Multi-Core Computing with Transactional Memory
Overview

- Introduction
- Difficulties with parallel (multi-core) programming
- A (partial) solution: Transactional Memory
- Contention Management
Multi-cores will be everywhere

- To increase computing speed, traditionally the clock speed of a CPU was increased
  - Problem: Overheating
- New approach: Have many cores on a single die
- Multi-core chips are used in every PC and soon in every mobile phone
- It is likely that we see a doubling of cores every 2 years like we saw a doubling of clock speed
- BUT: Parallel programming brings new problems and adds complexity for software engineers
Why is parallel programming more difficult?

- We need synchronization...
  - Parallel reservation system for cinema tickets without synchronization

<table>
<thead>
<tr>
<th>Time</th>
<th>Thread 1 - Return 5 tickets</th>
<th>n = Number of sold tickets</th>
<th>Thread 2 - Buy 3 tickets</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Read n (Return 100)</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>100</td>
<td>Read n (Return 100)</td>
</tr>
<tr>
<td>3</td>
<td>New value for n: 100-5 =95</td>
<td>95</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Set n to 95</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>103</td>
<td>New value for n: 100+3=103</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Set n to 103</td>
</tr>
</tbody>
</table>
Two kinds of parallelism

- **Data parallelism**
  - different data for each thread (running on a core)
  - every core works separately
  - No overlapping, no problem!
  - Ex.: Each thread sorts a given set of data unknown to other threads

- **Task parallelism**
  - several tasks working on same/overlapping data
  - Ex.: All threads insert/delete elements in the same tree
Concurrent programming today

- Synchronization using locks or monitors
  - Locks implemented via test-and-set or compare and swap operations
  - Monitor: Mutual exclusion
    - e.g. Java “synchronized method”
    - Easy but slow -> only 1 thread runs at a time

- Coarse grained vs. fine grained locking
  - Easy but slow program
  - Difficult, cumbersome but fast programs
  - Little (no) parallelism
  - Lots of code, deadlocks...

Only 1 thread can operate on the data

Deadlock possible: Thread 1 locks A, while Thread 2 locks B, then both are stuck...
Example: Deleting an element from a linked list

- Sequential code/Coarse grained locking
  - < 10 lines of code
- Concurrent linked list: See below...

```java
public boolean List::delete (KeyType search_key) {
    Node *right_node, *right_node_next, *left_node;
    do {
        right_node = search (search_key, &left_node);
        if ((right_node == null) || (right_node.key != search_key)) /*T1*/
            return false;
        right_node_next = right_node.next;
        if (!is_marked_reference(right_node.next))
            if (CAS(&right_node.next, /*C3*/
                right_node_next, get_marked_reference(right_node.next)))
                break;
    } while (true); /*B4*/
    if (CAS(&left_node.next, right_node, right_node.next)) /*C4*/
        right_node = search (right_node.key, &left_node);
    return true;
}

private Node *List::search (KeyType search_key, Node **left_node) {
    Node *right_node_next, *right_node;
    search_again:
    do {
        Node *t = head;
        Node *t_next = head.next;
        /* 1: Find left_node and right_node */
        do {
            if (!is_marked_reference(t.next)) {
                *left_node = t;
                left_node.next = t.next;
            }
            t = get_unmarked_reference(t.next);
            if (t == tail) break;
            t.next = t.next;
            } while (is_marked_reference(t.next) || (t.key < search_key)); /*B1*/
            /*T1*/
        right_node = t;
        /* 2: Check nodes are adjacent */
        if (left_node_next == right_node)
            if ((right_node != tail) && is_marked_reference(right_node.next))
                goto search_again; /*C1*/
            else
                return right_node; /*R1*/
        /* 3: Remove one or more marked nodes */
        if (CAS(&left_node.next, left_node.next, right_node)) /*C1*/
            if ((right_node != tail) && is_marked_reference(right_node.next))
                goto search_again; /*C2*/
            else
                return right_node; /*R2*/
    } while (true); /*B2*/
```
More problems with locking - Composability

- How to compose objects/components using locks
- If locks are external then programmer must handle locking himself
  - cumbersome (lots of code), error-prone (deadlocks)
- If locks are internal then it is not possible to achieve all desired behaviors
  - Example: Hash table T1 (contains number 1) and T2 (empty)
    No duplicates, each element unique
    2 threads moving elements between tables

```
Algorithm Move(Element e, Table from, Table to)
if from.find(e) then
  to.insert(e)
  from.delete(e)
end if
```
Example continued...

