization. Instead of following the traditional pessimistic scheme of avoiding memory conflicts, TM favors an optimistic scheme that detects and resolves conflicting accesses. The goal of this strategy is to increase the scalability in regard to a high number of threads and coevally to decrease the time needed for synchronization.

In this paper, we evaluate the applicability of TM for the method of Conjugate Gradients (CG), a solver for linear systems of equations that is frequently used in many fields of application, especially in the area of structural mechanics and computational fluid dynamics.

This paper is structured as follows. Section 2 reviews related work in the area of Transactional Memory research and describes the method of CG. In Section 3 we will discuss our implementations which leads us to Section 4 where we present our results. Section 5 concludes our work and presents ideas for future work.

2 Background on Transactional Memory

Writing efficient, highly scalable and correct parallel software is a challenging task for programmers. They are in charge of the synchronization and communication of the involved threads in order to avoid memory conflicts and deadlocks. Furthermore, one should have consolidated knowledge of the mechanisms of the underlying runtime/operating system.

The idea behind TM is to simplify the process of writing parallel code by providing basic constructs for synchronization. Originally Herlihy and Moss invented TM in 1993 as an architectural extension to enable lock-free data structures [17]. The basic construct is called a transaction and guarantees to execute the comprising load and store commands with three properties: atomicity, consistency and isolation [11]. In contrast to traditional synchronization approaches that enforce mutual exclusion, transactions are executed optimistically in parallel and conflicts are detected and resolved by a TM run time system. The TM system can be implemented in hardware [21,20], software [12,13,14] or as a combination of both as hybrid TM [19,16,18]. In case of a Software Transactional Memory (STM) system a user-level library fulfills this task. All transactional accesses to shared memory are performed through this STM. Often this library comes with compiler support. Then a programmer can use a specific keyword to mark a transaction in the code. For this region of code the compiler inserts calls into the STM instead of performing accesses to shared memory directly. This approach offers the most convenience for the programmer, but also comes at some cost. The compiler makes pessimistic assumptions and, thus, may instrument more memory accesses than absolutely necessary. This phenomenon is known as over-instrumentation [22]. Further, STMs suffer from overheads due to the managing of meta data and acquiring and releasing locks [15]. In our work, we use an STM-only approach with manually instrumented memory accesses, which has the advantage that the resulting binary does not suffer from over-instrumentation through the compiler. OpenMP [6] and Pthreads APIs [7] provide the thread management for the STM.

2.1 Conjugate Gradients

The Method of **Conjugate Gradients** (CG) is a common solver in many fields of application, especially in the area of structural mechanics and computational fluid dynamics. There, finite element and volume methods (FEM/FDM) are frequently employed. Within most linearization methods linear systems have to be solved, consuming often most of the time within the solution process. If those systems are symmetric and positive definite, CG can be applied. Usually, CG is used in combination with an appropriate preconditioning depending on the problem that is solved. Within this paper, we evaluate a pure version of CG.

CG is an improvement of the methods of Steepest Descent and Conjugate Directions where the disadvantage in building the search directions disappears. By conjugation of the residuals the search directions are constructed and it is no longer needed to store the old search vectors (see [5] for a detailed explanation).

Algorithm 1. Conjugate Gradients

```
1: $r_0 = b - Ax_0$, $p_0 = r_0$, A spd
2: for $i = 0, 1, 2, ...$ do
3:     $\alpha_i = \frac{r_i^T r_i}{p_i^T A p_i}$
4:     $x_{i+1} = x_i + \alpha_i p_i$
5:     $r_{i+1} = r_i - \alpha_i A p_i$
6:     $\beta_i = \frac{r_{i+1}^T r_{i+1}}{r_i^T r_i}$
7:     $p_{i+1} = r_{i+1} + \beta_i p_i$
8: end for
```

In the following, n denotes the dimension of the matrix A that is introduced in Algorithm 1. There are one matrix-vector product, three vector updates and two dot-products per iteration cycle. In general the matrix-vector product for computing Ap_j needs n^2 floating-point multiplications and $n^2 - n$ summations, leading to a asymptotic complexity of $O(n^2)$. The complexity for the vector updates is $O(n)$, because n multiplications and n summations for each update are needed. The inner product has also a complexity of $O(n)$. Hence the total complexity per iteration step is dominated by the matrix-vector product. If sparse matrices are used and only nonzero entries are saved the complexity decreases. Supposing a matrix having nnz nonzero entries and $nnz << n^2$. Now, nnz floating-point multiplications are needed and at most $nnz - 1$ summations. The total complexity is $O(nnz)$ compared to $O(n^2)$ in the dense case.

