Konfliktkonsistente Serialisierbarkeit für hohe Parallelität in Datenbanken und bei numerischen Algorithmen

Armin Feßler*

Zusammenfassung

Wir untersuchen die Möglichkeiten, die Stapelschedules im Vergleich zu herkömmlicher Datenbanktransaktionsverwaltung bieten. In Stapelschedules lassen sich die Serialisierungsordnungen der einzelnen Schichten vorgeben, was im allgemeinen höhere Parallelität zulässt. Dies verifizieren wir am Beispiel numerischer Algorithmen, deren klassische Korrektheitskriterien wir zunächst mit denen der Datenbanktheorie vergleichen.

1 Ausführungsordnungen in Datenbanken

Ausgehend vom Read/Write-Modell betrachten wir zunächst die herkömmlichen Ausführungsarten der sequentiellen und der parallelen Ausführung [BHG87, VGH93]. Dann zeigen wir im Mehrschichtentransaktionsmodell mit semantisch reichen Transaktionen, dass diese beiden Möglichkeiten nicht erschöpfend sind. Wir führen einen dritten Begriff ein, der die in Mehrschichtenschedules zusätzlich möglichen Ausführungsordnungen beschreibt, welche nicht von den Konzepten der Sequentialität und Parallelität abgedeckt wird [ABFS97]. Solche Ausführungsordnungen erfassen das korrekte Zusammenarbeiten mehrerer Scheduler. Die Anwendungen reichen von Mehrschichtentransaktionen [BBG89, Wei91] über verteilte Systeme [OV91, CGM88] bis zu numerischen Algorithmen. In diesem Papier beleuchten wir letzteren Aspekt.

1.1 Konventionelle Ausführungsordnungen

In der konventionellen Datenbanktheorie, genauer in der Transaktionsverwaltung, kannte man bis vor Kurzem nur die folgenden zwei Arten paralleler Ausführung.

Sequentielle Ausführung: Wenn zwei Operationen oder zwei Transaktionen sequentiell ausgeführt werden, wird die zweite Operation bzw. Transaktion erst dann begonnen, nachdem die erste beendet worden ist. Sequentielle Ausführungsordnungen bestehen natürlicherweise zwischen allen Operationen/Transaktionen, die in dieser rein zeitlichen Beziehung zueinander stehen. Darüberhinaus gibt es Fälle, in denen eine sequentielle Ausführung *gefordert* wird:

Operationen, die miteinander in Konflikt stehen, müssen sequentiell ausgeführt werden. Im ersten Beispiel von Abbildung 1 stehen die beiden Write-Operationen in Konflikt und müssen sequentialisiert werden, auch wenn sie nicht durch die Intratransaktionsordnung geordnet sind. Eine sequentielle Ausführungsordnung kann auch zwischen Operationen verlangt werden, die nicht miteinander in Konflikt stehen (*(Intra-)Transaktionsordnung*). Im zweiten Beispiel von Abbildung 1 müssen die beiden Operationen nur aufgrund der Intratransaktionsordnung in der

*Institut für Informationssysteme, ETH Zentrum, CH-8092 Zürich, fessler@inf.ethz.ch

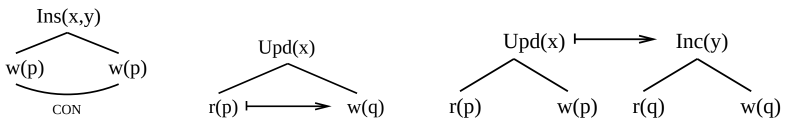


Abbildung 1: Sequentielle Ausführung von Operationen und Transaktionen

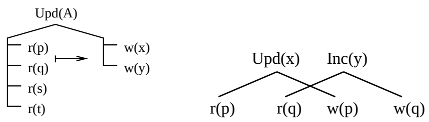


Abbildung 2: Parallele Ausführung von Operationen und von Transaktionen

spezifizierten Reihenfolge sequentialisiert werden. Überdies ist es in der traditionellen Transaktionsverwaltung nicht möglich, Transaktionen sequentiell zu ordnen. Im dritten Beispiel von Abbildung 1 ist Upd(x) durch die sogenannte *Intertransaktionsordnung* vor Inc(y) geordnet, weshalb alle Operationen von Upd(x) vor allen Operationen von Inc(y) ausgeführt werden müssen.

