

Available online at www.sciencedirect.com**SciVerse ScienceDirect**

Procedia Computer Science 18 (2013) 501 – 510

Procedia
Computer Science

International Conference on Computational Science, ICCS 2013

Parallelizing the sparse matrix transposition: reducing the programmer effort using transactional memory

Miguel A. Gonzalez-Mesa, Eladio D. Gutierrez, Oscar Plata*

Dept. Computer Architecture, University of Malaga, 29071 Malaga, Spain

Abstract

This work discusses the parallelization of an irregular scientific code, the transposition of a sparse matrix, comparing two multithreaded strategies on a multicore platform: a programmer-optimized parallelization and a semi-automatic parallelization using transactional memory (TM) support. Sparse matrix transposition features an irregular memory access pattern that depends on the input matrix, and thereby its dependencies cannot be known before its execution. This situation demands from the parallel programmer an important effort to develop an optimized parallel version of the code. The aim of this paper is to show how TM may help to simplify greatly the work of the programmer in parallelizing the code while obtaining a competitive parallel version in terms of performance. To this end, a TM solution intended to exploit concurrency from sequential programs has been developed by adding a fully distributed transaction commit manager to a well-known STM system. This manager is in charge of ordering transaction commits when required in order to preserve data dependencies.

Keywords: Software transactional memory, sparse matrix transposition, optimistic parallelization

1. Introduction

Today's commodity computers are providing to users an enormous computational power that needs to be exploited suitably. The ball is now in the court of the programmers, who must make a significant effort in order to let applications execute efficiently on the modern multithreaded architectures and chip multiprocessor (CMP) platforms [1]. In this context, Transactional Memory (TM) has emerged [2] as a powerful abstraction that allows an effective management of concurrency [3].

A transaction is defined as a piece of code that is executed isolated and with atomicity, being a replacement of a critical section but without the disadvantage of forcing a mandatory serialization. Transactions accessing shared memory data are speculatively executed and their changes are tracked by a data version manager. If two concurrent transactions conflict (write/write, read/write the same shared memory location), one of them must abort. After restoring its initial state, the aborted transaction is able to retry its execution. When a transaction finishes without aborting, it must commit, making its changes in memory definitive. It is said that version management is eager if changes are immediately translated into memory and a undo-log is used to restore the state of aborted transactions. By contrast, in a lazy version management, changes are stored in a write-buffer and not written in memory until

*Corresponding author. Tel.: +34-952-133318 ; fax: +34-952-132790.

Email addresses: magonzalez@ac.uma.es (Miguel A. Gonzalez-Mesa), eladio@uma.es (Eladio D. Gutierrez), oplat@uma.es (Oscar Plata)

commit takes place. As well, a transaction can abort just when the conflict is detected (eager conflict detection), or postpone the conflict checks until its end (lazy conflict detection). TM proposals can be found both in software (STM) and in hardware (HTM).

Encouraged by TM benefits, efforts are being devoted to the parallelization of sequential applications relying on TM. In addition to benchmark suites oriented to evaluate TM systems, like [4, 5, 6], research focused on legacy code parallelization by using TM approaches are found in literature [3, 7, 8, 9, 10]. Likewise, frameworks aimed at facilitating the writing of transactional codes in multiprocessors are also found [11, 12].

Concerning iterative computations, the efficiency obtained when parallelizing a loop lies in the correct resolution of dependencies. Many applications have regular control and memory access patterns inside loops, so that they can be statically (compile time) parallelized using fully/partially independent concurrent execution of iterations (DOALL/DOACROSS) [13]. In contrast, many other programs exhibit irregular cross-iteration dependencies that cannot be determined statically by the compiler. In this way, the optimistic concurrency exploited by TM systems may help parallelizing irregular loops. Loop iterations may be executed as concurrent transactions so as the system is in charge of tracking memory accesses in order to stall or abort conflicting transactions. However, some ordering constraints amongst transactions must be fulfilled to preserve cross-iteration data dependencies.

In this work, a low-overhead transactional memory support is proposed in order to exploit the loop parallelism with no data dependence information available before runtime. The proposal allows executing blocks of iterations in parallel whose data dependencies are preserved through a fully distributed transaction commit management in an eager/lazy (conflict detection/data versioning) system. Although this approach can be deployed regardless of the baseline TM system, our implementation is based on TinySTM [14]. Moreover, some improvements, such as data forwarding between transactions were added.

As a case study, a code for the sparse matrix transposition is analyzed and parallelized using our TM proposal. This code exhibits an irregular memory access pattern that makes it suitable for being written using TM abstraction, as dependencies are given by the input and they are not easily analyzable. This type of applications demands from the parallel programmer an important effort to develop an optimized parallel version of the code. The aim of this paper is to show how TM may help to simplify greatly the work of the programmer in parallelizing the code while obtaining a competitive parallel version in terms of performance.

In the next section, the features of the sparse matrix transposition code are discussed. Next, we present our transactional memory support for parallelizing loops with statically-unresolvable dependencies. In section 4 experimental results are presented and compared. Finally, conclusions are drawn in the last section.

