Karlsruhe Institute of Technology

Software Transactional Memory, OpenMP and Pthread implementations of the Conjugate Gradients Method - a Preliminary Evaluation

S. Janko, B. Rocker, M. Schindewolf,
V. Heuveline, W. Karl

No. 2012-01

Preprint Series of the Engineering Mathematics and Computing Lab (EMCL)

KIT – University of the State of Baden-Wuerttemberg and
National Research Center of the Helmholtz Association

www.emcl.kit.edu

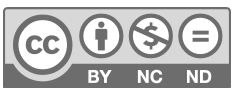
Preprint Series of the Engineering Mathematics and Computing Lab (EMCL)
ISSN 2191–0693
No. 2012-01

Impressum

Karlsruhe Institute of Technology (KIT)
Engineering Mathematics and Computing Lab (EMCL)

Fritz-Erler-Str. 23, building 01.86
76133 Karlsruhe
Germany

KIT – University of the State of Baden Wuerttemberg and
National Laboratory of the Helmholtz Association



Software Transactional Memory, OpenMP and Pthread implementations of the Conjugate Gradients Method - a Preliminary Evaluation

Sven Janko[1], Björn Rocker[2,3], Martin Schindewolf[1], Vincent Heuveline[2], Wolfgang Karl[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, bjoern.rocker}@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.

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. To begin with, Section 2 gives a short review of the above-mentioned programming paradigms. Also, the method of CG is described. 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

2.1 OpenMP

One of the most common shared memory programming models is OpenMP (Open Multi-Processing) [6]. More precisely, OpenMP is a shared memory application programming interface (API) that can be used with the programming languages Fortran, C and C++. With OpenMP it is possible to describe how computations are shared on different threads running on one or different processors or cores. To do so *compiler directives* (often called only directives) are used to specify how the instructions have to be computed in parallel. Of course, the compiler has to support OpenMP which is nowadays the case for most compilers (e.g. compilers from GNU, IBM, Intel, PGI, Pathscale etc.).

In many cases it is possible to parallelize sequential code with OpenMP, especially when the code contains loops and the data dependency between several loop iterations is low. Sometimes this is not enough to get a sufficient level of performance and a reorganization of the code is needed to create parallelism.

2.2 Pthreads

Pthreads stands for POSIX Threads, an API for managing threads in user space [7]. This multithreading model can be used with the programming languages C and C++ and implementations of the library exist for most Unix-like operating systems as well as for Windows.

Compared to OpenMP more effort is required to achieve and manage parallelization. One has to exploit every loop by defining the borders for each threads calculation and one has to write constructs for reduction and similar patterns by oneself, which can be easily accomplished with one line of code when using OpenMP. In exchange the developer has full control over the granularity of the parallelization as well as a higher flexibility in using complex data structures.

2.3 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. These constructs are called transactions and guarantee to execute the comprising load and store commands with three properties: atomicity, consistency and isolation [10]. 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. In case of a Software Transactional Memory system a user-level library fulfills this task. This STM is combined with an API for thread management such as the before-mentioned OpenMP and Pthreads APIs.

2.4 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, a pure version of CG is evaluated.

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: $\quad \alpha_i = \frac{r_i^T r_i}{p_i^T A p_i}$
4: $\quad x_{i+1} = x_i + \alpha_i p_i$
5: $\quad r_{i+1} = r_i - \alpha_i A p_i$
6: $\quad \beta_i = \frac{r_{i+1}^T r_{i+1}}{r_i^T r_i}$
7: $\quad p_{i+1} = r_{i+1} + \beta_i p_i$
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 2.4 using the C programming language and OpenMP. Afterwards this code was transformed to a similar Pthreads variation 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 of the implicit barriers 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

```
1 #pragma omp for private(...) schedule(static)
2 for (i=0; i<n; i++){ ... }
3 ...
4 #pragma omp for reduction(+:scp_temp) schedule(static)
5 for (i=0; i<n; i++) scp_temp += p[i]*v[i];
6 ...
7 #pragma omp for schedule(static)
8 for (i=0; i<n; i++){ ... }
9 ...
```

```
10 #pragma omp for reduction(+:scp_temp) schedule(static)
11 for (i=0; i<n; i++) scp_temp += r[i]*r[i];
12 ...
13 #pragma omp for schedule(static)
14 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 (OpenMP). We achieved this by calling the simple function shown in Listing 1.2.

Listing 1.2: Pthreads barrier implementation

