Chapter 13

Multi-Core Computing

This chapter is based on the article"Distributed Computing and the Multicore Revolution" by Maurice Herlihy and Victor Luchangco. Thanks!

13.1 Introduction

In the near future, nearly all computers, ranging from supercomputers to cell phones, will be multiprocessors. It is harder and harder to increase processor clock speed (the chips overheat), but easier and easier to cram more processor cores onto a chip (thanks to Moore's Law). As a result, uniprocessors are giving way to dual-cores, dual-cores to quad-cores, and so on.

However, there is a problem: Except for "embarrassingly parallel" applications, no one really knows how to exploit lots of cores.

13.1.1 The Current State of Concurrent Programming

In today's programming practice, programmers typically rely on combinations of locks and conditions, such as monitors, to prevent concurrent access by different threads to the same shared data. While this approach allows programmers to treat sections of code as "atomic", and thus simplifies reasoning about interactions, it suffers from a number of severe shortcomings.

- Programmers must decide between *coarse-grained* locking, in which a large data structure is protected by a single lock (usually implemented using operations such as test-and-set or compare and swap(CAS)), and *fine-grained* locking, in which a lock is associated with each component of the data structure. Coarse-grained locking is simple, but permits little or no concurrency, thereby preventing the program from exploiting multiple processing cores. By contrast, fine-grained locking is substantially more complicated because of the need to ensure that threads acquire all necessary locks (and only those, for good performance), and because of the need to avoid deadlocks, when acquiring multiple locks. The decision is further complicated by the fact that the best engineering solution may be

Algorithm Move(Element e, Table from, Table to)

```
if from.find(e) then
    to.insert(e)
    from.delete(e)
end if
```

platform-dependent, varying with different machine sizes, workloads, and so on, making it difficult to write code that is both scalable and portable.

- Conventional locking provides poor support for code composition and reuse. For example, consider a lock-based hash table that provides atomic `insert` and `delete` methods. Ideally, it should be easy to *move* an element atomically from one table to another, but this kind of composition simply does not work. If the table methods synchronize internally, then there is no way to acquire and hold both locks simultaneously. If the tables export their locks, then modularity and safety are compromised. For a concrete example, assume we have two hash tables T_1 and T_2 storing integers and using internal locks only. Every number is only inserted into a table, if it is not already present, i.e., multiple occurrences are not permitted. We want to atomically move elements using two threads between the tables using Algorithm Move. If we have external locks, we must pay attention to avoid deadlocks etc.

	Table T1 is contains 1 and T2 is empty	
Time	Thread 1	Thread 2
	Move(1,T1,T2)	Move(1,T2,T1)
1	T1.find(1)	delayed
2	T2.insert(1)	
3	delayed	T2.find(1)
4		T1.insert(1)
5	T1.delete(1)	T2.delete(1)
	both T1 and T2 are empty	

- Such basic issues as the mapping from locks to data, that is, which locks protect which data, and the order in which locks must be acquired and released, are all based on convention, and violations are notoriously difficult to detect and debug. For these and other reasons, today's software practices make lock-based concurrent programs (too) difficult to develop, debug, understand, and maintain.

The research community has addressed this issue for more than fifteen years by developing nonblocking algorithms for stacks, queues and other data structures. These algorithms are subtle and difficult. For example, the pseudo code of a delete operation for a (non-blocking) linked list, recently presented at a conference, contains more than 30 lines of code, whereas a delete procedure for a (non-concurrent, used only by one thread) linked list can be written with 5 lines of code.

13.2 Transactional Memory

Recently the *transactional memory* programming paradigm has gained momentum as an alternative to locks in concurrent programming. Rather than using locks to give the illusion of atomicity by preventing concurrent access to shared data with transactional memory, programmers designate regions of code as transactions, and the system guarantees that such code appears to execute atomically. A transaction that cannot complete is aborted—its effects are discarded—and may be retried. Transactions have been used to build large, complex and reliable database systems for over thirty years; with transactional memory, researchers hope to translate that success to multiprocessor systems. The underlying system may use locks or nonblocking algorithms to implement transactions, but the complexity is hidden from the application programmer. Proposals exist for implementing transactional memory in hardware, in software, and in schemes that mix hardware and software. This area is growing at a fast pace.