2. Sparse Matrix Transposition

As case study and motivating application, the Sparse Matrix Transposition algorithm proposed by S. Pissanetzky [15] has been considered, which is one of the most efficient sequential algorithm for the sparse transposition. It works with matrices stored in the CRS (Compressed Row Storage) format [16]¹, which is a widely used space-efficient scheme for representing sparse matrices [17]. Sparse matrix transposition features an irregular memory access pattern that depends on the input matrix, and thereby, its dependencies cannot be known before its execution. In turn, this code can be useful as benchmark to test transactional systems as the dependence pattern can be tuned by selecting a given input matrix.

Fig. 1a depicts the sequential code using Fortran-like syntax. After an initialization phase, of complexity $O(Nrows + Ncols)$, where the row counters of the transposed matrix are computed, the transposition itself ($O(Nrows \times Ncols)$) takes place. Observe that the code contains complex subscript patterns with up to three in-direction levels in some LHS assignment sentences. In the general case, the dependencies between iterations will

¹A sparse matrix of size $Nrows \times Ncols$ with $Nzeros$ non-zero elements is represented by CRS using three linear vectors: $COLUMN[Nzeros]$, $DATA[Nzeros]$ and $ROW[Nrows + 1]$. $DATA$ contains, row by row, those values that are not null, $COLUMN$ contains the corresponding column index, and each element of ROW point to that index in $COLUMN$ where the list of non-zero elements of this row starts. For example, $\begin{bmatrix} 0 & a & b \\ c & 0 & d \\ 0 & 0 & e \end{bmatrix}$ will be stored as $ROW = [1, 3, 5, 6]$, $COLUMN = [2, 3, 1, 3, 3]$, $DATA = [a, b, c, d, e]$, considering all indexes starting at 1. Note that $ROW[Nrows + 1] = Nzeros + 1$.

```

1  c --- Initialization
2  do i=2, NCOLS+1
3    ROWT(i)=0
4  enddo
5
6  do i=1, ROW(NROWS)+1-1
7    ROWT(COLUMN(i)+2) = ROWT(COLUMN(i)+2)+1
8  enddo
9
10 ROWT(1)=1; ROWT(2)=1
11 do i=3, NCOLS+1
12   ROWT(i) = ROWT(i)+ROWT(i-1)
13 enddo
14
15 c --- Transposition
16 do i=1,NROWS
17   do j=ROW(i), ROW(i+1)-1
18     DATAT(ROWT(COLUMN(j)+1)) = DATA(j)
19     COLUMNNT(ROWT(COLUMN(j)+1)) = i
20     ROWT(COLUMN(j)+1) = ROWT(COLUMN(j)+1)+1
21   enddo
22 enddo

```

```

c --- Input matrix: A = {ROW, COLUMN, DATA}
c --- (Local) submatrix for thread id:
c ---   Alocid = {ROWlocid,
c ---             COLUMNlocid, DATAlocid}
c --- Transposed matrix: AT = {ROWT, COLUMNT, DATAT}
C$OMP PARALLEL
  id = omp_get_thread_num()
  Nthreads = omp_get_num_threads()
c --- Partition into submatrices
  call sp_partition(A, Alocid, Nthreads)
c --- Local transposition
  call transpose(Alocid, AlocidT)
C$OMP BARRIER
C$OMP MASTER
c --- Gather transposed submatrices
  call sp_join(AlocidT, AT, Nthreads)
C$OMP MASTER
C$OMP END PARALLEL

```

(a) Pissanetzky's sparse matrix transposition sequential code. (b) Data-parallel style parallelization pseudo-code.

```

C$OMP TRANSDO SCHEDULE(static, blockSize) ORDERED
  do i=1,NROWS
  do j=ROW(i), ROW(i+1)-1
    DATAT(ROWT(COLUMN(j)+1)) = DATA(j)
    COLUMNNT(ROWT(COLUMN(j)+1)) = i
    ROWT(COLUMN(j)+1) = ROWT(COLUMN(j)+1)+1
  enddo
  enddo
C$OMP END TRANSDO

```

(c) TM version using OpenTM-style notation.

Fig. 1: Sparse Matrix Transposition. Both the input and the transposed matrix are stored in CRS format.

be determined by the non-zero element distribution of the matrix to be transposed. In the case of a pure diagonal matrix, no dependencies at all would be present.

Analyzing the three sentences inside the double loop (lines 18—20) we can observe several sources of dependencies. First, a loop-carried dependence is caused by the subscript $ROWT(COLUMN(j) + 1)$ in the updating of vectors $DATAT$ and $COLUMNNT$. Both vectors are subscripted through vector $ROWT$ whose value can have been calculated in a different iteration from this one that is using it. A second loop-carried dependence is given by the reduction operation on $ROWT$ inside the transposition block (line 20), as such a value is read and written in the same sentences. And a third possible source of dependencies is the writing in positions of $DATAT$ and $COLUMNNT$. Because they are subscripted through subscripts, two different iterations could potentially write in the same memory location. Nevertheless, with further knowledge of the algorithm, this dependence is not relevant, as each position of $DATAT$ and $COLUMNNT$ is written only once in memory, because each element of the input matrix appears only once in the transposed. However, a ill-formed matrix in CRS format can break this assumption. Consequently a simple parallelization should guarantee atomic access to memory locations of vectors inside the loop and keep the iteration order during writings to a given position. This, in practice, would involve a high degree of serialization.