- Threads might be delayed for some reasons: interrupts, cache miss...

<table>
<thead>
<tr>
<th>Time</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Move(1,T1,T2)</strong></td>
<td><strong>Move(1,T2,T1)</strong></td>
</tr>
<tr>
<td>1</td>
<td><strong>T1.find(1)</strong></td>
<td>delayed</td>
</tr>
<tr>
<td>2</td>
<td><strong>T2.insert(1)</strong></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>delayed</td>
<td><strong>T2.find(1)</strong></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td><strong>T1.insert(1)</strong></td>
</tr>
<tr>
<td>5</td>
<td><strong>T1.delete(1)</strong></td>
<td><strong>T2.delete(1)</strong></td>
</tr>
<tr>
<td></td>
<td><strong>both T1 and T2 are empty</strong></td>
<td></td>
</tr>
</tbody>
</table>

- Where is the ‘1’?
Transactional memory (TM) - a (partial) solution

- Simple for the programmer
  
  ```
  Algorithm Move(Element e, Table from, Table to)
  Begin transaction
  if from.find(e) then
    ...
  End Transaction
  ```

- Composable
  
  ```
  Method Table.find(Element e)
  Begin transaction
  ...
  End transaction
  ```

- Many TM systems (internally) still use locks
- But the TM system (not the programmer) cares about
  - Performance
  - Progress/correctness (no deadlocks...)
What is a transaction?

- Nothing new, has been used in databases for a long time
- Characterized by 3 properties (ACI)
  - Atomicity
    - Either a transaction finishes all its operations or no operation has an effect on the system
  - Consistency
    - All objects are in a valid state before and after the transaction
  - Isolation
    - A transaction cannot access or see data in an intermediate (possibly invalid) state of any parallel running transactions.
- For databases also durability
  - If a transaction has completed, its changes are permanent
    - Written on a disk not just in memory
Implementation of a TM system

- Systems exist in hardware, software and as a mix (hybrid)
- (Usually) transactions are executed optimistically
  - i.e. without knowing whether they use the same data
- If transactions work on
  - different data, everything is ok
  - modify the same data, conflicts arise that must be resolved…
  - Transactions might get delayed (has to wait) or aborted.
- A transaction keeps track of all modified values and restores all values, if it is aborted due to a conflict.
- A transaction successfully finishes with a commit
  - Only after the commit, other transactions notice its changes.
Conflicts – A contention manager decides

- A contention manager can abort or delay a transaction
- Important impact on performance
- Example
  - Initially: A=1, B=1

```
Trans. 1
... A:=2
conflict

Trans. 2
B:=2
... A:=3
```

Abort (undo all changes, i.e. set A:=1) and restart

```
Trans. 1
... A:=2
conflict

Trans. 2
B:=2
... A:=3
```

Abort (set B:=1) and restart, retry or wait
Just another example of a contention manager

Trans. 1
A:=1
...
Abort

Trans. 2
...

conflict
A:=2
A:=2

Abort
Why is TM only a partial solution? – Open issues

- I/O support
  - Imagine a document is printed within a transaction and the transaction gets aborted => waste of paper

- Interaction with old, non-transactional (legacy) code

- Efficiency
  - TM still too slow, but catching up quickly…

- Despite the problems:
  - TM system already on the market, partially supporting hardware TM
  - many software TM libraries exist
Open issues from a research perspective

- **Why research?**
  - Help understanding to improve efficiency
  - Create (provable) secure systems

- **System model not sufficient**
  - PRAM: assumes threads are synchronous
  - Only read/write access to memory (e.g. no test and set)
  - No multilevel caching

- **How to resolve conflicts?**
  - What is the ‘best’ contention manager?
Some theory on contention management

- Model: \( n \) transactions (and threads) starting concurrently on \( n \) cores
- \( S \) (shared) resources (variables/objects)
- Transaction = sequence of operations
- Operation:
  - takes 1 time unit
  - 2 kinds: Write, compute/abort/commit
  - Write = modify (shared) resource and lock it until commit
- A conflict arises if transaction A wants to lock a resource that is already locked by B
Model continued...

