Bounds on Contention Management Algorithms

Johannes Schneider
Roger Wattenhofer

How to handle access to shared data?

- Locks, Monitors…
  - Coarse grained vs. fine grained locking

    easy but slow program — demanding, time consuming but fast programs

lock all data
modify/use data
unlock all data

Only 1 thread can execute

Thread 1

lock Element A
lock Element B
modify/use A,B
lock Element C
modify/use A,B,C
unlock A
modify/use B,C
unlock B,C

Thread 2

lock Element B
lock Element A
modify/use A,B
unlock A,B

Deadlock!

- Problems: difficult, error prone, composability…

Transactional memory(TM) - a possible solution

- Simple for the programmer

```
Begin transaction
modify/use data
End transaction
```

- Composable

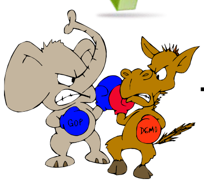
```
Method A.x()
Begin Transaction
B.y()
…
End Transaction
```

```
Method B.y()
Begin transaction
   …
End transaction
```

- Idea from database community
- Many TM systems (internally) still use locks
- But the TM system (not the programmer) takes care of
  - Performance
  - Correctness (no deadlocks...)

Transactional memory systems

- If transactions modify
  - different data, everything is ok

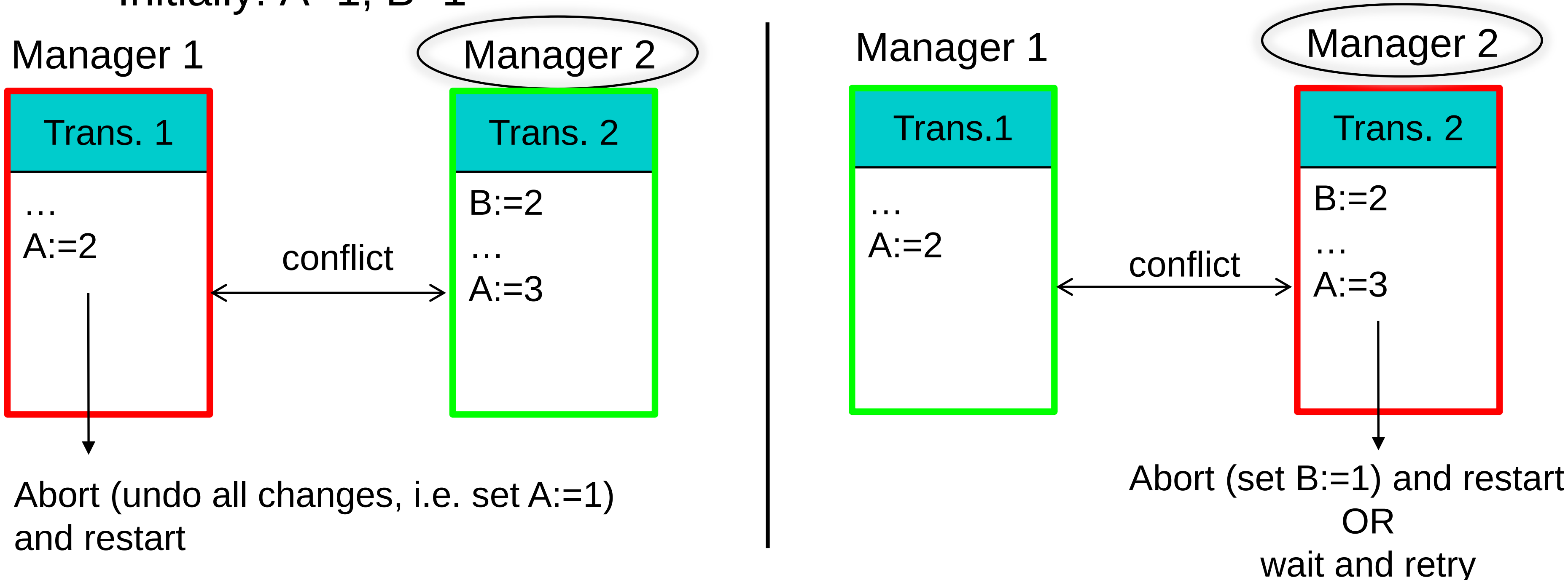


  - the same data, conflicts arise that must be resolved
    - Transactions might get delayed or aborted
    - ⇒ Job of a contention manager

- A transaction keeps track of all modified values
- It restores all values, if it is aborted
- A transaction successfully finishes with a commit

Conflicts – A contention manager decides

- A contention manager can abort or delay a transaction
- Distributed
  - Each thread has its own manager
- Example
  - Initially: A=1, B=1