A well known approach to solve the parallelization of this code is by following a data-partitioning strategy [18]. The code in Fig. 1b sketches this possibility. Basically, the input sparse matrix needs to be partitioned into smaller submatrices, also sparse, that needs to be distributed among threads. In the figure it is assumed that there exists as many submatrices as threads. This partition can be as complex as desired (block, cyclic, ...) and it implies an index translation from the CRS structure of the input matrix to those of the submatrices. A simple partition

can consist of partitioning the matrix by rows, with all submatrices of the same size. In this case the partitioning procedure can be executed fully in parallel. As we are assuming a shared memory architecture, parallelism has been expressed in the figure by using OpenMP notation. After partitioning the input matrix, each thread invokes the sequential transposition applied to the submatrix or submatrices of which is in charge. In a final step, all transposed submatrices have to be gathered to get the transposed of the input matrix. Depending on the partitioning strategy, this step can be more or less complex, as different column data may need to be interleaved to reconstruct the original matrix structure. For example, in the simple case of a partitioning into contiguous equal-sized blocks of rows, transposed submatrices require to be concatenated by columns to get the right transposition. For this reason, and being conservative this last step has been marked as serial in the figure, although same amount of parallelism could be exploited.

As discussed, the amount of information about the problem can guide the programmer towards a more efficient solution when parallelizing codes with complex memory patterns, as is the case for sparse transposition. Transactional memory abstractions can make the programmer's work easier in these cases, because it allows exploiting concurrency optimistically without needing extra information. Fig. 1c shows how the programmer might codify the parallel version if a TM support that preserves data dependencies is available (OpenTM-style notation is used [19]). Note that the programmer effort to develop this version is much lower than in the previous data-parallel code, and with no deep knowledge about the application.

3. Design and Implementation

3.1. Baseline STM system

We have implemented our parallelizing approach on top of a well-known full software TM system (TinySTM [14]), a lightweight STM implementation, that provides the basic support for conflict detection and data versioning management. This STM system, like most word-based STMs, uses a shared array of fine-grain locks to manage concurrent accesses to memory, as well as a time-based approach to guarantee transactional consistency.

TinySTM implements an eager conflict detection policy that allows transactions to detect and solve memory conflicts as soon as they arise, minimizing the execution of useless work. This policy is implemented using a shared array of locks. Transactions set the least significant bit of each lock associated to each memory address to indicate the lock acquisition. To limit memory consumption, a hash function is used to map memory addresses to locks. However, collisions may be present in hash functions, involving that each lock may cover a set of different addresses (aliases). This situation is known as a false sharing problem, where accesses to different aliased addresses are mistakenly identified as conflicts.

Writing a memory location requires acquiring the corresponding lock during the operation. To do this, transaction maps the memory address to the corresponding lock and tries to set the least significant bit. If acquisition is successful, transaction becomes the owner (of the lock). If another transaction tries to write the same address, lock acquisition will fail, detecting a data conflict and aborting immediately one of them (generally, the newest one). Once transaction is the owner, transaction writes the new value using a write-through or a write-back approach. While write-through, or eager version management, update new values to memory and saves the older ones into a undo-log, the write-back, or lazy version management, design keeps memory untouched during the transaction execution. All memory addresses and the new values are stored in a local write buffer until commit time when new values will be updated to memory.

By contrast, reads do not require lock acquisition, but transactions must ensure that read addresses are valid. A time-based approach is used to guarantee transactional consistency. So, transactions and memory addresses are labeled with a version number provided by a shared counter used as clock. Checking the version associated to each memory address between the beginning and end of the read operation, transactions determine if the value read has been changed. If so, transaction will retry the read operation.

3.2. Customized STM system

We have added some key features to the baseline STM system in order to support speculative parallelization of loops. The first feature is the ordering of transactions. In general, applications present data dependencies that must preserve a strict sequential order execution to ensure the correctness of the final result. This is the case of

loop carried dependencies, where a loop iteration may require data computed in previous iterations. Frequently, these loops cannot be parallelized easily, especially when such dependencies are irregular or data-dependent. In many cases, the parallelization requires a good knowledge of the problem and/or a complex partitioning of the loop iteration space and corresponding data structures. In general, TM systems do not preserve order among concurrent transactions, so they cannot be used directly to parallelize loops with potential dependencies. In our customized STM, we add a lightweight technique for transaction coordination that allows an out-of-order transaction execution and commit, but imposing an strict commit order only when data dependencies are present (memory conflicts).

Including order over transactions may involve a performance degradation, due to each transaction waiting for commit their updates to memory is wasting a thread. If the waiting time is long (too many concurrent threads and data conflicts) the drop in performance can be important. This problem is associated to the ordering property. In order to alleviate the impact on performance, we developed in our customized STM a technique based on decoupling the transaction execution and commit phases, allowing threads launch new transactions while the previous ones are waiting for the right time to commit.