Parallele Ausführung: Nicht sequentiell ausgeführte Operationen oder Transaktionen nennt man parallel ausgeführt. Eine parallele Ausführung zweier Operationen wird als korrekt angesehen, wenn beide möglichen sequentiellen Ausführungen die gleichen Resultate liefern würden. Somit kann beispielsweise eine Schreiboperation $w^1(x)$ nie parallel mit einer Leseoperation $r^2(x)$ auf demselben Objekt ausgeführt werden. Im ersten Beispiel von Abbildung 2 werden alle Read-Operationen der Transaktion Upd(A) parallel und vor allen ihren Write-Operationen geordnet ausgeführt. Eine parallele Ausführung zweier Transaktion hingegen ist korrekt, wenn sie die gleichen Resultate wie eine der beiden möglichen sequentiellen Ausführungen ergibt. Im zweiten Beispiel von Abbildung 2 werden die beiden Transaktionen Upd(x) und Inc(y) parallel ausgeführt, indem ihre Operationen lediglich „verzahnt“ werden.

1.2 Eingeschränkt parallele Ausführung

In Mehrschichtenschedules machte man die Beobachtung, dass insgesamt mehr Parallelität zu erzielen ist, wenn man eine sogenannte *schwache Ordnung* einführt. Diese schwache Ordnung schreibt lediglich die Serialisierungsordnung von Transaktionen vor. Im ersten Beispiel von Abbildung 3 werden die beiden Transaktionen Sal-Bonus() und GetNewAverageSal() parallel ausgeführt, aber ihre Serialisierungsreihenfolge entspricht der spezifizierten schwachen Ordnung, da alle $Upd(s_x)$ vor den jeweils konfligierenden $Read(s_x)$ ausgeführt werden. Der Effekt dieser Ausführung ist also derselbe wie der einer seriellen Ausführung Sal-Bonus()–GetNewAverageSal() wie in derselben Abbildung (rechts) gezeigt. Die schwache Ordnung stellt nun kein zusätzliches Korrektheitskriterium auf, sondern nutzt nur das vorhandene der Konfliktserialisierbarkeit besser aus. Ein einzelner Schedule eines solchen Mehrschichtenschedules, jetzt *Stapelschedule* genannt, ist also korrekt, wenn er konfliktserialisierbar ist. Außerdem muss er die vorgegebene Serialisierungsordnung, i.e. die schwache Ordnung, und wie schon in bisherigen Modellen [Wei91, WS92] die vorgeschriebene Sequentialisierungsordnung, jetzt *starke Ordnung* genannt, einhalten, was zum folgenden Kriterium führt.

Konfliktkonsistenz (CC): Ein Schedule S ist *konfliktkonsistent*, wenn er zu einem seriellen Schedule konfliktäquivalent ist, der die schwache und starke Ordnung von S respektiert. In der formalen Definition wird Konfliktäquivalenz von S und S' jedoch nicht explizit gefordert, da

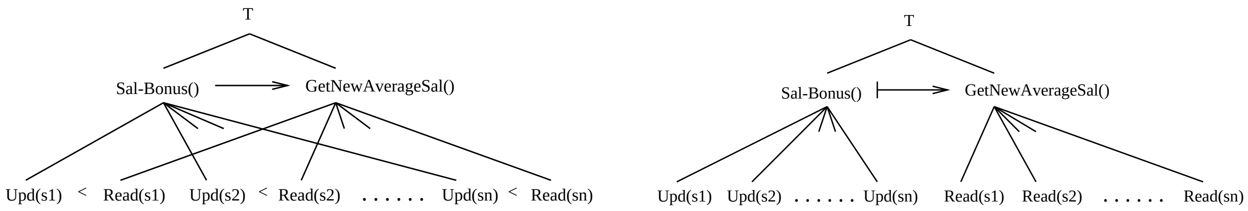
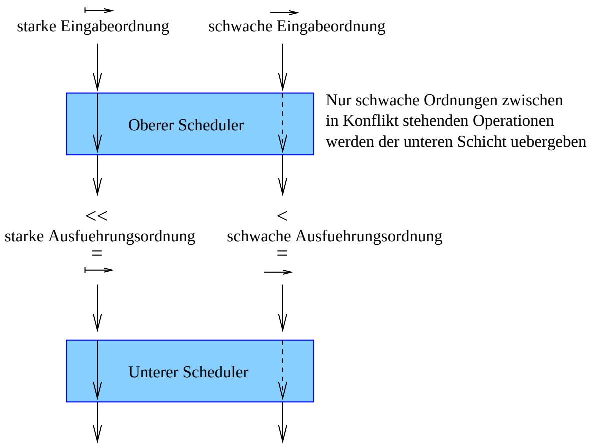


Abbildung 3: Schwach geordnete parallele Ausführung von TA und entsprechende Serialisierung



Parallele Programmierung	DB-Transaktionen (flach)
Variable	Datenobjekt
x = y + z	r(y) r(z) w(x)
Flow-Dependence	wr-Konflikt
Anti-Dependence	rw-Konflikt
Output-Dependence	ww-Konflikt

Abbildung 4: Übergabe von Ordnungen an tiefere Schedules und Vergleich der Begriffe

sie sich in der Inklusion der schwachen Ausführungsordnungen versteckt. Wir fordern nämlich, dass je zwei in Konflikt stehende Operationen eines Schedules mindestens schwach geordnet sind. Wann ist jedoch der gesamte Stapelschedule korrekt?