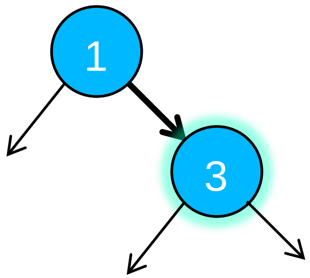
Model

- *n* transactions (and threads) starting concurrently on *n* cores
- Transaction
  - sequence of operations
  - operation takes 1 time unit
  - Duration (number of operations) is fixed
  - 2 types of operations
    - Write = modify (shared) resource and lock it until commit
    - Compute/abort/commit

Contention management is an online problem

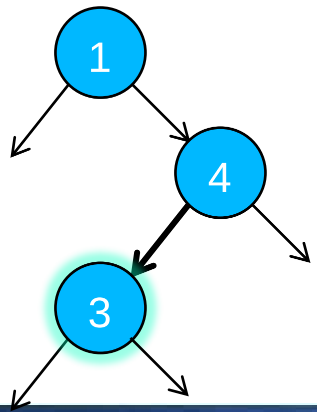
- A transaction demands unknown resources/variables
  - Dynamic data structures change over time
  - Eg.: Binary tree, a transaction wants to insert 3

Modify node 1



⇒If transaction executes a little later, data structure might have changed.

Modify node 4

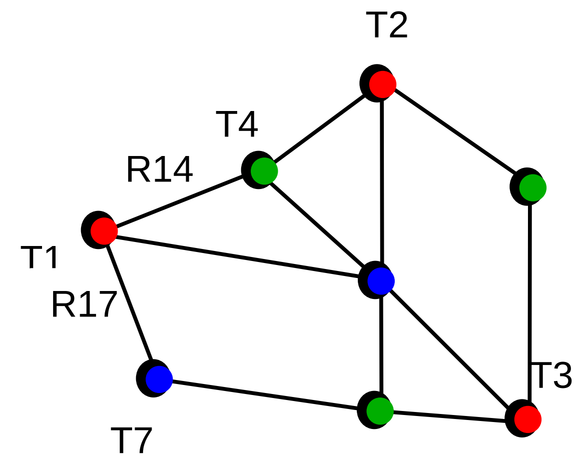


Properties of a contention manager (CM)

- Throughput
  - Makespan = How long it takes until all *n* transactions committed
  - Competitive ratio = $\frac{\text{makespan my CM}}{\text{makespan optimal CM}}$
  - Look at worst case
    - Oblivious adversary = knows my CM (but not random choices)
    - Optimal CM knows decisions of adversary and all conflicts
- Progress guarantees
  - Wait, lock, obstruction freedom

Problem complexity

- Reduction to coloring
  - Coloring problem → CM problem → CM solution → Coloring solution
  - Nodes = transactions
  - Edges = resources (conflicts)
  - Transactions have same duration ***t***
  - Resource acquisition at start, takes no time
  - Transactions of same color don't conflict



Time	[1,t]	[t,2t]	[2t,3t]
Trans. Run&commit	T1,T2,T3	T4,T5,T6	T7,T8

- How hard is it to approximate an optimal vertex coloring?
  - Optimal = Minimum number of colors $\chi(G)$
  - NP-hard to approximate a coloring $\chi(G)^{\frac{\log \chi(G)}{25}}$ [Even et al '08]

Still contention managers are needed…

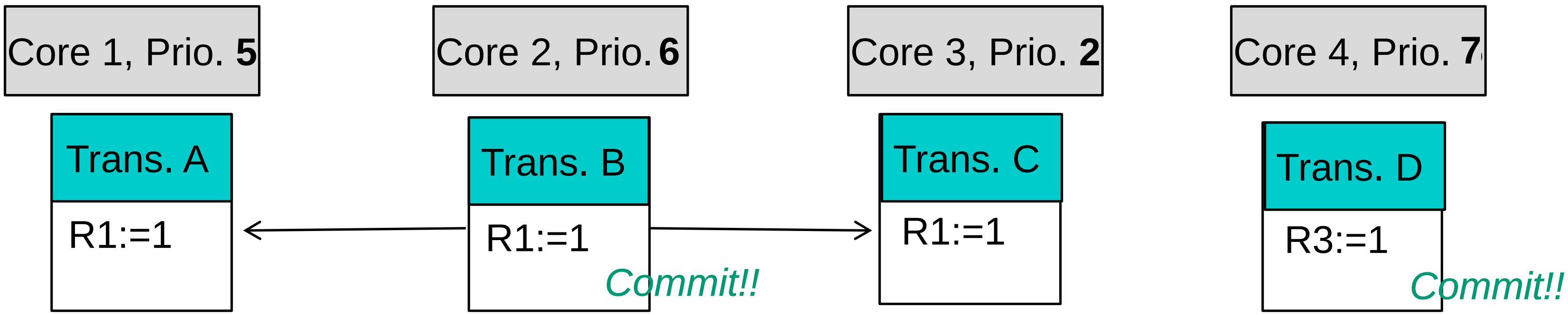
…and there are lots of proposals: [Scherer et al., Ramadan et al., Guerraoui et al.]