Finally, we also included in the customized STM a data forwarding support between transactions, so that a memory read inside a transaction can obtain a value written by other concurrent transaction before committing. A careful management of these forwarded data is included in order to not compromise data consistency.

To implement all the features discussed above, we selected a lazy version management scheme in the baseline STM system. This design choice is justified by the fact that an ordered STM system can not allow writing new data to memory, if not absolutely sure that correctness is maintained. The main problem is that non-transactional code could access such data without notice. Furthermore, keeping written values in a local write buffer, instead of in memory, facilitates decoupling the execution and commit phases. The out-of-order transaction commit must also comply with a single but important requirement. No transaction can commit until all previous transactions (in a sequential ordering) have finished their execution.

3.2.1. Creation of transactions

To explain how our customized STM works, we will focus on the parallelization of a loop with potential loop-carried data dependencies. To create transactions, the loop iteration space is partitioned, assigning a group of iterations to each transaction, and executing such transactions concurrently. By using an ordered STM system, the correct parallel execution of the loop is guaranteed.

The way iterations are grouped into transactions and how they are assigned to threads is important. For instance, a pure block grouping of iterations, that is, a block of consecutive iterations assigned to a single transaction, and a distribution of consecutive transactions amongst threads, may not be suitable. In such a case, a transaction cannot be committed until previous transactions (in loop order), assigned to different threads, finish their execution. Block grouping introduces a very unbalanced waiting-for-committing time, affecting parallel performance negatively.

This situation can be avoided if transactions are distributed amongst threads on a round-robin basis. In our customized STM system, transaction distribution uses this scheme. The basic grouping granularity is a single loop iteration, that is, all transactions are single loop iterations that are distributed cyclically across threads. However, to improve data locality, blocks of iterations (batches) could be considered to be grouped into transactions. Larger batch sizes will help to exploit more locality but the performance can be lower because larger transactions usually involve a higher number of aborts.

Choosing an optimal batch size is not an easy decision due to several issues that have a strong influence on the execution performance, such as the size of transactions, the overhead of the transactional system and the number of loop-carried dependencies. A good trade-off for the batch size should take advantage of the inter-iteration reference locality but without making transactions excessively large.

3.2.2. Tracking of data dependencies

During the concurrent execution of transactions, conflicts are detected by the TM system when reads and writes from different transactions are issued to the same memory location. That, in fact, corresponds to a data dependence. In an ordered STM system such as ours, we can determine the type of data dependence and how to proceed to assure a correct parallel execution. Possible cases are (previous and subsequent terms refer to the sequential loop ordering):

- *RAW dependence (flow dependence)*: This type of dependence occurs when a transaction reads a memory location written by another previous transaction. There exist two possible scenarios according to the timing ordering of the two memory operations. If the write operation occurs before the read operation, it is a true RAW dependence. This case can be solved using different approaches. While the baseline STM aborts and rolls back the reader transaction until writer transaction commits, our customized STM includes also the stalling of the reader transaction until writer transaction commits, avoiding aborting and repeating computation already done. We also include a data forwarding scheme from the writer transaction to the reader one, avoiding to stall that one. On the other hand, if the write operation occurs after the read operation, it is a false WAR dependence. This case cannot be solved in an ordered TM system without aborting the reader transaction because the read value becomes invalid when the previous writer transaction commits.
- *WAR dependence (anti-dependence)*: It occurs when a transaction writes to a memory location that has been read in a previous transaction. According to the timing ordering of the read and write operations, this dependence can be a true WAR dependence (read executes before write) or a false RAW dependence (write executes before read). However, in any case, this data dependence is always preserved in an ordered STM system, as the read operation occurs in the previous transaction. Invalid data will never be read because the writer transaction cannot commit (update memory) until all previous ones finish their execution.
- *WAW dependence (output dependence)*: This dependence occurs when two transactions write to the same location. This is a special situation because the ordering of write operations is not really important, as both save the new values in their local write buffer. The problem appears at commit time when both transactions update the memory. The value that must survive at memory is the written one by the last transaction in sequential ordering. As we explained, in our customized STM system transactions can commit only when all previous transactions have been executed (not really committed). This strategy allows transactions to commit out-of-order if there are no conflicts. Hence, in the presence of a WAW data dependence, we have to force a strict sequential commit order between the conflicting transactions in order to guarantee the correctness.

3.2.3. Management of transaction ordering

In our ordered customized STM system, each created transaction is given an order number according to the sequential loop ordering. This transaction order number is not only used for conflict resolution, as discussed previously, but also to maintain data consistency. As discussed previously, a transaction can only abort subsequent transactions. Hence, if all previous transactions have been executed, we can be sure that the current transaction cannot be aborted. In that case, that transaction can commit.