3 Implementations

In the first step we implemented the CG-algorithm as described in Section 2.1 using the C programming language and OpenMP. Then this code was transformed to a similar Pthreads variant and afterwards this version was modified using TM commands. With this approach, we were able to get results that were comparable to each other. The main calculation takes part in five for-loops, corresponding to lines 3 to 7 in Algorithm 1, each iterating n times where n still is the dimension of the underlying matrix of the algorithm.

3.1 OpenMP

In our OpenMP program the parallelization is achieved by inserting *#pragma omp for*-statements on top of each for-loop. Because a for-loop has an implicit barrier, we did not have to care about data dependencies between the several for-loops.

Listing 1.1 shows the five for-loops where most of the execution time is spent. In line 4 and 10 we make use of an OpenMP feature that is called *reduction*.

Every thread, that is part of the calculation, gets its own private copy of the variable *scp_ temp*. Each thread then uses this copy for calculations inside of the loop. Afterwards an addition takes place and the variable *scp_ temp* can be used as the sum of all thread-private variables. As this reduction is generated by the OpenMP compiler and hence is hidden from the programmer, this is exactly where we had to insert commands to achieve mutual exclusion when writing the Pthreads versions (with and without TM, respectively).

Listing 1.1. OpenMP parallelization

```
#pragma omp for private(...) schedule(static)
 for (i=0; i<n; i++){ ... }
 ...
#pragma omp for reduction(+:scp_temp) schedule(static)
 for (i=0; i<n; i++) scp_temp += p[i]*v[i];
 ...
#pragma omp for schedule(static)
 for (i=0; i<n; i++){ ... }
 ...
#pragma omp for reduction(+:scp_temp) schedule(static)
 for (i=0; i<n; i++) scp_temp += r[i]*r[i];
 ...
#pragma omp for schedule(static)
 for (i=0; i<n; i++){ ... }
```

3.2 Pthreads

The basic idea of the OpenMP-to-Pthreads transformation was to pass the main calculation to each created thread modifying the start and end index of each for-loop. With this practice we tried to keep very close to the internal implementation of our OpenMP model. Of course, we also had to reproduce the implicit barriers in OpenMP. We achieved this by inserting explicit barriers that are implemented using the simple function shown in Listing 1.2.

3.3 Transactional Memory

The third model of the CG-algorithm was written using our Pthreads program as basis. Only a few lines in the TM-implementation differ from this code. We used the same thread creation concept and also the same barriers. We customized our code mainly in two places by inserting TM instructions to generate a transaction. With this transaction, threads will optimistically read and write the shared variable *scp_ temp* concurrently. Listing 1.3 shows a TM version of the reduction that was previously mentioned in Section 3.1.

Listing 1.2. Pthreads barrier implementation

```
typedef struct barrier {
  pthread_cond_t complete;
  pthread_mutex_t mutex;
  int count;
  int crossing;
} barrier_t;

void barrier_cross(barrier_t *b) {
  pthread_mutex_lock(&b->mutex);
  b->crossing++;
// one more thread through
  if (b->crossing < b->count) {
// if not all here, wait
    pthread_cond_wait(&b->complete, &b->mutex);
  } else {
    pthread_cond_broadcast(&b->complete);
// last thread arrived
    b->crossing = 0;
// Reset for next time
  }
  pthread_mutex_unlock(&b->mutex);
}
```

Listing 1.3. TM reduction

```
for (i = thread->start; i < thread->end; i++) {
  scp_temp_private += p[i]*v[i]; }
START(thread->id, RW);
  scp_temp_private += (double)LOAD_DOUBLE(&scp_temp);
  STORE_DOUBLE(&scp_temp, scp_temp_private);
COMMIT;
```

4 Numerical Experiments

4.1 Hardware and Software Environment

All experiments were run on two computers C1 and C2 which are described in detail in Table 1. As compiler, *gcc-4.4* was invoked with options -*O3* and -g3. As Software Transactional Memory library we chose TinySTM [9,10]. TinySTM is a lightweight and efficient word-based STM implementation. Its time-based algorithm is derived from LSA and its lock-based design borrows several key elements from other word-based STMs, such as TL2.