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

  Initially: Must lock/modify right pointer of node 1

  Assume transaction got aborted and another transaction inserted 4 meanwhile.
  Now: Must lock/modify left pointer of node 4

- Duration (number of operations) is fixed
  - Not true, but mostly only a constant factor away

- Model is a simplification
  - Ex.: There are also reads
  - Ex.: A write access, does not always require a resource to be locked
Contention manager (CM)

- Distributed
  - Each thread has its own manager
- Does not know future(potential) conflicts
  - Conflicts also not learnable, might change
  - Online scheduling problem

Manager 1

Trans. 1

...  
A:=2

Manager 2

Trans. 2

B:=2  
...  
A:=3

Manager 2 resolves conflict

Manager 3

Trans. 3

...  
C:=2
Properties of a contention manager

- **Throughput**
  - Makespan = How long it takes until all n transactions committed = length of a schedule
  - Schedule of transactions defined by decisions of CM
  - Look at worst case
  - Competitive ratio = makespan my CM / makespan optimal CM
    - Oblivious adversary = knows my CM (not random choices)
    - Optimal CM knows decisions of adversary and all conflicts…

- **Progress guarantees**
  - wait freedom (strongest guarantee)
    - all threads(transactions) make progress in a finite number of steps
  - lock freedom
    - one thread makes progress in a finite number of steps
  - obstruction freedom (weakest)
    - a thread makes progress in a finite number of steps in absence of contention (no conflicts, no shared data)
Example of a CM

- Strategy: Be aggressive
  - If a transaction A wants a resource locked by B, then B is aborted

- Throughput?
  - (Possibly) none
    - Livelock: Transactions repeatedly abort each other
    - Eg: 2 Transactions that write/lock the same resource

- Progress guarantees?
  - Obstruction freedom
Problem complexity, it is (NP) hard...

- How long does it take to compute a good schedule?
  - = Is it NP-hard to approximate the optimal makespan by a constant factor?

- …as long as approximating an optimal vertex coloring
  - Optimal = Minimum number of colors = \( \chi(G) \)
  - NP-hard to compute a coloring with
  \[
  \chi(G') \frac{\log \chi(G)}{25}
  \]

- Reduction to coloring
  - Graph -> Scheduling problem -> Schedule -> Coloring
  - Nodes = transactions
  - Edges = resources (conflicts)
  - Transactions have same duration \( t (=1) \)
  - Transactions of same color don’t conflict
    - if resource acquisition takes almost no time, otherwise more complex

- This holds even, if all transactions (potential) conflicts are known and transactions don’t change
It is hard, so what can be done? Another example...

- **CM Strategy: Avoid wasting work**
  - Approximate the work done
  - Each transaction gets a (unique) timestamp $t$ on startup (and after an abort)
  - Conflict: The younger transaction, having performed less work, is aborted

- **Throughput? Progress guarantees?**
  - Oldest transaction will always commit
  - Lock freedom
    - At least one out of $n$ cores successfully executes a transaction
Competitive ratio of the time stamp manager

- S resources
- \( n \) transactions that start concurrently
- Assume each transaction \( T_i \) locks a resource directly after its start for its whole duration \( t_{T_i} \)
- Observe: At most S transactions can run in parallel
  - If \( S+1 \) run in parallel at least 2 must attempt to lock the same resource
- Thus the optimal makespan is at least:
- Makespan CM timestamp is at most:
  - all run sequentially in the worst case
- Competitive ratio = timestamp/ optimal
  \[
  \frac{\sum_{i=0}^{n} \frac{t_{T_i}}{s_i}}{\sum_{i=0}^{n} t_{T_i}} = s = \Omega(s)
  \]
Lower bound on competitive ratio

- Thm: Competitive ratio of any CM (deterministic and randomized) is $\Omega(n)$ if number of resources $S \geq n$
- Proof (only for deterministic CM)