```
1  typedef struct barrier {
2    pthread_cond_t complete;
3    pthread_mutex_t mutex;
4    int count;
5    int crossing;
6  } barrier_t;
7
8  void barrier_cross(barrier_t *b)
9  {
10   pthread_mutex_lock(&b->mutex);
11   b->crossing++; // one more thread through
12   if (b->crossing < b->count) { // if not all here, wait
13     pthread_cond_wait(&b->complete, &b->mutex);
14   } else {
15     // last thread arrived
16     pthread_cond_broadcast(&b->complete);
17     /* Reset for next time */
18     b->crossing = 0;
19   }
20   pthread_mutex_unlock(&b->mutex);
21 }
```

3.3 Transactional Memory

The third model of the CG-algorithm was written using our Pthreads program as basis. Only 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 on two places where there had to be mutual exclusion by inserting TM instructions to generate a transaction. A code example can be seen in Listing 1.3. It shows the reduction that was previously mentioned in Section 3.1.

Listing 1.3: TM reduction

```
1 for (i = thread->start; i < thread->end; i++) {
2   scp_temp_private += p[i]*v[i];
3 }
4 START(thread->id, RW);
5   scp_temp_private += (double)LOAD_DOUBLE(&scp_temp);
6   STORE_DOUBLE(&scp_temp, scp_temp_private);
7 COMMIT;
```

4 Numerical Experiments

4.1 Hardware and Software Environment

All experiments were run on two computers, the main parameters are shown in Table 1, for complete topology see Figures 1 and 2. As compiler, *gcc-4.4* was invoked with *-O3 -g3* as options.

	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

Table 1: Experimental Setup

As Software Transactional Memory library we chose TinySTM [8,9]. 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.

Fig. 1: Topology of the system for Computer 1

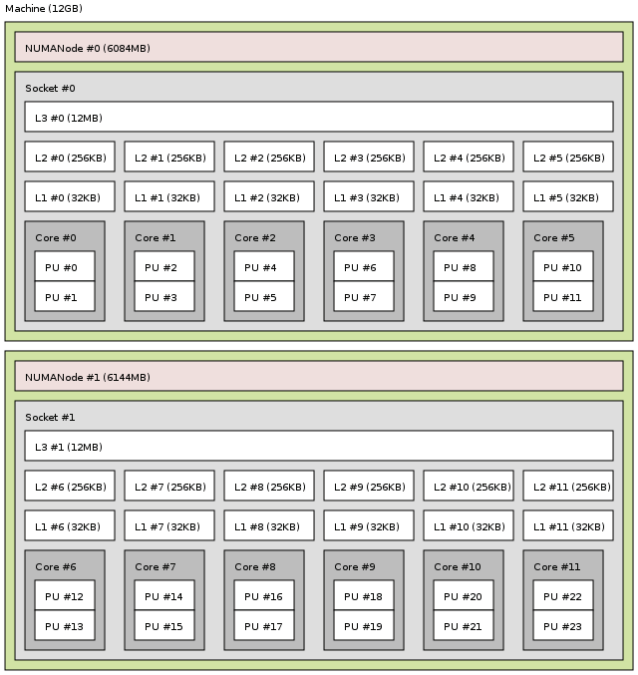
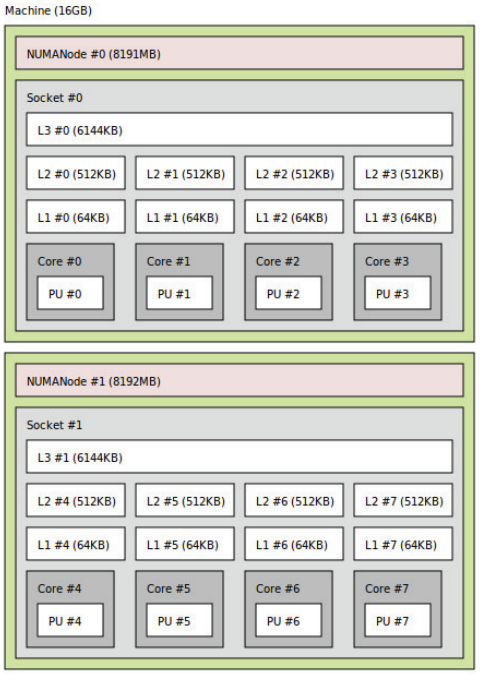


Fig. 2: Topology of the system for Computer 2



4.2 Numerical Results

Each of our tests were run several (>15) times 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.