Table 1. Experimental Setup

	Computer 1 (C1)	**Computer 2 (C2)**
CPU name	Intel Xeon X5670[1]	AMD Opteron 2378[2]
#Sockets	two	two
CPU frequency	2.93 GHz	2.36 GHz
RAM	12 GB	16 GB
Size of L1	32 KB	64 KB
Size of L2	256 KB	512 KB
OS	GNU/Linux (Ubuntu)	GNU/Linux (Ubuntu)
Kernel version	2.6.32-29-server	2.6.38-12-server
Architecture	x86_64	x86_64
Hyper-threading	yes	no
NUMA	yes	yes

4.2 Numerical Results

Each of our tests were run several times (>15) taking into account the exclusive computing time for the process. Afterwards we calculated the arithmetic mean of the results omitting the fastest and the slowest run. Thus, every value in the subsequent figures is an arithmetic average of at least 14 executions.

We evaluated the performance assuming a sparse matrix described by means of a CSR format. The linear system is obtained from a finite element discretization of the stationary heat equation without heat source (homogeneous case) which represents a prototype of Laplace's equation. It is equivalent to a finite differences discretization based on the 3-point-stencil. The matrix has a dimension of 5 000 000 and 14 999 998 nonzero entries (nnz). The residual stopping criteria for the residual is set to 10^{-13}.

Performance. As expected, with all three paradigms we could achieve significant speedups over the respective single thread execution time by increasing the number of threads from one to two, three, four and more. On Computer 1 we achieved a speedup of $S_8 = 2.72$ (OpenMP), $S_8 = 3.42$ (Pthreads) and $S_8 = 3.79$ (STM) by increasing the number of threads from one to eight. See Figure 1. The dimension of the underlying matrix was set to $5M$ in this case. Although there are clear differences in the above-named speedups, the execution time does not differ much with eight threads on C1. The good speedup with STM is also due to the high single thread overhead. A special case is 24 threads and OpenMP: the calculation takes slightly longer than with the single threaded concept. A model that describes the effects of the scheduling on the run time of the application explains the peak with 24 threads. Christmann et al. developed this model when they where researching the impact of oversubscription on the application throughput [8]. The scheduling algorithm must be fair (each process

[1] Registered Trademark by Intel Corporation.
[2] Registered Trademark by AMD.

gets a fair share of time), balance the load across cores (or hardware threads) and pins a process to a processing element as long as possible. Our case meets all of these assumptions. The explanation for the peak in execution time is that a fully loaded node (with 24 OpenMP threads) competes with some background process for computing resources. Eventually, after a long stall time, one of the OpenMP threads gets migrated leading to a prolonged overall execution time. Later experiments verified that a later Linux kernel (with version number 3.0.0-23-server) that enables a fair scheduling of groups instead of processes does not show this behavior anymore.

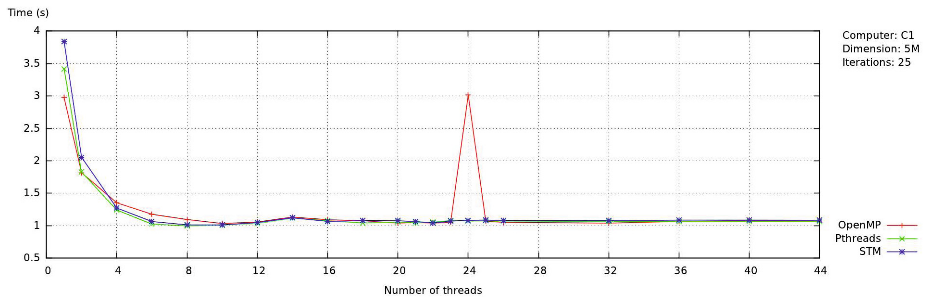


Fig. 1. Runtime analysis of the CG method (OpenMP, Pthreads, STM)

Another finding of our research is that the Pthread-program (and also the TM-program) is in the majority of cases slightly slower than the OpenMP-variation. We see mainly two causes therefor: a) more cache misses (see Section *Cache-Efficiency Analysis*) and b) more time is spent at the barriers. We will discuss the second argument in more detail now. We measured the time that the threads had to wait at each barrier in the Pthreads-program on C2. For two threads it took 7-15% of the overall execution time to wait at the barriers. Four threads waited about 25%, six threads about 43% and eight threads even about 70% of the execution time. What we discovered with this analysis is, that the time at the barriers increases rapidly if there are pairs of threads that have the same Hardware-Thread-ID. That means these threads cannot be executed in parallel because they are mapped to the same hardware entity and hence have to run one after the other. Those pairs appear even if the number of threads is less than the number of possible hardware threads in the system, which is an important insight. Apparently this is nothing the software developer is able to control.

