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 multiprocesor (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), oplata@uma.es (Oscar Plata)

doi:10.1016/j.procs.2013.05.214

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 evalutate 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 multiprocesors 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 indirection levels in some LHS assignment sentences. In the general case, the dependencies between iterations will

[1]A sparse matrix of size $Nrows \times Ncols$ with $Nnzeros$ non-zero elements is represented by CRS using three linear vectors: $COLUMN[Nnzeros], DATA[Nnzeros]$ 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 $ROW[Nrows + 1] = Nnzeros + 1$.