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

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 was considered on a unit square using linear test-functions, which is equivalent to a finite differences discretization based on the 5-point-stencil. The matrix has the following characteristics: 4.000.000 degrees of freedom (dofs) and 19.992.000 nonzero entries (nnz). The residual stopping criteria for the residual is set to 10^{-10}.

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. The dimension of the underlying matrix was set to $5M$ in this case. See Figure 3. Although there are clear differences in the above-named speedups, the execution time does not differ much between the three paradigms for more than eight threads on Computer 1. Except for the calculation with 24 threads and OpenMP: here it takes slightly longer than the single threaded concept. Another finding of our research is that the Pthread-

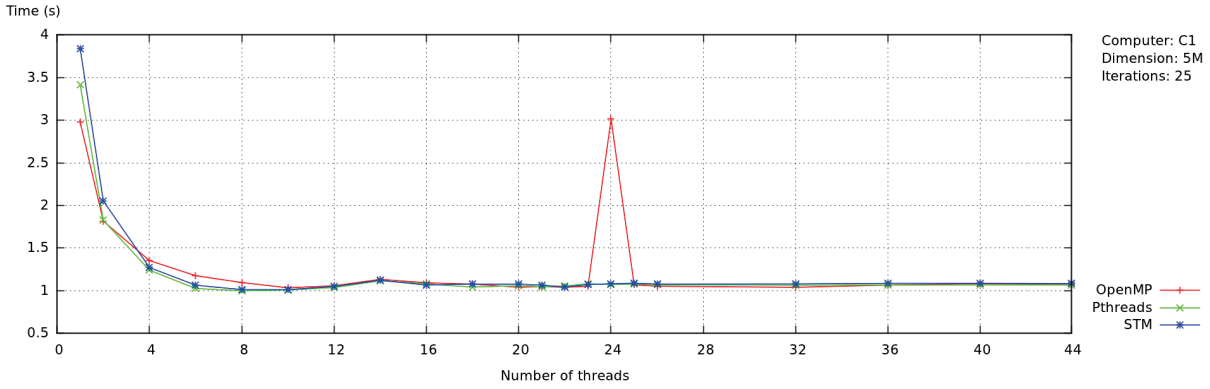


Fig. 3: Runtime analysis of the CG method (OpenMP, Pthreads, STM)

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. This second argument we will discuss in more detail now. We measured the time that the threads had to wait at each barrier in the Pthreads-program on Computer 2. 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 Computer 2) are shown in Figure 4.

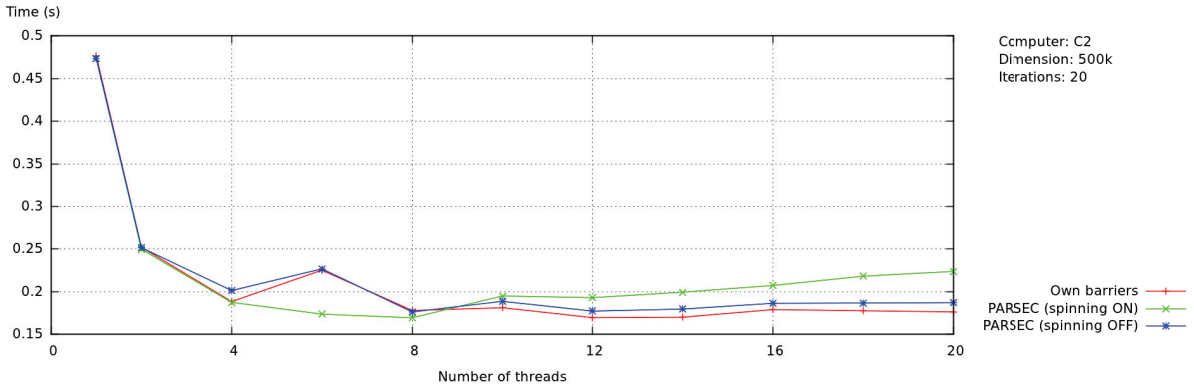


Fig. 4: PARSEC barrier comparison

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 4, 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 Computer 2.

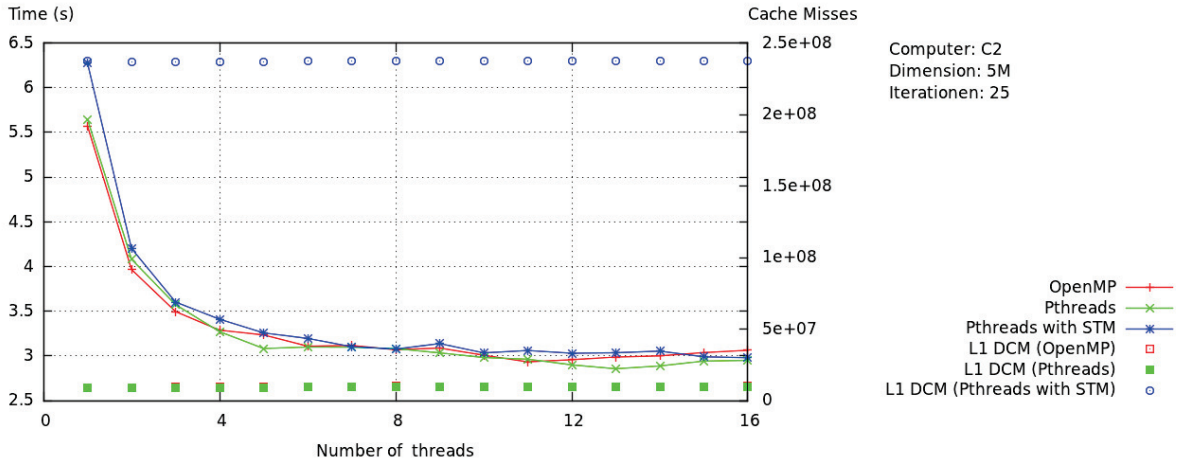


Fig. 5: Level 1 data cache misses

As one can see in Figure 5, the data cache misses of the first level cache (L1 DCM) do not change with an increasing number of threads[4], whereas the L2 DCMs increase at the same time (see Figure 6). 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 6 we educe that there is no direct correlation of the L2 DCMs and the execution time of the program. Rising 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.