More formally, a transaction is defined as follows:

Definition 13.1. *A transaction in transactional memory is characterized by three 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: Other transactions cannot access or see data in an intermediate (possibly invalid) state of any parallel running transaction.*

Remarks:

- For database transactions there exists a fourth property called durability: If a transaction has completed, its changes are permanent, i.e., even if the system crashes, the changes can be recovered. In principle, it would be feasible to demand the same thing for transactional memory, however this would mean that we had to use slow hard discs instead of fast DRAM chips...
- Although transactional memory is a promising approach for concurrent programming, it is not a panacea, and in any case, transactional programs will need to interact with other (legacy) code, which may use locks or other means to control concurrency.
- One major challenge for the adoption of transactional memory is that it has no universally accepted specification. It is not clear yet how to interact with I/O and system calls should be dealt with. For instance, imagine you print a news article. The printer job is part of a transaction. After printing half the page, the transaction gets aborted. Thus the work (printing) is lost. Clearly, this behavior is not acceptable.
- From a theory perspective we also face a number of open problems. For example:

– System model: An abstract model for a (shared-memory) multiprocessor is needed that properly accounts for performance. In the 80s, the PRAM model became a standard model for parallel computation, and the research community developed many elegant parallel algorithms for this model. Unfortunately, PRAM assume that processors are synchronous, and that memory can be accessed only by read and write operations. Modern computer architectures are asynchronous and they provide additional operations such as test-and-set. Also, PRAM did not model the effects of contention nor the performance implications of multilevel caching, assuming instead a flat memory with uniform-cost access. More realistic models have been proposed to account for the costs of interprocess communication, but these models still assume synchronous processors with only read and write access to memory.

– How to resolve conflicts? Many transactional memory implementations "optimistically" execute transactions in parallel. Conflicts between two transactions intending to modify the same memory at the same time are resolved by a contention manager. A contention manager decides whether a transaction continues, waits or is aborted. The contention management policy of a transactional memory implementation can have a profound effect on its performance, and even its progress guarantees.

13.3 Contention Management

After the previous introduction of transactional memory, we look at different aspects of contention management from a theoretical perspective. We start with a description of the model.

We are given a set of *transactions* $S := \{T_1, ..., T_n\}$ sharing up to *s resources* (such as memory cells) that are executed on n *threads*. Each thread runs on a separate processor/core $P_1, ..., P_n$. For simplicity, each transaction T consists of a sequence of t_T operations. An operation requires one time unit and can be a write access of a resource R or some arbitrary computation.[1] To perform a write, the written resource must be acquired exclusively (i.e., locked) before the access. Additionally, a transaction must store the original value of a written resource. Only one transaction can lock a resource at a time. If a transaction A attempts to acquire a resource, locked by B, then A and B face a conflict. If multiple transactions concurrently attempt to acquire an unlocked resource, an arbitrary transaction A will get the resource and the others face a conflict with A. A *contention manager* decides how to resolve a conflict. Contention managers operate in a distributed fashion, that is to say, a separate instance of a contention manager is available for every thread and they operate independently. Contention managers can make a transaction wait (arbitrarily long) or abort. An aborted transaction undoes all its changes to resources and frees all locks before restarting. Freeing locks and undoing the changes can be done with one operation. A successful transaction finishes with a commit and simply frees

[1]Reads are of course also possible, but are not critical because they do not attempt to modify data.

all locks. A contention manager is unaware of (potential) future conflicts of a transaction. The required resources might also change at any time.

The quality of a contention manager is characterized by different properties:

- Throughput: How long does it take until all transactions have committed? How good is our algorithm compared to an optimal?

  Definition 13.2. *The makespan of the set S of transactions is the time interval from the start of the first transaction until all transactions have committed.*

  Definition 13.3. *The* competitive ratio *is the ratio of the makespans of the algorithm to analyze and an optimal algorithm.*

- Progress guarantees: Is the system deadlock-free? Does every transaction commit in finite time?