However, to implement this behaviour transactions require status information about other transactions. We use a lightweight mechanism that allows transactions to decide when the requirements for committing safely were reached. This mechanism is based on a shared table data structure, composed of as many rows as the numbers of threads and two columns by row. Each row is assigned to a different thread. The first column stores the number of executed (finished) transactions by the thread, and the second column stores the number of committed transactions. Each thread updates their own two counters while executing transactions. When a transaction reaches the commit point, the execution counter in the above table for the owner thread is incremented. After such an update, the thread tries to commit all owned finished but uncommitted transactions, one by one, starting from the oldest one. The commit condition consists of checking if $C_T \leq E_{T'} - 1, \forall T'$ previous to T and if $C_T \leq E_{T'}, \forall T'$ subsequent to T , where C_T is the value of the commit counter and E_T is the value of the execution counter. If both conditions are fulfilled, the oldest finished transaction can commit safely, C_T is incremented, and the commit conditions are again checked to determine if the next finished transaction can commit. This way, we ensure that data dependencies are preserved in a strict sequential order.

On the other hand, in section 3.2.2 we explained that true RAW dependencies can be solved using data forwarding. To implement this operation, transactions must notify each other. If a transaction reads a memory location written by another previous transaction, a forwarding request can be sent together with the memory address. Then, the writer transaction looks for it in its local write buffer and sends the value to the requester transaction. Finally, this transaction inserts the memory address in its read set. Note that we only permit data forwarding from a previous transaction to subsequent ones. Data forwarding allows transactions to carry on the execution optimistically.

However, it also presents a potential problem (cascade aborts): If the previous transaction is aborted (by another previous transaction), all subsequent transactions which received the forwarded data must also abort.

3.2.4. Checkpointing uncommitted transactions

As discussed previously, our customized STM allows decoupling the transaction execution and commit phases, allowing threads launch new transactions while the previous ones are waiting for commit. To permit this behaviour a checkpointing mechanism was implemented. When a thread finishes the execution of a transaction, it tries to commit it. If the committing condition is not fulfilled at that moment, the thread saves the basic state of that transaction (read/write sets and write buffer) and it launches a new transaction.

We use a small circular buffer per thread with a limited number of reusable slots to implement the checkpointing mechanism. Each slot saves the necessary information (a local copy of the read and write sets and the write buffer) of a finished but uncommitted transaction to support conflict detection and commit. Hence, at commit time, if a thread cannot commit the current executed transaction, the read and write sets and the write buffer are swapped out, and empty ones are set up to initiate a new transaction. This strategy enables a thread to launch new transactions although its previous transactions are not yet committed, which reduces the time spent in stall cycles and improves concurrency exploitation.

The optimal number of checkpoint slots, which the system must be able to track, is a loop-dependent parameter to be determined. It basically depends on the loop-carried dependencies of the particular loop. A large number of checkpoint slots involves a large number of uncommitted transactions that increase the conflict probability as well as the transactional overhead due to a large number of checkings of read and write sets. In general, we observed that a moderately small number of checkpoint slots is enough to achieve good performance. As an example, for the sparse matrix transposition application studied in this paper, the maximum number of used slots was three. Whenever all defined checkpoints are used up, the current thread must stall and wait until one of the finished transactions commits.

4. Experimental Evaluation









In this section, we evaluate the performance of our customized STM system against the baseline STM and the data-parallel version (Sec. 2) of the sparse matrix transposition code. All experiments were conducted on a server with quad Intel Xeon X7550 processors (32 cores in total) at 2GHz running Linux Kernel 2.6.32 (64-bits). In our experiments, we collected measurements about speedup, throughput and transaction commit rate (TCR) [20]. TCR is defined as the percentage of committed transactions out of all executed ones and is a suitable parameter that measures the exploited concurrency.

As mentioned before, the parallelization effort using the customized STM support is minimal, it would be only necessary a single directive (see Fig. 1c). The TM support handles the loop partitioning, the creation of transactions and their distribution among threads (Sec. 3.2.1). All the memory operations inside the loop are replaced by their transactional versions. The data-parallel version was coded using OpenMP.

A set of sparse matrices from Matrix Market repository [21] have been selected, covering a wide range of configurations (size, density, and structure). Table 1 summarizes the used test matrices. As explained in section 3.2.2, in presence of true RAW data dependencies, a transaction could require data from a previous one. Although we propose a data forwarding mechanism as an optimistic solution, other possibilities have been considered, such as stalling or aborting transactions (default mechanism in the baseline STM). These three options have been evaluated in terms of TCR. Sparse Matrix Transposition is not a computationally intensive application as it only performs data movements. To mitigate the effect of high overhead introduced by the baseline STM system, a synthetic workload has been added to the inner transposition loop. This computational load consists in a mathematical floating-point operation over each matrix element value, so that the matrix structure is unaltered. That is, the memory access patterns are preserved. Finally, to evaluate our proposal in presence of a large number of data conflicts, different transaction sizes have been tested by grouping more iterations in a transaction.

