A Generalized Transaction Theory for Database and Non-database Tasks

Armin Feßler and Hans-Jörg Schek

Institute of Information Systems
ETH Zentrum, CH-8092 Zürich, Switzerland
{fessler,schek}@inf.ethz.ch

Abstract. In both database transaction management and parallel programming, parallel execution of operations is one of the most essential features. Although they look quite different, we will show that many important similarities exist. As a result of a more careful comparison we will be able to point out that recent progress in database transaction management theory in the field of composite stack schedules can improve the degree of parallelism in databases as well as in parallel programming. We will use an example from numerical algorithms and will demonstrate that in principle more parallelism can be achieved.

1 Motivation

Parallel programming (PP) and database transaction management (DBTM) are two separate fields that both provide mechanisms for executing tasks in parallel. At a first glance, these two models look rather different. A more careful inspection, however, reveals much commun ground. Therefore we are interested in a more careful comparison: Does one of these provide more possible parallelism than the other? Can the one model benefit from the other?

These are the questions we want to answer in this paper. In the next section we compare PP with DBTM. It turns out that both models are, in fact, similar in that both control the flow of information in a parallel execution in way that equivalence to a sequential execution is ensured. In both areas there is one level of abstraction where a scheduler (in case of DBTM) or a parallizing compiler (in case of PP) is located and ensures correctness. However, recent progress made in the foundation of database transactions shows that a higher degree of parallelism can be achieved be considering several schedulers at several layers of abstraction. Therefore, we contribute to the comoarison between PP and DBTM and we elaborate the idea of htransforming a single non-DB task into several artificial DB transactions with the aim of increasing the degree of parallelism in PP, too.

Section 2 compares PP and DBTM in more detail. In order to ensure readability, in Section 3 we explain informally the main insights from the theory of schedulers working at several layers of abstraction ("stack schedules"). Finally, in Section 4, we show how this theory could be adopted for the parallel execution of numerical algorithms as an example of a problem that is considered a non-database task.

P. Amestoy et al. (Eds.): Euro-Par'99, LNCS 1685, pp. 459–468, 1999.
© Springer-Verlag Berlin Heidelberg 1999

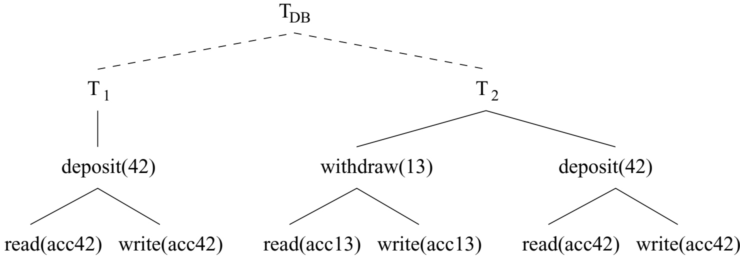

Fig. 1. Example for transaction invocations in a dbs

We are not aware of other work having considered comparisons between PP and DBTM or having applied multi-level transactions to non-DB tasks. In our own previous work [1] we have presented this idea the first time. The paper here gives a more detailed presentation: It contains a comparison between PP dependences and DBTM conflicts and elaborates on the applicability of the DBTM theory using several simple examples from matrix/vector computations. More details are available in [4].

2 Parallelization in Databases and in Parallel Programming

2.1 High-Level Comparison

Let us use simple examples for an overview of database transactions on the one hand, and numerical algorithms on the other hand.

First, we consider database transactions as those shown in figure 1. Traditionally, transactions (like T_1 and T_2) are given to the database system (DBS) – usually by different users. There, each transaction starts operations. Transaction T_1 starts a deposit operation on account number 42, which itself starts a read and a write operation. T_2 – a transfer transaction – starts an operation that withdraws some money from account number 13 and deposits it on account number 42. Each of these operations start a read and a write operation. Now, the scheduler of the DBS has to decide which operations can be executed in parallel, based on the information whether operations of different transactions commute. Since there are two layers of abstraction, there are also two possibilities for placing a scheduler: If it is placed at the upper layer, the scheduler decides on commutativity of the operations of T_1 and T_2. Deposit operations are always commutable. Withdraw commutes with each of the two deposit operations being on another account. If the scheduler is placed at the lower layer (this is the usual layer where DB schedulers work in practice), the scheduler decides on commutativity of the read and write operations as usually. In any case, the scheduler orders pairs of conflicting operations and makes sure that there exists always a sequential execution that orders conflict pairs the same way.