Stapel-Konfliktkonsistenz (SCC): Man kann zeigen, dass ein Stapelschedule SS korrekt ist, genannt *stapel-konfliktkonsistent*, wenn jeder Schedule S_i von SS konfliktkonsistent ist. Die Weitergabe der schwachen/starken Ausführungsordnungen eines Schedules von SS an den unmittelbar darunterliegenden Schedule als Eingabeordnungen (siehe Abbildung 4, links) gewährleistet dabei, dass die Serialisierungsordnungen auf allen Ebenen (Schedulern) jeweils allen höheren Ebenen entsprechen.

2 Numerische Algorithmen

2.1 Vergleich der Korrektheitskriterien

Kann man die neuen Erkenntnisse der Transaktionsverwaltung, die in Stapelschedulern unter Ausnutzung semantischer Information mehr Parallelität erzielen können, auch zur Parallelisierung von numerischen Algorithmen benutzen? Dazu müssen wir zunächst untersuchen, nach welchen Korrektheitskriterien numerische Algorithmen parallelisiert werden und ob diese mit denjenigen der Datenbanktransaktionsverwaltung vergleichbar sind [Bra89, ZC91, Bau92, KLS$^+$94]. Überraschenderweise ergibt sich eine genaue Übereinstimmung verschieden klingender Begriffe wie in Abbildung 4 (rechts) dargestellt. Eine Variable in paralleler Programmierung heißt in der Datenbanktheorie Objekt, Zuweisungen werden durch Reads und Writes realisiert und die Konflikte, die Reads und Writes in Datenbanken erzeugen können, sind in paralleler Programmierung als Flow-, Anti- und Output-Dependence bekannt. Diese Abhängigkeiten müssen in paralleler

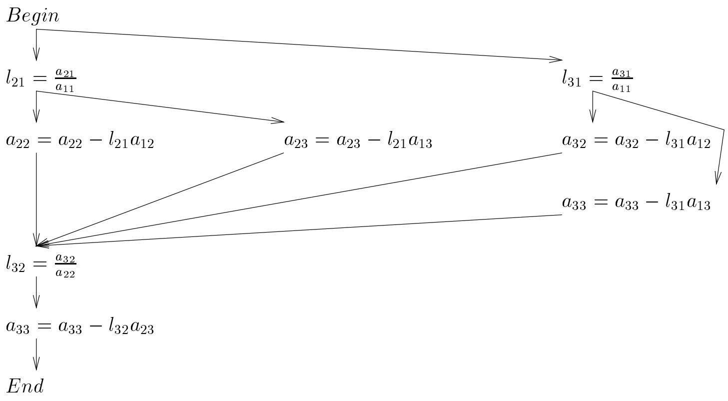


Abbildung 5: Abhängigkeitsgraph des parallelen Programms

Programmierung bewahrt werden, was nichts anderes ist, als dass in der Datenbanktheorie die Reihenfolge von Operationen mit einem wr-, rw- bzw. ww-Konflikt erhalten bleibt.

Schauen wir dazu das Beispiel des ersten Schrittes des Gaußalgorithmus für die Auflösung linearer Gleichungssysteme an. Seien n=3 und die Matrizen A, C_1 und C_2 wie folgt:

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

2.2 Konventionelle Behandlung

Ein paralleles Programm könnte dazu mit herkömmlichen Methoden wie folgt aussehen:

```
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
```

Unter der Annahme, dass mindestens 3 Prozesse/Prozessoren zur Verfügung stehen, sieht der Abhängigkeitsgraph wie in Abbildung 5 aus. Wenn jede Einzelberechnung eine Zeiteinheit benötigt, sind somit 5 Zeiteinheiten zur Abarbeitung des Programms nötig.