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

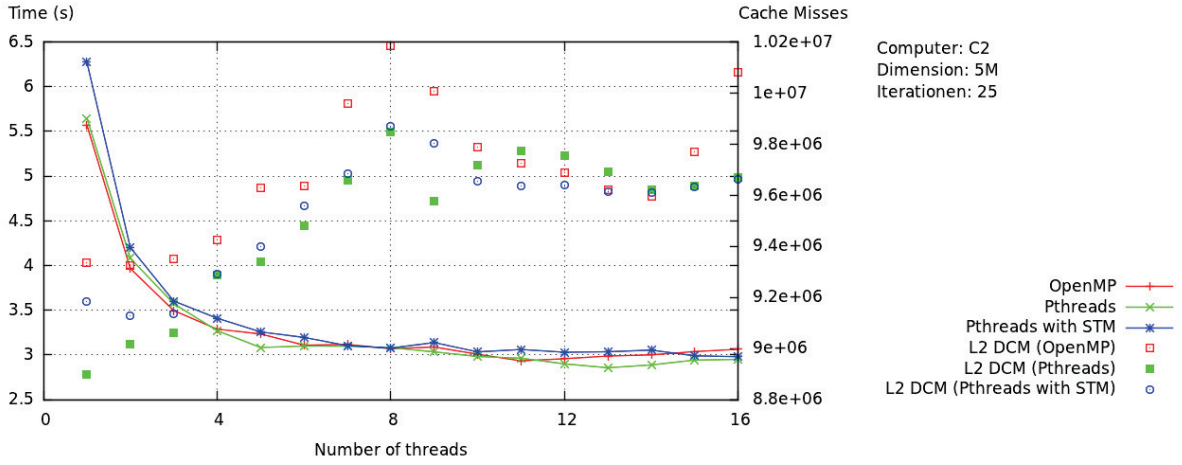


Fig. 6: Level 2 data cache misses

If we now compare Figure 6 and 4, 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. 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.

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 the programs.

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 TM advantages.

References

1. Strzodka, R., Göddeke, D.: Pipelined Mixed Precision Algorithms on FPGAs for Fast and Accurate PDE Solvers from Low Precision Components. 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, vol. 22, 2003, pp. 917-924.
4. Goodnight, N., Lewin, G., Luebke, D., Skadron, K.: A multigrid solver for boundary-value problems using programmable graphics hardware. Eurographics/SIGGRAPH Workshop on Graphics Hardware, 2003, pp. 102-111.
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. 1997, Addison-Wesley Longman Publishing Co., Inc.
8. Felber, P., Fetzer, C., Marlier, P., Riegel, T.: Time-Based Software Transactional Memory. 2010.
9. Felber, P., Fetzer, C., Riegel, T.: Dynamic performance tuning of word-based software transactional memory. Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, 2008.
10. Larus, J., Rajwar, R.: Transactional Memory. Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers 2007,

Preprint Series of the Engineering Mathematics and Computing Lab

recent issues

No. 2011-17 Hartwig Anzt, Jack Dongarra, Vincent Heuveline, Piotr Luszczek: GPU-Accelerated Asynchronous Error Correction for Mixed Precision Iterative Refinement

No. 2011-16 Vincent Heuveline, Sebastian Ritterbusch, Staffan Ronnås: Augmented Reality for Urban Simulation Visualization

No. 2011-15 Hartwig Anzt, Jack Dongarra, Mark Gates, Stanimire Tomov: Block-asynchronous multigrid smoothers for GPU-accelerated systems

No. 2011-14 Hartwig Anzt, Jack Dongarra, Vincent Heuveline, Stanimire Tomov: A Block-Asynchronous Relaxation Method for Graphics Processing Units