It seems to be a major point that usually transactions are independent and are entered by different clients. However there is nothing wrong if we assume that both transactions (e.g. T_1 and T_2 in our example) are entered by one and the same client, and still the scheduler will parallelize them considering the conflicts between their operations. We therefore can always think of an (artificial) transaction T_{DB} starting all client transactions like T_1 and T_2 in our example. The reason for this "trick" is that DB transactions look more similar to a numerical task that usually is entered by one client.

We now consider the essentials of parallel programs using a simple example. In figure 2 there is an invocation hierarchy of the numerical task T that calculates the sum of two matrices and a matrix vector product as follows:

```
B = A + B;
u = B v;
```

T invokes subprograms (possibly a simple loop) that calculate logical units of work – the sum $B = A + B$ and the matrix-vector product $u = Bv$. The subprograms invoke operations calculating the single elements of B and u. The main point here is that the algorithm T itself is similar to T_{DB} in DBTM. As in DBTM, T invokes subtasks that, in turn, invoke operations. In PP we distinguish two cases: Either the programmer decides whether operations can be executed in parallel using PP languages [3], or a parallelizing compiler (also called super-compiler [7]) transforms a given sequential program into a PP. The programming constructs and outcomes of both methods are the same, so we do not discriminate them in the following. In High Performance Fortran there are two types of parallel loops, `FORALL` and `INDEPENDENT DO`, in the example:

```
INDEPENDENT DO i=1, m
    DO j=1, n
        b(i,j) = a(i,j) + b(i,j)
        u(i) += b(i,j) * v(j)
    END DO
END DO
```

The outer loop is a parallel loop which executes its code independently for every i (the row index of B). The inner loop is a sequential loop over the column index j of B. On a high level, we now see that PP and DBTM are very similar. In both areas, a task is divided into subtasks, which possibly start subtasks themselves. But how does the parallel compiler decide which tasks can be parallelized and which not? Does PP have a notion similar to conflict pairs?

2.2 Dependences in Parallel Programs

A parallelizing compiler realizes what (sub)tasks can be executed concurrently by looking at data dependences. It ensures that all dependences of the (sequential) input program are kept in the produced parallel program. Three kinds of dependences are distinguished: flow-, anti-, and output dependence. These are easy to understand by observing that they are essentially the same as the three kinds of conflicts in transaction processing of the read/write model of DBTM.

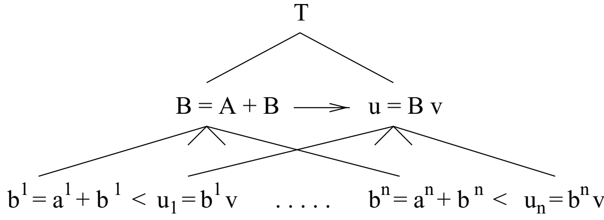

Fig. 2. Interleaved execution of ordered operations within a task

As a thorough analysis proves, the following counterparts between DBTM and PP notions can be found:

Parallel Programming	Database Transactions
variable	data-object
$x = y + z$	r(y) r(z) w(x)
flow dependence	wr-conflict
anti dependence	rw-conflict
output dependence	ww-conflict

In essence, the statement $x = y + z$ in PP is nothing else than reading y and z, adding them, and writing the result into the object x in DBTM terminology. A flow dependence is a type of a data dependence, when a variable is set and then read, in the example $x = 1; y = x;$. Anti dependence is the case, when a variable is read and then set, e.g., $y = x; x = 1;$. Output dependence appears, whenever a variable is set by two operations, e.g., $x = 10; x = 3;$. The order cannot be reversed without changing the result. Obviously, the three cases correspond to a write/read-, read/write-, and write/write-conflict in DBTM, resp.