2.3 Eingeschränkt parallele Ausführung

In Abbildung 6 ist die Ausführung des obigen Programms in einem Stapelscheduler dargestellt, der auch schwache Ordnungen benutzt. Beispielsweise bedeutet der Pfeil von T_1 nach T_2 eine schwache Ordnung, d.h. der Effekt der Ausführung der beiden Transaktionen muss der gleiche sein wie wenn sie in der spezifizierten Reihenfolge ausgeführt würden, kurz: die Serialisierungsreihenfolge muss T_1–T_2 sein. Die Anwendung schwacher Ordnungen ermöglicht in diesem kleinen Beispiel gegenüber der herkömmlich parallelen Ausführung nur die zusätzliche parallele Ausführung von T_{211} und T_{113}, was jedoch immerhin zu einer Verminderung der benötigten Zeitschritte auf 4 bedeutet, also eine Einsparung von 20%.

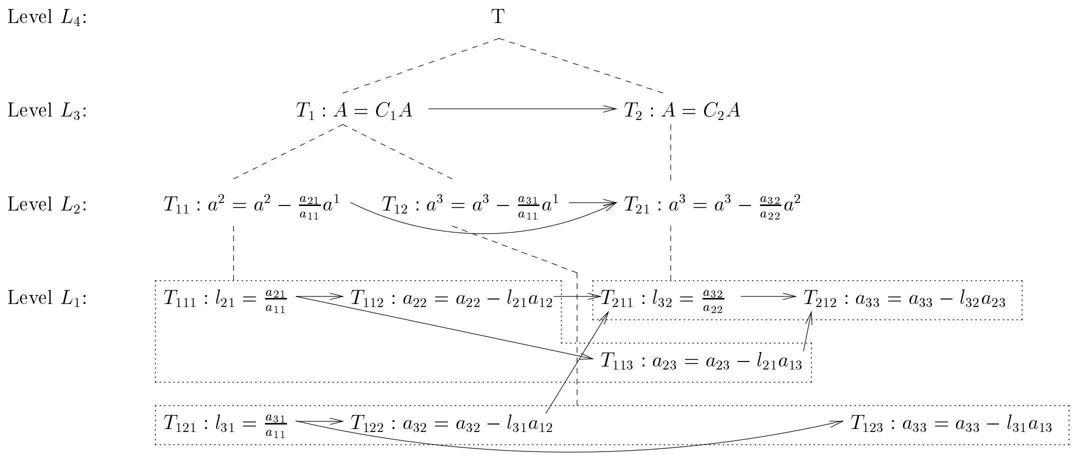


Abbildung 6: Abhängigkeitsgraph des parallelen Programms

3 Diskussion

Wir haben gesehen, dass sich in Stapelschedules die Serialisierungsordnungen der einzelnen Schichten vorgeben lassen, was in allen Fällen, in denen man eine Mehrschichtenarchitektur zur Anwendung bringen kann, höhere Parallelität ermöglicht. Dies ist z.B. bei Datenbankenmehrschichtenschedules und verteilten Transaktionen, aber auch bei numerischen Algorithmen der Fall, von denen wir letzteres Gebiet näher untersucht haben. Voraussetzung dafür war die Feststellung, dass die Korrektheitskriterien der klassichen parallelen Programmierung mit denen der Datenbanktheorie übereinstimmen.

Literatur

[ABFS97] 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.

[Bau92] Barr E. Bauer. *Practical Parallel Programming.* Academic Press, 1992.

[BBG89] Catriel Beeri, Philip A. Bernstein, and Nathan Goodman. A model for concurrency in nested transactions systems. *Journal of the ACM*, 36(2):230–269, April 1989.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, 1987.

[Bra89] Steven Brawer. *Introduction to Parallel Programming.* Academic Press, 1989.

[CGM88] W. Cellary, E. Gelenbe, and T. Morzy. *Concurrency Control in Distributed Database Systems.* North Holland, Amsterdam, 1988.

[KLS+94] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele Jr., and M. E. Zosel. *The High Performance Fortran Handbook.* MIT Press, 1994.

[OV91] M. Tamer Ozsu and Patrick Valduriez. *Principles of Distributed Database Systems.* Prentice-Hall International Editions, 1991.

[VGH93] G. Vossen and M. Groß-Hardt. *Grundlagen der Transaktionsverarbeitung.* Addison-Wesley, 1993.

[Wei91] Gerhard Weikum. Principles and Realisation Strategies of Multilevel Transaction Management. *ACM Transactions on Database Systems*, 16(1):132, March 1991.

[WS92] Gerhard Weikum and Hans-J. Schek. Concepts and Applications of Multilevel Transactions and Open Nested Transactions. In *Database Transaction Models for Advanced Applications*, chapter 13. Morgan Kaufmann, 1992.

[ZC91] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* Addison-Wesley, 1991.