Figure 2 shows the measurement of TCR for both TM systems with the extra workload. Observe that our customized STM noticeably outperforms the baseline system for almost all matrices. Proposed modifications allow to exploit more concurrency and reduce the number of aborts regarding the baseline STM. Only for *bcsstk18*

Table 1: Sparse matrix characteristics

Matrix	Spy plot	Size	Non-zeros (Density)	Sym	Matrix	Spy plot	Size	Non-zeros (Density)	Sym
1138_bus		1138x1138	4054 (0,31%)	Yes	494_bus		494x494	1666 (0,68%)	Yes
bcsprw04		274x274	1612 (2,15%)	Yes	bcsprw10		5300x5300	21842 (0,08%)	Yes
bcsstk18		11948x11948	149090 (0,10%)	Yes	beacxc		497x506	50409 (20,04%)	No
jpwh_991		991x991	6027 (0,61%)	Yes	wm3		207x260	2948 (5,48%)	No

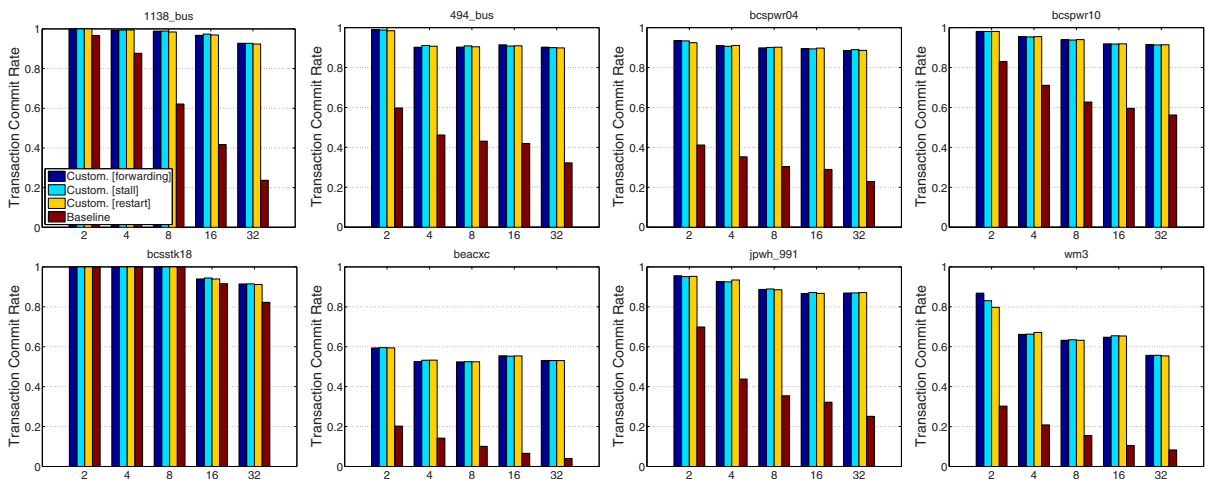


Fig. 2: Transaction commit rates on sparse matrix transposition for different numbers of threads.

TCRs are similar. While the baseline system exhibits poor scalability, reducing the concurrency about 45% on average (and up to 70% in *1138_bus*), our proposed STM scales better than the baseline STM. As expected, an increment in the number of transactions causes more data conflicts and aborts, which reduce the concurrency. However, this reduction is between 5% and 10% on average in our implementation. A particularly interesting case is *beacxc*. Although it is a relatively small matrix, it has high density. For this reason, the number of memory conflicts is high, limiting the concurrency. Even so, the achieved TCR is nearly 60%, while the concurrency of the baseline system is much lower. The three considered mechanisms to solve true RAW data dependencies perform similarly, because computational load is balanced in all cases (data forwarding does not provide a significant advantage). As a consequence, in the rest of the experiments we only consider the implementation with data forwarding. The introduction of synthetic extra workload in the loop does not have influence in the TCR because the concurrency is not a function of the computational load, but of the memory conflict pattern. Nevertheless, this extra workload will affect execution time measurements.

Fig. 3 shows the speedups considering the extra workload. Observe the disparity on performance that cannot be only attributable to the transactional overhead. In contrast to what we expected, the data-parallel version does not achieve the best performance for all matrices. In most cases it performs similar to our TM system. Our proposed system achieves better speedups for *494_bus* and *1138_bus* matrices. Only for *bcsprw10* and *bcsstk18* the data-parallel version outperforms the transactional system. This makes sense because matrix splitting and