Traditionally, in both PP and in DBTM, there is only one level, on which tasks are scheduled. As a major departure from this one-layer-scheduling, in DBTM it came out that parallelism can be gained by introducing additional layers of abstraction and allowing a scheduler at every level of abstraction. In our simple introductory example a higher-layer scheduler will not order any pair because there is no conflict between the operations: The two deposits commute since deposits always commute, and `deposit(42)` commutes with `withdraw(13)` because it is another account. However, if the intermediate level of the deposit and withdraw operations would be eliminated to achieve a traditional one-layer scheduler, the execution in the figure would not be correct any more. In the following we will informally explain this essential extension to DBTM theory, the formal definitions of which can be found in [1].

3 Stack Schedules

When executing transactions, a scheduler restricts parallelism because it must, first, observe the order constraints between the operations of each transaction

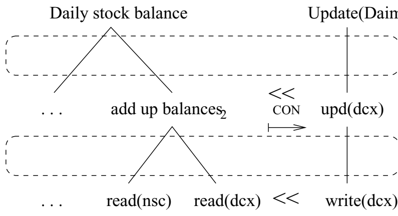

Fig. 3. Restr. parallelism (strong order)

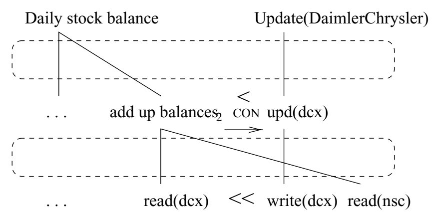

Fig. 4. Restr. parallelism (weak order)

and, second, impose order constraints between conflicting operations of different transactions. The restriction in parallelism occurs because, in a conventional scheduler, ordered operations are executed sequentially. As shown in [1] and as we explained just above, this is too restrictive. But parallelizing conflicting operations over several levels requires to relax some of the ordering requirements of traditional schedulers. In addition, a mechanism is needed to specify to a scheduler what is a correct execution from the point of view of the invoking scheduler. For these two purposes we use the notion of *weak* and *strong* orders:

Strong and Weak Order: A and B are any tasks (actions, transactions).

- Sequential (strong) order: $A \ll B$, A has to complete before B starts.
- Unrestricted parallel execution: $A \| B$, A and B can execute concurrently equivalent to any order, i.e., $A \ll B$ or $B \ll A$.
- Restricted parallel (weak) order: $A < B$, A and B can be executed concurrently but equivalent to executing $A \ll B$. □

To have a closer look at this concept, let us consider an example. In figure 3 there are two schedulers, one above the other. On the higher level there are two transactions: One calculates the daily stock balance by invoking subtransactions that calculate certain types of stocks each, possibly on different servers. Here, it suffices to consider only one of these subtransactions, `add up balances`$_2$, which reads the prices of two stocks, DaimlerChrysler, `dcx`, and netscape, `nsc`.

The other transaction updates the price of the DaimlerChrysler stock. The upper schedule knows that these two subtransactions, `add up balances`$_2$ and `upd(dcx)`, are in conflict. In the case of a conflict, traditional theories sequentialize the regarding operations, which we call a strong order. Because of this strong order, all operations of `add up balances`$_2$ are executed before the operation of `upd(dcx)`. Here is the main point that we lose parallelism as `read(nsc)` is strongly ordered before `write(dcx)`.

To prevent a loss of parallelism we make now use of the weak order (figure 4). Conflicting operations are ordered, but only weakly. This weak order is a constraint for the lower-level scheduler: Its serialisation graph has to follow the given weak order. Therefore `write(dcx)` can be executed in parallel to `read(nsc)` or in any arder as long as `write(dcx)` is executed after `read(dcx)`.

3.1 Transactions, Schedules

The distinction between weak and strong order require slight changes in the definitions of traditional notions. A **transaction** is a set of operations with partial strong AND weak order constraints between the operations. A **schedule** receives as input transactions and a weak and a strong input order. The output of a schedule consists of the set of all operations of all input transactions and and of the weak and strong (output) orders. Commutativity of operations is globally given by a conflict predicate. A schedule must weakly order every pair of conflicting operations from different transactions without contradicting the weak input order. Furthermore, all weak and strong transaction orders are contained in the weak and strong output orders, respectively. Naturally, strong input order are propagated from the transactions to their operations, thereby separating the execution tree of strongly ordered transactions.