PARSEC Barrier Tests. Another test concerning the barriers was the comparison of two slightly varying Pthreads programs. On the one hand, we used the constructs for the barriers as described in Listing 1.2, on the other hand, the PARSEC barriers were tested [2]. When using the PARSEC barriers, one can choose between two modes: 1) spinning ON and 2) spinning OFF. The results (executed on C2) are shown in Figure 2.

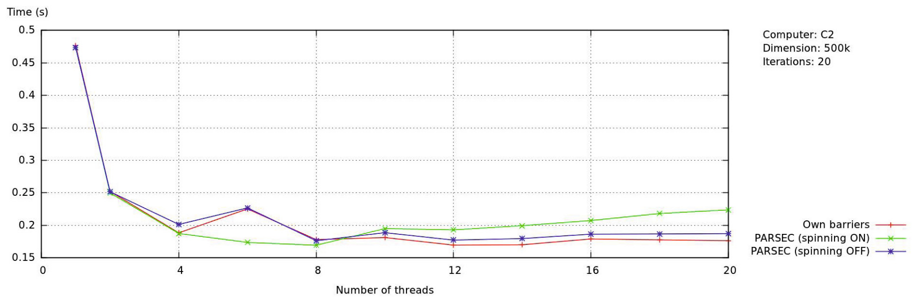


Fig. 2. PARSEC barrier comparison

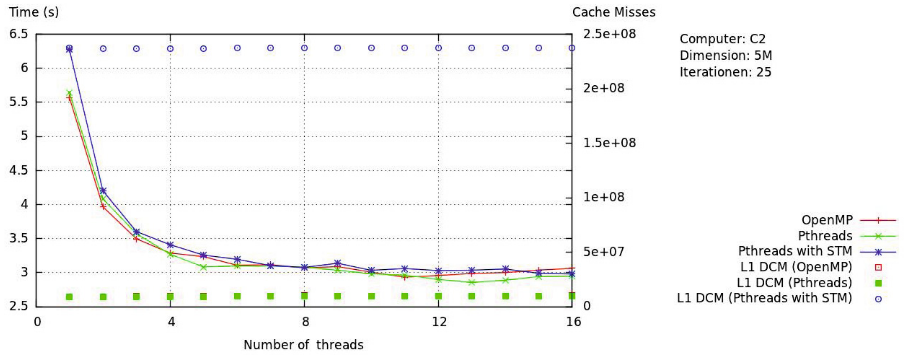


Fig. 3. Level 1 data cache misses

In general, using the PARSEC barriers did not bring strong advantages over the simple implementation which we used earlier. On the contrary, it was even slower for most configurations. Only for four to eight threads, if the spinning option was set to *ON*, it resulted in a faster runtime. As shown in Figure 2, the execution time increases for more than eight threads. That is exactly as we expected. In this example, spinning does not make any sense for a higher number of threads.

Cache-Efficiency Analysis. In order to understand the differences in runtime we also studied the cache behavior in detail. Our main focus was on the data cache, because the instruction cache analysis did not reveal noticeable results. The following designations apply to C2. As one can see in Figure 3, the data cache misses of the first level cache (L1 DCM) do not change with an increasing number of threads[3], whereas the L2 DCMs increase at the same time (see Figure 4). This holds as long as the number of threads is less or equal the number of possible hardware threads (here 8) in the system. Beyond this point the L2 DCMs are not increasing anymore. From Figure 4 we educe that there is no direct correlation of the L2 DCMs and the execution time of the program. Rising

[3] The DCMs of OpenMP are hidden behind the DCMs of Pthreads.