No. 2011-13 Vincent Heuveline, Wolfgang Karl, Fabian Nowak, Mareike Schmidtobreick, Florian Wilhelm: Employing a High-Level Language for Porting Numerical Applications to Reconfigurable Hardware

No. 2011-12 Vincent Heuveline, Gudrun Thäter: Proceedings of the 4th EMCL-Workshop Numerical Simulation, Optimization and High Performance Computing

No. 2011-11 Thomas Gengenbach, Vincent Heuveline, Mathias J. Krause: Numerical Simulation of the Human Lung: A Two–scale Approach

No. 2011-10 Vincent Heuveline, Dimitar Lukarski, Fabian Oboril, Mehdi B. Tahoori, Jan-Philipp Weiss: Numerical Defect Correction as an Algorithm-Based Fault Tolerance Technique for Iterative Solvers

No. 2011-09 Vincent Heuveline, Dimitar Lukarski, Nico Trost, Jan-Philipp Weiss: Parallel Smoothers for Matrix-based Multigrid Methods on Unstructured Meshes Using Multicore CPUs and GPUs

No. 2011-08 Vincent Heuveline, Dimitar Lukarski, Jan-Philipp Weiss: Enhanced Parallel ILU(p)-based Preconditioners for Multi-core CPUs and GPUs – The Power(q)-pattern Method

No. 2011-07 Thomas Gengenbach, Vincent Heuveline, Rolf Mayer, Mathias J. Krause, Simon Zimny: A Preprocessing Approach for Innovative Patient-specific Intranasal Flow Simulations

No. 2011-06 Hartwig Anzt, Maribel Castillo, Juan C. Fernández, Vincent Heuveline, Francisco D. Igual, Rafael Mayo, Enrique S. Quintana-Ortí: Optimization of Power Consumption in the Iterative Solution of Sparse Linear Systems on Graphics Processors

No. 2011-05 Hartwig Anzt, Maribel Castillo, José I. Aliaga, Juan C. Fernández, Vincent Heuveline, Rafael Mayo, Enrique S. Quintana-Ortí: Analysis and Optimization of Power Consumption in the Iterative Solution of Sparse Linear Systems on Multi-core and Many-core Platforms

No. 2011-04 Vincent Heuveline, Michael Schick: A local time–dependent Generalized Polynomial Chaos method for Stochastic Dynamical Systems

No. 2011-03 Vincent Heuveline, Michael Schick: Towards a hybrid numerical method using Generalized Polynomial Chaos for Stochastic Differential Equations