3.2 Correct Schedule

The distinction between the two orderings requires also to modify the traditional notion of correctness. We will assume, as usual, that a transaction executed in isolation is correct.
A schedule S is correct or conflict consistent (CC)*, if there is a serial schedule, whose strong input and weak output order contain the weak input and output order of S, resp.*
The extension to the traditional theory may become clearer when we use the serialisation graph defined as in the classical theory. We have shown (proof see [2]):
A schedule is conflict consistent iff the union of its weak input order and its serialisation graph is acyclic.

3.3 Stack Conflict Consistency

With these preparations we are able to explain **stack schedules**: in a stack of schedules the output of one schedule is directly used as input to the next. Every level has a scheduler S that provides a set $\hat{O}_S$ of operation invocations to be used to build transactions. I.e., an operation of a scheduler can be a transaction of the next lower level scheduler. Every scheduler S has a commutativity specification expressed by the conflict predicate CON_S. Every scheduler works locally ensuring correctness with respect to its (local) CON_S. Fortunately we can prove [2] the following **theorem**:
An n-level stack schedule SS is correct, iff each individual schedule S_i in SS is conflict consistent, for $1 \leq i \leq n$.
We will use SCC as a shorthand for correctness of a stack schedule. Thus, every scheduler, except the lowest one, produces an executable plan for the next lower scheduler. The lowest scheduler, however, has to execute the operations it produces, thereby changing all weak output orders into strong ones. If higher-level schedulers make this change, the produced output is correct, too, but the achievable degree of parallelism is reduced.

In figure 4, the lower level schedule is conflict consistent, since there is a corresponding serial schedule, which is the lower level schedule in figure 3.

4 Parallelization of Numerical Algorithms

4.1 Application to the Introductory Example

SCC provides a larger class of correct executions, and introduces a possibility for parallelism which was not possible neither in previous DBTM nor in PP. This comes from perhaps the most interesting aspect of SCC: the possibility of executing operations in parallel, even if they conflict. This raises the possibility of implementing a *parallel-do* operation. By parallel-do we mean that the operations specified are executed in parallel, while preserving imposed serialisation dependences between them. In the example shown in figure 2, the programmer will specify something like:

Begin_Parallel_Do

$(B = A + B) \ \rightarrow \ (u = Bv)$

End_Parallel_Do

thereby indicating that they should be executed in parallel while preserving the given ordering. Now, is the execution shown in the figure correct? Obviously, the upper level schedule is CC. In the lower schedule, every conflict pair $(b^i = a^i + b^i, u_i = b^i v)$ is ordered in the same way. So, there is a serialisation graph edge from $B = A + B$ to $u = Bv$, consistent with the weak input order. Thus, the lower schedule is CC, and hence, the stack is SCC and correct.

4.2 Application to a Complex Example

Traditional Strategies We move to a more complex example and show what we can gain by the approach. Consider the first step of the Gauss-algorithm for $n = 3$. This requires $A = C_1 A,\ A = C_2 A$ with:

$$A = \begin{pmatrix} a^1 \\ a^2 \\ a^3 \end{pmatrix}, \quad C_1 = \begin{pmatrix} 1 & 0 & 0 \\ -\frac{a_{21}}{a_{11}} & 1 & 0 \\ -\frac{a_{31}}{a_{11}} & 0 & 1 \end{pmatrix} = \begin{pmatrix} c_1^1 \\ c_1^2 \\ c_1^3 \end{pmatrix}, \quad C_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -\frac{a_{32}}{a_{22}} & 1 \end{pmatrix} = \begin{pmatrix} c_2^1 \\ c_2^2 \\ c_2^3 \end{pmatrix}$$

In parallel programming and parallelizing compilers, this can be parallelized using two parallel and one sequential loops:

```
DO j=1, n-1
    INDEPENDENT DO i=j+1, n
        l(i,j) = a(i,j)/a(j,j)
        INDEPENDENT DO k=j, n
            a(i,k) = a(i,k) - l(i,k) * a(j,k)
        END DO
    END DO
END DO
```