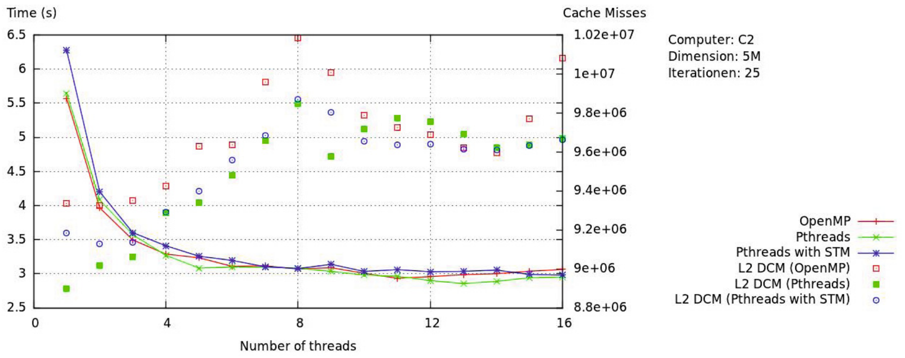


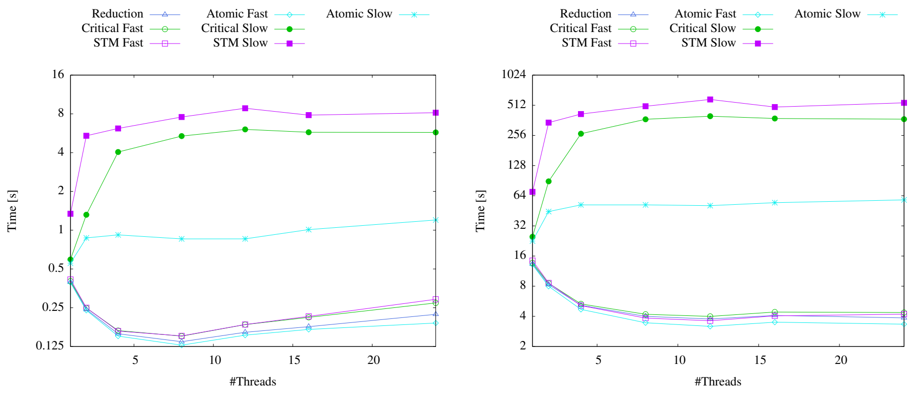
Fig. 4. Level 2 data cache misses

L2 DCMs do not necessarily bring a slower execution time and on the contrary, falling L2 DCMs do not always result in a faster execution time. This holds for all three programs.

If we now compare Figure 4 and 2, it becomes apparent that the waiting time at the barriers dominates the execution time of the programs. As one can see in Listing 1.2, the main function of the barrier construct is to pause a thread at a specific point of execution until all other threads reach the barrier. That means, that the last thread significantly increases the execution time. Thus, increasing the number of threads only makes sense, if the time that is spent at the barriers is improved, too.

4.3 Experiments with Matrices from Structural Engineering

In this section we will add additional experiments with two more matrices to provide a richer evaluation of the implemented CG variants. In order to complement the findings from the previous section, we also distinguish two more implementation variants that differ in the implementation of the reduction. The two reductions in CG are each implemented in two ways: *Fast*, and *Slow*. *Fast* uses a thread-local variable to accumulate the results over a private part of the vector that is assigned to this specific thread. Then, a single update adds the thread-local variable to the shared memory one that is guarded by a critical section or transaction. Thus, contention between threads only arises from the update of the shared memory variable. The *Fast* version of CG updates one shared memory location per thread and reduction pattern. Thus, the number of executed transactions equals the number of threads times the number of reductions per iteration. This is the reduction pattern that has also been used for the previous experiments presented in this paper. The *Slow* version updates the shared memory location in one transaction or critical section and does not use thread-local variables. Because each reduction only updates one shared memory location, OpenMP atomic is a perfect fit because it maps to a processor instruction that assures the atomicity of the update (if the processor supports



(a) Matrix bcsstk14. (b) Matrix bcsstk18.

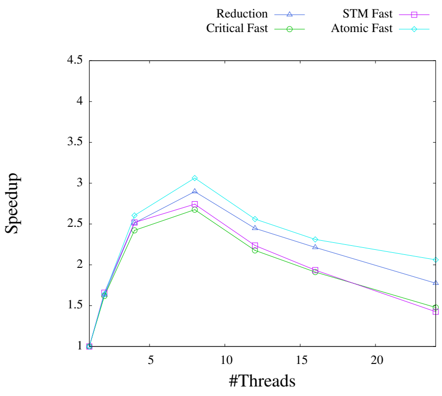
Fig. 5. Run times with OpenMP and all variants of synchronization mechanisms

atomics). The *Fast* version, again, uses thread-local variables whereas the *Slow* version does not. This atomicity is limited to one memory location and can not be extended. Thus, the *Atomic Fast* uses the thread-local variables to update the shared memory locations and the *Atomic Slow* updates the shared memory location for each new value. Since each value must be updated with a separate atomic instruction there is no need to distinguish between long and short sections. These self-made reductions are complemented by the OpenMP reduction, denoted as *Reduction*, that the programmer specifies through using a `#pragma omp for reduction(+:var) schedule(static)`.