- Timestamp
- Kindergarten
- Karma
- Polka
- SizeMatters
- …

- None performs well in the worst case
  - Livelocks or O(n) competitive ratio at best [This paper]
- Some need globally shared data
  - E.g. a global clock, that becomes a bottleneck

Deterministic CM

- Assign priorities to cores
- Transaction TP running on a core P uses P's priority
- Priority of core P changed on commit of TP

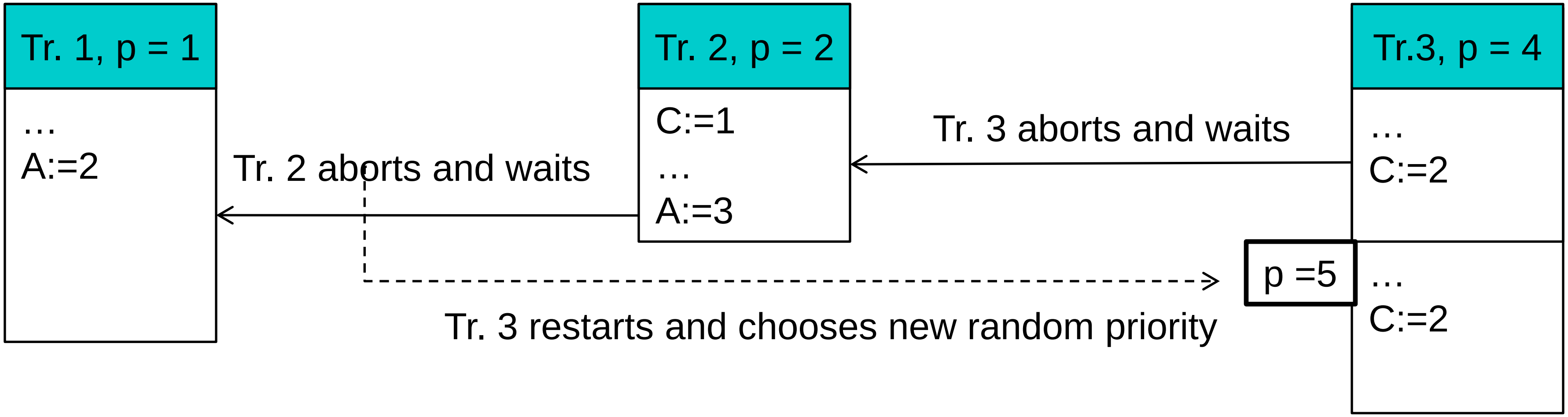


  - Core 2 priority = 1+ max(5,2,1) = 6
  - Core 4 priority = 1+ max(6) = 7

- Worst case: All transactions are executed sequentially
  - But: no global resource

How about a randomized approach?

- Choose a random priority ***p*** from [1,n] on startup
- Transaction A with smaller priority wins against B
  - B aborts and waits until A commits or aborts
  - Then B restarts with new random priority



Rough Analysis

- Assume:
  - Longest transaction takes time ***t***
  - Any transaction conflicts with at most ***d*** -1 other transactions
- After time 2*t* any transaction can restart and draw a new random number
  - Execute for time < *t*, abort, then wait for at most time *t* until restart
- Probability highest random number: 1/*d*
- Choose *e*·*d*·log *n* random numbers
  => Probability to commit is: $1-(1-\frac{1}{e\cdot d})^{e\cdot d\cdot\log n}\approx 1-\frac{1}{e}^{\log n}=1-\frac{1}{n}$
- Time to choose *e*·*d*·*log n* random numbers is *O(t*·*d*·*log n*)

Discussion

- Complexity measure
  - Dependent on number of conflicting transactions *d*
  - Previous: Dependent on number of total resources [Guerraoui et al '05]
    - One can have a lot of parallelism despite many shared resources
- Performance
  - Assume: conflict. transactions *d* = O(1), Duration of transaction *t* = O(1)
  - Makespan of randomized CM: O(log *n*) with 'high' probability
    - Competitive ratio O(log *n*)
  - Deterministic: O(*n*) same as timestamp [Attiya '06]
    - Competitive ratio O(*n*)
    - But: Do not need a global clock (bottleneck)

⇒*Exponential gap* randomized and deterministic algorithm

More results in the paper

- Worst case analysis for other CM algorithms
- Lower bound depending on the power of adversary

Thanks for your attention!