  Definition 13.4. *We look at three levels of progress guarantees:*

  – *wait freedom (strongest guarantee): all threads make progress in a finite number of steps*

  – *lock freedom: one thread makes progress in a finite number of steps*

  – *obstruction freedom (weakest): one thread makes progress in a finite number of steps in absence of contention (no other threads compete for the same resources)*

Remarks:

- For the analysis we assume an *oblivious* adversary. It knows the algorithm to analyze and chooses/modifies the operations of transactions arbitrarily. However, the adversary does not know the random choices (of a randomized algorithm). The optimal algorithm knows all decisions of the adversary, i.e., first the adversary must say how all transactions look like and then the optimal algorithm, having full knowledge of all transactions *and* all malicious acts of the adversary, computes an (optimal) schedule.

- Wait freedom implies lock freedom. Lock freedom implies obstruction freedom.

- Here is an example to illustrate how needed resources change over time: Consider a dynamic data structure such as a balanced tree. If a transaction attempts to insert an element, it must modify a (parent) node and maybe it also has to do some rotations to rebalance the tree. Depending on the elements of the tree, which change over time, it might modify different objects. For a concrete example, assume that the root node of a binary tree has value 4 and the root has a (left) child of value 2. If a transaction A inserts value 5, it must modify the pointer to the right child of the root node with value 4. Thus it locks the root node. If A gets aborted by a transaction B, which deletes the node with value 4 and commits, it will attempt to lock the new root node with value 2 after its restart.

- There are also systems, where resources are not locked exclusively. All we need is a correct serialization (analogous to transactions in database systems). Thus a transaction might speculatively use the current value of a resource, modified by an uncommitted transaction. However, these systems must track dependencies to ensure the ACI properties of a transaction (see Definition 13.1). For instance, assume a transaction T_1 increments variable x from 1 to 2. Then transaction T_2 might access x and assume its correct value is 2. If T_1 commits everything is fine and the ACI properties are ensured, but if T_1 aborts, T_2 must abort too, since otherwise the atomicity property was violated.

- In practice, the number of concurrent transactions might be much larger than the number of processors. However, performance may decrease with an increasing number of threads since there is time wasted to switch between threads. Thus, in practice, load adaption schemes have been suggested to limit the number of concurrent transactions close to (or even below) the number of cores.

- In the analysis, we will assume that the number of operations is fixed for each transaction. However, the execution time of a transaction (in the absence of contention) might also change, e.g., if data structures shrink, less elements have to be considered. Nevertheless, often the changes are not substantial, i.e., only involve a constant factor. Furthermore, if an adversary can modify the duration of a transaction arbitrarily during the execution of a transaction, then any algorithm must make the exact same choices as an optimal algorithm: Assume two transactions T_0 and T_1 face a conflict and an algorithm *Alg* decides to let T_0 wait (or abort). The adversary could make the opposite decision and let T_0 proceed such that it commits at time t_0. Then it sets the execution time T_0 to infinity, i.e., $t_{T_0} = \infty$ after t_0. Thus, the makespan of the schedule for algorithm *Alg* is unbounded though there exists a schedule with bounded makespan. Hence the competitive ratio is unbounded.

Problem complexity

In graph theory, coloring a graph with as few colors as possible is known to be hard problem. A (vertex) coloring assigns a color to each vertex of a graph such that no two adjacent vertices share the same color. It was shown that computing an optimal coloring given complete knowledge of the graph is NP-hard. Even worse, computing an approximation within a factor of $\chi(G)^{\log \chi(G)/25}$, where $\chi(G)$ is the minimal number of colors needed to color the graph, is NP-hard as well.

To keep things simple, we assume for the following theorem that resource acquisition takes no time, i.e., as long as there are no conflicts, transactions get all locks they wish for at once. In this case, there is an immediate connection to graph coloring, showing that even an *offline* version of contention management, where all potential conflicts are known and do not change over time, is extremely hard to solve.