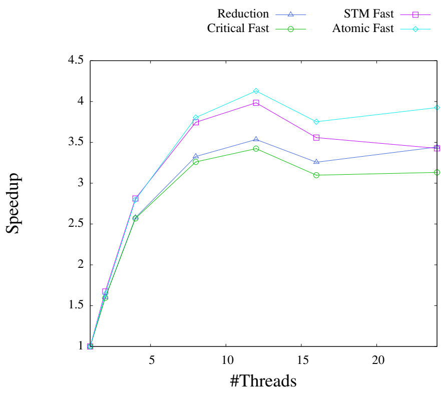
The two additional matrices are taken from the matrix market[4]. This assures that other researchers may compare their results with ours. The first matrix is called bcsstk14, has a dimension of 1806 with 32630 entries. The matrix has a Frobenius norm of $6.5 * 10^{10}$ and an estimated condition number of $1.3 * 10^{10}$. The matrix is used for static analysis in structural engineering and models the roof of the Omni Coliseum in Atlanta. The second matrix, called bcsstk18, has a dimension of 11948 with 80519 entries, a Frobenius norm of $2.4 * 10^{11}$ and an estimated condition number of 65. Both matrices are from the set BCSSTRUC2 of Prof Mac Will, Georgia Institute of Technology. As experimental setup, we use again C1 with OpenMP parallelization only this time running Linux kernel version number 3.0.0-23-server that enables a fair scheduling of groups instead of processes and, thus, does not show the peak in the run time with 24 threads (cf. to Section 4.2).

Figure 5 highlights the run time and shows that the implementation strategy of the reduction is more important than the choice of the synchronization mechanisms for this reduction. Clearly all *Slow* variants perform worse than their single-threaded counter parts. This is due to the contention on the shared variables that are updated in each loop iteration. The *Fast* variants show a far better scalability due to a reduction in time with an increasing thread number.

[4] `http://math.nist.gov/MatrixMarket`

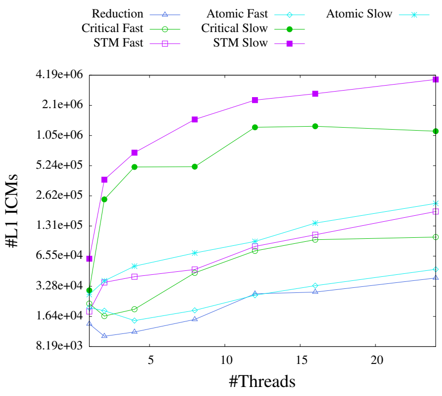


(a) Matrix bcsstk14.

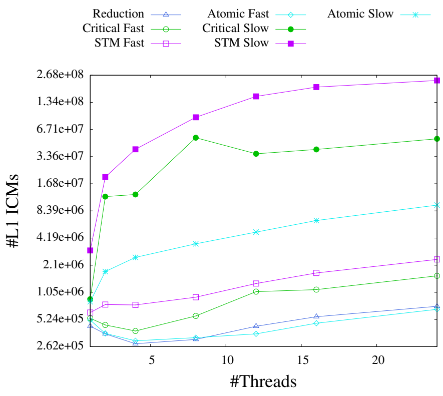


(b) Matrix bcsstk18.

Fig. 6. Speedup of the *Fast* synchronization variants over the respective single thread performance



(a) Matrix bcsstk14.



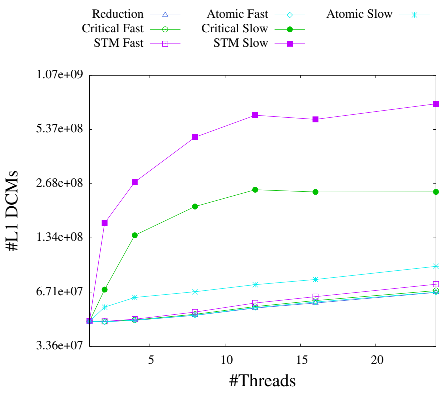
(b) Matrix bcsstk18.

Fig. 7. L1 instruction cache misses with different synchronization variants

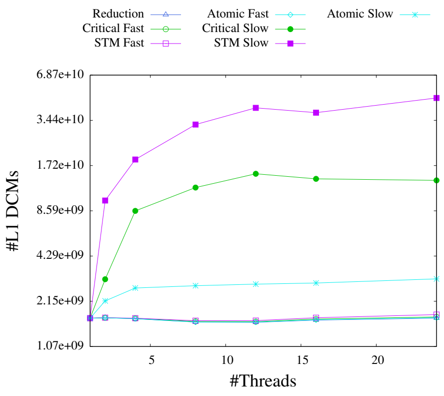
In Figure 6 the speedup over the respective single thread performance of the *Fast* variants. *Atomic Fast* achieves the highest speedup for bcsstk14 with 8 and for bcsstk18 with 12 threads. For bcsstk18 *STM Fast* also performs almost as good as *Atomic Fast*.