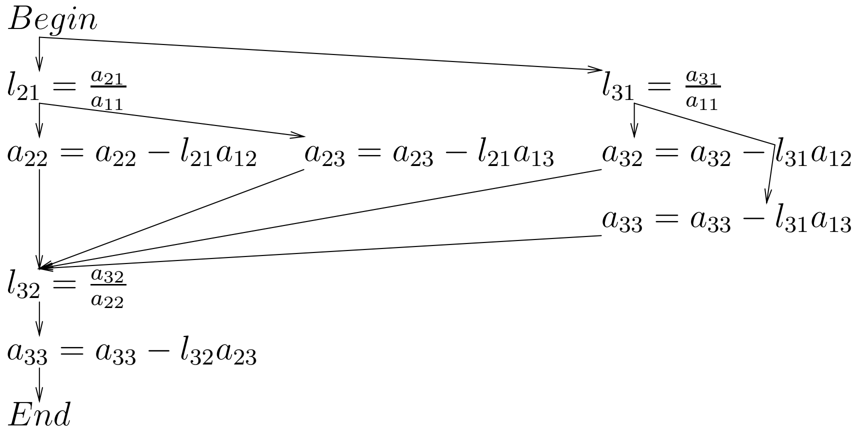

Fig. 5. Parallelization of a nested loop

However, the dependence graph for this code fragment (figure 5) shows that there are unnecessary operation orders. It does not show the data dependences, but the control dependences of the parallel loop, i.e., the data dependences are coded with the control dependences. As parallelizing compilers can only parallelize loops as a whole, the above outer loop is a sequential one. Therefore, the operation $l_{32} = \frac{a_{32}}{a_{22}}$ is executed after $a_{33} = a_{33} - l_{31}a_{13}$, although there is no data dependence between these two operations. If we make the simplifying assumption that every operation is executed in one time step, and three processors are available, these unnecessary orderings force a total computation of five steps.

Parallelizing Using Stack Conflict Consistency The idea consists in introducing compilers on several layers by decomposing a numerical algorithm into steps and considering them as transactions that run concurrently, observing some given external partial order. Each step, in turn, is decomposed into simpler steps that are considered as operations in the DBTM terminology.

Let us consider what exactly has to be done to be able to apply stack conflict consistency to our example. First, the whole computation is defined as a single transaction T consisting of operations $T_1 : A = C_1 A$ and $T_2 : A = C_2 A$, which is the algorithm in matrix form (figure 6[1]). Since order matters between T_1 and T_2, they are weakly ordered. The interesting part is on the lower scheduler. There, not all operation pairs from different (weakly ordered) transactions are conflicting. Therefore, some of those operations can be executed in parallel: E.g., T_{123} can be executed concurrently with T_{211}, although their parents T_{12} and T_{21} are (weakly) ordered. Since this order is only weak, it does not matter if non-conflicting operations are reversed. This type of parallelism does not exist in conventional PP.

[1] To make the presentation clearer, only (weak) input orders are shown

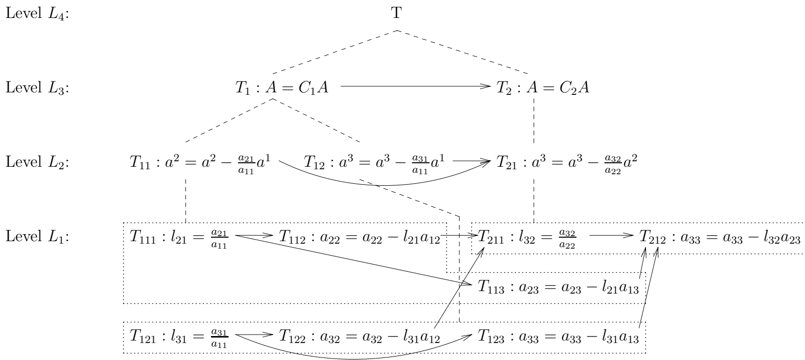

Fig. 6. Interleaved execution of ordered steps of a parallel program.