Any CM must abort $\frac{1}{2}$ of all transactions $S_T$, say $S_A$
- Adversary knows the aborted trans. $S_A$
- She/he lets all of them lock the same resource $R_0$
- All aborted transaction ($\frac{1}{2} n$) must run sequentially
- Optimal lets all transactions $S_A$ commit and aborts the other $\frac{1}{2}$
Analysis of algorithm timestamp revisited

- For the lower bound the adversary reduced the parallelism dramatically
  - This is unlikely to happen
- Assume the demanded resources don’t change over time
  - i.e. the adversary cannot reduce parallelism at run-time
- Is the competitive ratio still $\Omega(n)$ (for $S\geq n$)?
  - Yes (proof next slide)
  - All transactions start concurrently
  - Adversary knows timestamps of all transactions
Proof

Tr. 1, t=0
R1 := 1
R2 := 1

Tr. 2, t=1
R2 := 1
R3 := 1

Tr. 3, t=2
R3 := 1
R4 := 1

Tr. 4, t=3
R4 := 1
R3 := 1

Duration t_T = n

...
Proof continued...

- Transaction Ti (>1) aborts at time n-i+1, Trans. 1 commits
- After a restart Transaction Ti (>2) aborts after running for time n-i+2, Trans. 2 commits
- After the next restart Transaction Ti (>3) aborts after running for time n-i+3, Trans. 3 commits
- The time until transaction i=n commits is $\sum_{i=1}^{n} (n-i) = \Omega(n^2)$
- Optimal:
  - Schedules all transaction Ti with even i then the rest
  - $O(n)$
- Competitive ratio: $\Omega(n)$
How about a randomized approach?

- Choose a random priority \( r \) from \([1,n]\) on startup
- Transaction A with larger or same random number wins conflict against B
  - B aborts and waits
  - Restart with a new random number as soon as A either commits or aborts
Analysis

- **Assume:**
  - (needed) resources are not modified
  - Longest transaction takes time $t$
  - Any transaction conflicts with at most $d$ other transactions

- **After time** $2t$ any transaction can restart and draw a new random number
  - Execute for time $t-1$ and then aborts and wait for at most time $t$

- Probability highest rand. number: $1/d$

- Prob. random number unique: $(1 - 1/n)^d < (1 - 1/n)^n \approx 1/e$

- Choose $d \cdot e \cdot \log n$ random numbers
  and probability to commit is:

\[
1 - (1 - \frac{1}{e \cdot d})^{d \cdot \log n} \approx 1 - \frac{1}{e} \cdot \frac{\log n}{n} = 1 - \frac{1}{n}
\]
Analysis continued and evaluation

- Time to choose $d e \log n$ random numbers is $O(t d \log n)$

- How good is the algorithm?
  - For the analysis of algorithm timestamp $d = 2$, $t = n$
    - Makespan of randomized CM: $O(n \log n)$ with ‘high’ probability
    - Deterministic timestamp: $O(n^2)$

- Complexity measure
  - Originally: Dependent on number of resources
  - Now: Dependent on number of conflicts a transaction faces
  - Better?
Theory and practice

- For most benchmarks our randomized approach and a timestamp manager achieve comparable throughput.
- In general, the quality of a CM varies very much across different benchmarks:
  - A CM might be good for one benchmark but bad for another.
- A strategy that is (often) good:
  - After a conflict do some kind of exponential randomized backoff.
  - Reduces load on system, resolves livelocks.
Exponential backoff

- Example: Polka manager
  - Approximate work: priority = number of accessed resources
  - In case of a conflict: If have higher priority abort the other, if have lower priority, then perform an exponential backoff
  - Say priority difference of the two transactions is \( r \)

- Algorithm:
  For \( i = 0..r \)
  - If resource not locked then lock it
  - else wait random time span with mean \( 2^i \)
  After \( r \) unsuccessful trials abort transaction with higher priority
Semester/master theses

- Check the homepage
  - www.dcg.ethz.ch/theses.html

- For TM: Currently, more practical theses
  - Programming, but challenging programming…
  - Focus improve speed
    - Speeding up programs (on multi-core systems)
    - Efficient Multicore Systems with Transactional Memory
That’s it, have a nice vacation!