Figure 7 shows the L1 instruction misses for both matrices. The *Slow* variants have a significant higher number of instruction cache misses than the *Fast* variants. The interesting observation is that for bcsstk14 *Atomic Slow* is almost as good as *STM Fast*. This shows the large overhead in terms of instructions that is associated with using an STM system. The *Reduction* and *Atomic Fast* utilize the instruction cache the most efficiently.

For the L1 data cache misses, shown in Figure 8, the trend is similar as with the L1 instruction cache miss but the gap between STM and the other mechanisms

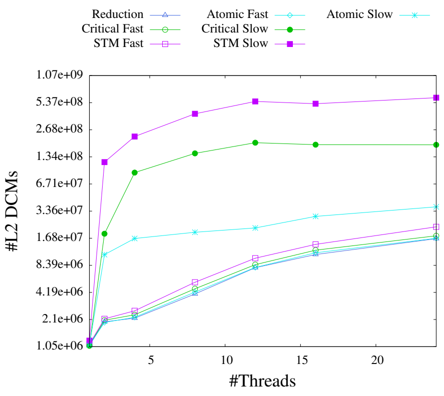


(a) Matrix bcsstk14.

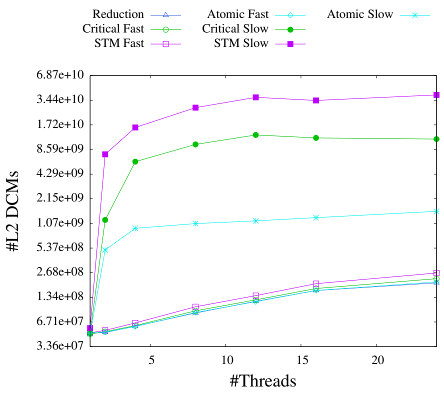


(b) Matrix bcsstk18.

Fig. 8. L1 data cache misses of all different synchronization variants



(a) Matrix bcsstk14.



(b) Matrix bcsstk18.

Fig. 9. L2 data cache misses with OpenMP and all variants of synchronization mechanisms

is not as big when it comes to the *Fast* variants. The *Slow* versions again have significantly more misses due to the contention on the shared variable.

Figure 9 highlights the L2 data cache misses of both matrices and implementation variants across thread counts. Again *STM Fast* is slightly worse than the other *Fast* variants but still significantly better than the *Slow* variants. For matrix bcsstk18 the gap between *STM Fast* and the rest seems smaller which may be due to the larger size of the matrix. Calculating the L2 data cache miss rate according to $\frac{L2\ data\ cache\ misses}{L2\ data\ cache\ accesses}$ yields the following results. For the *Fast* variants there is an almost linear increase in the L2 data cache miss rate with the number of threads whereas the *Slow* variants follow a logarithmic curve which results in a rate of more than 80 %. For *Fast* the rate stays well below 20 % for bcsstk18 and 30 % for bcsstk14.

5 Conclusion and Future Work

In our work we compared three similar implementations of the Conjugate Gradients Method. One that uses OpenMP, one that uses Pthreads without TM and one that uses Pthreads with TM constructs. The results showed that it is very important to reduce the waiting time at the barriers in order to improve execution time of these programs. Complementary experiments reveal that the choice for implementing the reduction is even more important than the choice of the synchronization primitive. Using thread-local variables for implementing the reduction is indispensable for a well-performing implementation. Further, these experiments with two additional matrices, lent from the static analysis in structural engineering, confirm the findings of the previous experiments regarding the cache efficiency of STM. In most cases, OpenMP is the fastest approach on both machines. This is the case because STM suffers from significantly more L1 cache misses compared to a pure OpenMP or Pthread implementation. In terms of performance, OpenMP is the first choice if the CG algorithm is used as done in this paper. As future work, the above-mentioned programs should be compared to other formulations of the Conjugate Gradients Method, such as the pipelined CG-algorithm described in [1] in order to benefit from advantages with TM. Further, we want to research the influence of using a NUMA machine (e.g., through employing a first touch policy for memory pages) on the performance of the different implementations of the CG method.