Now, is the schedule shown in figure 6 stack conflict consistent? We only have to prove whether every schedule of it is CC. The upmost schedule (level $L_4 - L_3$) certainly is conflict consistent, as there is only one transaction. The next schedule (level $L_3 - L_2$) consists of two transactions, T_1 and T_2, connected by a weak input order. Since this input order is consistent with the serialisation graph (T_{11} and T_{12} before T_{21}, resp.), this schedule is also correct. The lowest schedule (level $L_2 - L_1$) is the most complicated. There are two weak input orders (from transactions T_{11} and T_{12} to T_{21}, resp.). As one can prove, all conflicting operation pairs are weakly output ordered (we used here the same symbol as for the input order). All these weak output orders, however, are according to the weak input orders of their parents. For example, T_{112} is ordered before T_{211}, which is consistent with $T_{11} \rightarrow T_{21}$, the weak input order of their parents.

By summing up this example, we find that a stack schedule is similar to conventional parallelizing compilers on several layers of abstraction. Such compilers parallelize loops as a whole by determining if there are any data dependences contradicting the parallelization of this loop. We propose to have several layers of compilers working together like a stack schedule; a higher-level compiler gets the complete (sequential) source code and produces an execution plan, which is the input for next lower schedule that, in turn, produces an execution plan for the next scheduler. Only one scheduler of a stack, the lowest one, is actually executing its operations, by obeying the plan it produces and by only using strong output orders. In order to define parallel loops, every scheduler can make use of a new sort of loop: The *Parallel-Do with a weak order.*

What do we gain by applying stack scheduling to this example? In the last subsection, it came out that a parallelizing compiler would need five time steps to execute this program, given that three processors are available. As we can see in figure 6 only four steps are now necessary to execute all operations of level L_1 (which is the only level on which real computations are done)[2]. It is

[2] Note that the time axis is from left to right, as usual in DBTM.

expected that the profit of one time step in this scenario increases in the number of dimensions of the involved matrices, and in the complexity of the algorithm.

4.3 Other Approaches

There is a number of previous approaches to coping with multilevel schedules, the most advanced of which is Weikum's model. Weikum proposes a weaker condition than order preservation for independent scheduling in composite systems. This condition forces conflicting operations at a non-leaf level to have conflicting descendants at all lower levels (axiom 1 in [5]). This restriction, though often natural, restricts the scope of Weikum's model. E.g., it does not hold for multiversion CC algorithms, or in our context the addition of sparse matrices [4].

In level-by-level serialisability [5], when two operations conflict (that is, they are ordered by the weak execution order), they must also be strongly ordered. Consequently, the execution trees of conflicting operations cannot be interleaved. This is not the case for stack conflict consistency.

A rule-based approach to (partly) nested transactions are ULTRA transactions [6]. Similar to parallelizing compilers, operations are collected in an evaluation phase, and performed in a subsequent materialization phase. The ULTRA system executes basic updates simultaneously with the logical evaluation of the transactions. This optimistic method requires the possibility of compensation.

5 Acknowledgement

We would like to thank Gustavo Alonso for many fruitful discussions and the anonymous referees for their valuable comments.

References

[1] G. Alonso, S. Blott, A. Fessler, and H.-J. Schek. Correctness and parallelism in composite systems. In *Proc. of the 16th Symp. on Principles of Database Systems (PODS'97)*, Tucson, Arizona, May 1997.

[2] G. Alonso, A. Feßler, G. Pardon, and H.-J. Schek. Transactions in stack, fork, and join composite systems. In *7th International Conference on Database Theory (ICDT)*, Jerusalem, Israel, Jan. 1999.

[3] S. Brawer. *Introduction to Parallel Programming.* Academic Press, 1989.

[4] A. Feßler. *Eine verallgemeinerte Transaktionstheorie für Datenbank- und Nichtdatenbankaufgaben (to appear).* Dissertation, Department of Computer Science, ETH Zürich, 1999.

[5] G. Weikum. Principles and Realisation Strategies of Multilevel Transaction Management. *ACM Transactions on Database Systems*, 16(1):132, March 1991.

[6] C.-A. Wichert, A. Fent, and B. Freitag. How to execute ULTRA transactions. Technical Report MIP-9812, Universität Passau (FMI), 1998.

[7] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* Addison-Wesley, 1991.