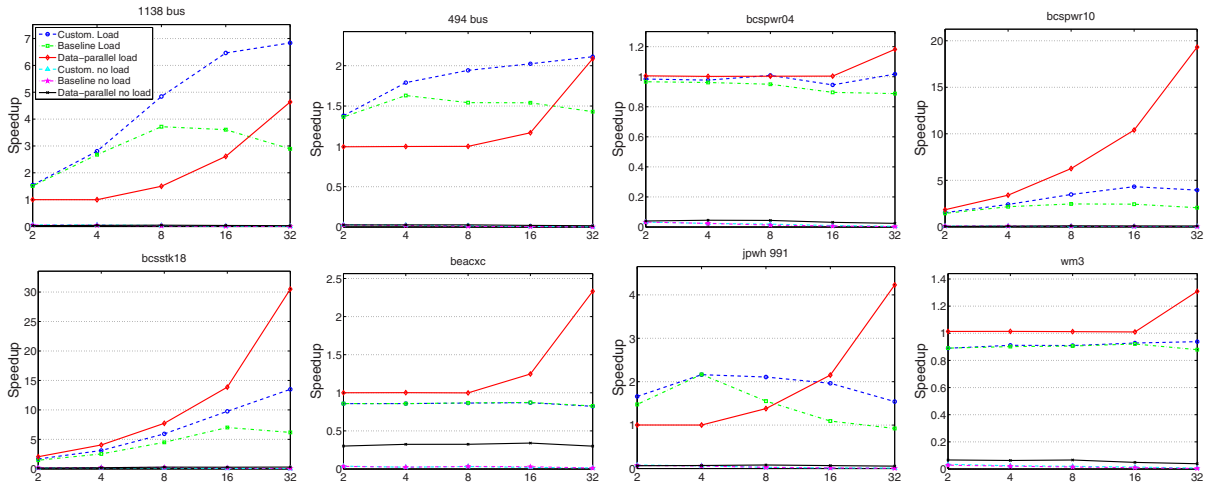


Fig. 3: Speedup over sequential code of the three parallelization schemes (customized STM, baseline STM and data-parallel) using both with and without extra workload for different numbers of threads.

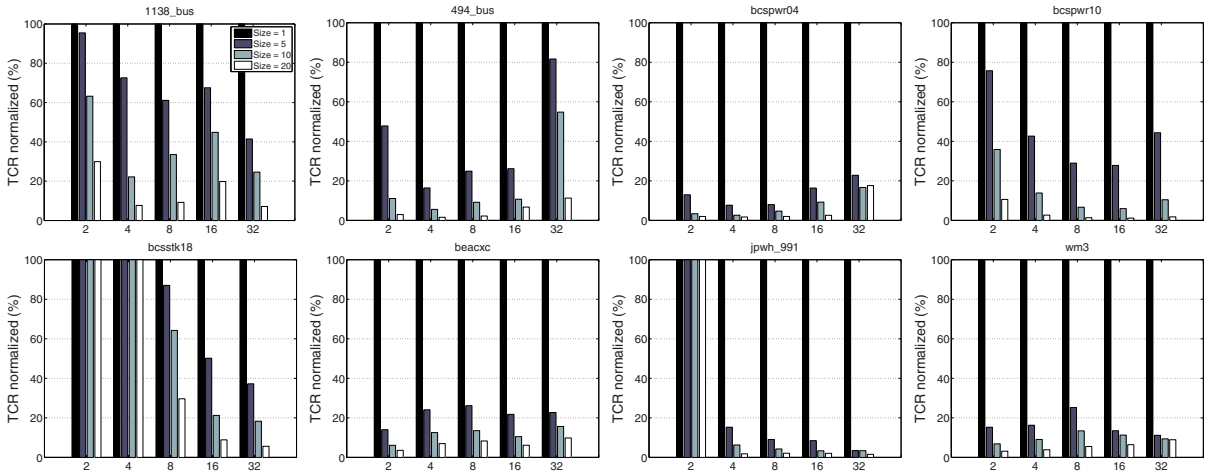


Fig. 4: Normalized TCR (in percentage related to the smallest TCR) for different transaction sizes and different numbers of threads.

merging stages present a significant overhead. However, for these matrices, the extra workload makes the main loop computationally more expensive, compensating this overhead. In all cases, our proposal presents better execution times than the baseline STM, because the number of transaction aborts has been significantly reduced, as shown by the previous TCR analysis. In general, our solution scales better, especially for large matrices. Note that speedups of all parallelization schemes stay close to 1 (bad scalability) for some matrices such as *bcsppwr04*, *beacxc* and *wm3*. Two common features characterize these matrices regarding the other ones: small size and higher density. In this case, the number of data dependencies increases significantly and it becomes a bottleneck that reduces the concurrency and hurts the performance. Removing the extra workload makes the STM overhead prevail over the parallel transposition execution time. Consequently, the parallel performance is very low in these experiments. A similar situation occurs for the data-parallel version.

Finally, the influence of the transaction size is evaluated. Fig. 4 shows the TCR normalized to this one for the smallest sized transaction (in percentage). Groups of 1, 5, 10 and 20 consecutive iterations per transaction are considered. In general, for all the tested matrices, the exploited concurrency falls down as the transaction

size increases. This behaviour is expected because the probability of data conflicts increases with the size of the transactions. Both, load and no-load configurations, exhibit a similar performance degradation because this metric only depends on the data conflict rate.

5. Conclusions and Future Work

It is known that many applications that exhibit complex memory access patterns require a good knowledge of the problem from the programmer and great efforts to develop efficient parallel versions. In this paper we show that transactional memory might help in obtaining parallel codes with similar performance but with much less demands on the programmer side. We have developed a software TM support that includes all features needed to speculatively parallelize loops preserving all data dependencies. Taking the sparse matrix transposition as a case example of a code with irregular memory access patterns, we compared a TM-based parallel version with an optimized data-parallel implementation. The TM version is much easier to develop and offers a comparable performance to the optimized version. We plan to implement our solution in a Hardware TM architecture, that can significantly increase the performance by getting rid of the software overhead. We also initiated research towards optimizing the management of uncommitted transactions to allow a faster checking of the read and write sets.