References

1. Strzodka, R., Göddeke, D.: Pipelined Mixed Precision Algorithms on FPGAs for Fast and Accurate PDE Solvers from Low Precision Components. In: IEEE Proceedings on Field-Programmable Custom Computing Machines (2006)
2. Bienia, C.: Benchmarking Modern Multiprocessors. Princeton University (January 2011)
3. Bolz, J., Farmer, I., Grinspun, E., Schröder, P.: Sparse matrix solvers on the GPU: conjugate gradients and multigrid. ACM Transactions on Graphics 22, 917–924 (2003)
4. Goodnight, N., Lewin, G., Luebke, D., Skadron, K.: A multigrid solver for boundary-value problems using programmable graphics hardware. In: Eurographics/SIGGRAPH Workshop on Graphics Hardware, pp. 102–111 (2003)
5. Saad, Y.: Iterative Methods for Sparse Linear Systems (2003)
6. OpenMP Architecture Review Board: OpenMP Application Program Interface. Version 3.1 (July 2011), `http://www.openmp.org/mp-documents/OpenMP3.1.pdf`
7. Butenhof, D.: Programming with POSIX threads. Addison-Wesley Longman Publishing Co., Inc. (1997)
8. Christmann, C., Hebisch, E., Weisbecker, A.: Oversubscription of Computational Resources on Multicore Desktop Systems. In: Pankratius, V., Philippsen, M. (eds.) MSEPT 2012. LNCS, vol. 7303, pp. 18–29. Springer, Heidelberg (2012)
9. Felber, P., Fetzer, C., Marlier, P., Riegel, T.: Time-Based Software Transactional Memory (2010)

10. Felber, P., Fetzer, C., Riegel, T.: Dynamic performance tuning of word-based software transactional memory. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (2008)
11. Larus, J., Rajwar, R.: Transactional Memory. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers (2007)
12. Lev, Y., Luchangco, V., Marathe, V., Moir, M., Nussbaum, D., Olszewski, M.: Anatomy of a Scalable Software Transactional Memory. In: Workshop on Transactional Computing TRANSACT 2009 (February 2009)
13. Saha, B., Adl-Tabatabai, A., Hudson, R., Minh, C., Hertzberg, B.: McRT-STM: a high performance software transactional memory system for a multi-core runtime. In: PPoPP 2006: Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 187–197 (2006) ISBN 1-59593-189-9
14. Dice, D., Shalev, O., Shavit, N.: Transactional Locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006)
15. Cascaval, C., Blundell, C., Michael, M., Cain, H.W., Wu, P., Chiras, S., Chatterjee, S.: Software Transactional Memory: Why is it Only a Research Toy? Queue 6(5), 46–58 (2008) ISSN 1542-7730
16. Lev, Y., Moir, M., Nussbaum, D.: PhTM: Phased Transactional Memory. In: TRANSACT 2007: 2nd Workshop on Transactional Computing (August 2007)
17. Herlihy, M., Moss, E.: Transactional memory: architectural support for lock-free data structures. SIGARCH Comput. Archit. News 21(2), 289–300 (1993) ISSN 0163-5964
18. Christie, D., Chung, J., Diestelhorst, S., Hohmuth, M., Pohlack, M., Fetzer, C., Nowack, M., Riegel, T., Felber, P., Marlier, P., Rivière, E.: Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack. In: EuroSys 2010: Proceedings of the 5th European Conference on Computer Systems, pp. 27–40 (2010) ISBN 978-1-60558-577-2
19. Damron, P., Fedorova, A., Lev, Y., Luchangco, V., Moir, M., Nussbaum, D.: Hybrid transactional memory. In: ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 336–346 (2006) ISBN 1-59593-451-0
20. Yen, L., Bobba, J., Marty, M., Moore, K., Volos, H., Hill, M., Swift, M., Wood, D.: LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In: IEEE 13th International Symposium on High Performance Computer Architecture (HPCA), pp. 261–272 (February 2007) ISBN 1-4244-0804-0
21. Hammond, L., Wong, V., Chen, M., Carlstrom, B., Davis, J., Hertzberg, B., Prabhu, M., Wijaya, H., Kozyrakis, C., Olukotun, K.: Transactional Memory Coherence and Consistency. In: Proceedings of the 31st Annual International Symposium on Computer Architecture, p. 102. IEEE Computer Society (June 2004)
22. Yoo, R., Ni, Y., Welc, A., Saha, B., Adl-Tabatabai, A., Lee, H.: Kicking the tires of software transactional memory: why the going gets tough. In: SPAA 2008: Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures, pp. 265–274 (2008) ISBN 978-1-59593-973-9