References

- [1] D. Geer, Industry trends: Chip makers turn to multicore processors, *IEEE Computer* 38 (5) (2005) 11–13.
- [2] J. R. Larus, R. Rajwar, *Transactional Memory*, Morgan & Claypool Publishers, USA, 2007.
- [3] M. Saad, M. Mohamedin, B. Ravindran, HydraVM: extracting parallelism from legacy sequential code using STM, in: *Proceedings of the 4th USENIX conference on Hot Topics in Parallelism*, USENIX Association, 2012, pp. 8–8.
- [4] M. Scott, M. Spear, L. Dalessandro, V. Marathe, Delaunay triangulation with transactions and barriers, in: *Workload Characterization, 2007. IISWC 2007. IEEE 10th International Symposium on*, IEEE, 2007, pp. 107–113.
- [5] M. Ansari, C. Kotselidis, I. Watson, C. Kirkham, M. Luján, K. Jarvis, Lee-TM: A non-trivial benchmark suite for transactional memory, *Algorithms and Architectures for Parallel Processing (2008)* 196–207.
- [6] C. Cao Minh, J. Chung, C. Kozyrakis, K. Olukotun, STAMP: Stanford transactional applications for multi-processing, in: *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization, 2008*.
- [7] V. Gajinov, F. Zyulkyarov, O. Unsal, A. Cristal, E. Ayguade, T. Harris, M. Valero, QuakeTM: parallelizing a complex sequential application using transactional memory, in: *Proceedings of the 23rd international conference on Supercomputing*, ACM, 2009, pp. 126–135.
- [8] S. Birk, J. Steffan, J. Anderson, Parallelizing FPGA placement using transactional memory, in: *Field-Programmable Technology (FPT), 2010 International Conference on*, IEEE, 2010, pp. 61–69.
- [9] K. Nikas, N. Anastopoulos, G. Goumas, N. Koziris, Employing transactional memory and helper threads to speedup Dijkstra's algorithm, in: *Parallel Processing, 2009. ICPP'09. International Conference on*, IEEE, 2009, pp. 388–395.
- [10] M. DeVuyst, D. M. Tullsen, S. W. Kim, Runtime parallelization of legacy code on a transactional memory system, in: *6th Int'l. Conf. on High Performance and Embedded Architectures and Compilers (HiPEAC'11)*, 2011, pp. 127–136.
- [11] M. Mehrara, J. Hao, P.-C. Hsu, S. Mahlke, Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory, in: *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'09)*, 2009, pp. 166–176.
- [12] C. Ferri, A. Marongiu, B. Lipton, R. Bahar, T. Moreshet, L. Benini, M. Herlihy, SoC-TM: integrated HW/SW support for transactional memory programming on embedded MPSoCs, in: *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, ACM, 2011, pp. 39–48.
- [13] R. Allen, K. Kennedy, *Optimizing compilers for modern architectures: A dependence-based approach*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.
- [14] P. Felber, C. Fetzer, P. Marlier, T. Riegel, Time-based software transactional memory, *IEEE Trans. on Parallel and Distributed Systems* 21 (12) (2010) 1793–1807.
- [15] S. Pissanetzky, *Sparse Matrix Technology*, Academic Press, 1984.
- [16] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, H. Van Der Vorst, *Templates for the solution of algebraic eigenvalue problems: a practical guide*, Vol. 11, Society for Industrial Mathematics, 1987.
- [17] F. Smailbegovic, G. Gaydadjiev, S. Vassiliadis, Sparse matrix storage format, in: *Proceedings of the 16th Annual Workshop on Circuits, Systems and Signal Processing*, 2005, pp. 445–448.
- [18] G. Bandera, E. Zapata, Data locality exploitation in algorithms including sparse communications, in: *Parallel and Distributed Processing Symposium., Proceedings 15th International*, IEEE, 2001, pp. 6–pp.
- [19] W. Baek, C. Minh, M. Trautmann, C. Kozyrakis, K. Olukotun, The OpenTM transactional application programming interface, in: *Parallel Architecture and Compilation Techniques, 2007. PACT 2007. 16th International Conference on*, IEEE, 2007, pp. 376–387.
- [20] M. Ansari, C. Kotselidis, K. Jarvis, M. Luján, C. Kirkham, I. Watson, Advanced concurrency control for transactional memory using transaction commit rate, *Euro-Par 2008–Parallel Processing (2008)* 719–728.
- [21] R. F. Boisvert, R. Pozo, K. Remington, R. F. Barrett, J. J. Dongarra, Matrix Market: A web resource for test matrix collections, in: *Proceedings of the IFIP TC2/WG2.5 working conference on Quality of numerical software: assessment and enhancement*, Chapman & Hall, Ltd., London, UK, UK, 1997, pp. 125–137.