Theorem 13.5. *If the optimal schedule has makespan k and resource acquisition takes zero time, it is NP-hard to compute a schedule of makespan less than*

$k^{\log k/25}$, *even if all conflicts are known and transactions do not change their resource requirements.*

Proof. We will prove the claim by showing that any algorithm finding a schedule taking $k' < k^{(\log k)/25}$ can be utilized to approximate the chromatic number of any graph better than $\chi(G)^{\frac{\log \chi(G)}{25}}$.

Given the graph $G = (V, E)$, define that V is the set of transactions and E is the set of resources. Each transaction (node) $v \in V$ needs to acquire a lock on all its resources (edges) $\{v, w\} \in E$, and then computes something for exactly one round. Obviously, this "translation" of a graph into our scheduling problem does not require any computation at all.

Now, if we knew a $\chi(G)$-coloring of G, we could simply use the fact that the nodes sharing one color form an independent set and execute all transactions of a single color in parallel and the colors sequentially. Since no two neighbors are in an independent set and resources are edges, all conflicts are resolved. Consequently, the makespan k is at most $\chi(G)$.

On the other hand, the makespan k must be at least $\chi(G)$: Since each transaction (i.e., node) locks all required resources (i.e., adjacent edges) for at least one round, no schedule could do better than serve a (maximum) independent set in parallel while all other transactions wait. However, by definition of the chromatic number $\chi(G)$, V cannot be split into less than $\chi(G)$ independent sets, meaning that $k \geq \chi(G)$. Therefore $k = \chi(G)$.

In other words, if we could compute a schedule using $k' < k^{(\log k)/25}$ rounds in polynomial time, we knew that

$$\chi(G) = k \leq k' < k^{(\log k)/25} = \chi(G)^{(\log \chi(G))/25}.$$

□

Remarks:

- The theorem holds for a central contention manager, knowing all transactions and all potential conflicts. Clearly, the *online* problem, where conflicts remain unknown until they occur, is even harder. Furthermore, the distributed nature of contention managers also makes the problem even more difficult.

- If resource acquisition does not take zero time, the connection between the problems is not a direct equivalence. However, the same proof technique shows that it is NP-hard to compute a polynomial approximation, i.e., $k' \leq k^c$ for some constant $c \geq 1$.

Deterministic contention managers

Theorem 13.5 showed that even if all conflicts are known, one cannot produce schedules which makespan get close to the optimal without a lot of computation. However, we target to construct contention managers that make their decisions quickly without knowing conflicts in advance. Let us look at a couple of contention managers and investigate their throughput and progress guarantees.

- A first naive contention manger: Be aggressive! Always abort the transaction having locked the resource. Analysis: The throughput might be zero, since a livelock is possible. But the system is still obstruction free. Consider two transactions consisting of three operations. The first operation of both is a write to the same resource R. If they start concurrently, they will abort each other infinitely often.

- A smarter contention manager: Approximate the work done. Assume before a start (also before a restart after an abort) a transaction gets a unique timestamp. The older transaction, which is believed to have already performed more work, should win the conflict.

  Analysis: Clearly, the oldest transaction will always run until commit without interruption. Thus we have lock-freedom, since at least one transaction makes progress at any time. In other words, at least one processor is always busy executing a transaction until its commit. Thus, the bound says that all transactions are executed sequentially. How about the competitive ratio? We have s resources and n transactions starting at the same time. For simplicity, assume every transaction T_i needs to lock at least one resource for a constant fraction c of its execution time t_{T_i}. Thus, at most s transactions can run concurrently from start until commit without (possibly) facing a conflict (if $s+1$ transactions run at the same time, at least two of them lock the same resource). Thus, the makespan of an optimal contention manager is at least: $\sum_{i=0}^{n} \frac{c \cdot t_{T_i}}{s}$. The makespan of our timestamping algorithm is at most the duration of a sequential execution, i.e. the sum of the lengths of all transactions: $\sum_{i=0}^{n} t_{T_i}$. The competitive ratio is:

  $$\frac{\sum_{i=0}^{n} t_{T_i}}{\sum_{i=0}^{n} \frac{c \cdot t_{T_i}}{s}} = \frac{s}{c} = O(s).$$

  Remarks:

  - Unfortunately, in most relevant cases the number of resources is larger than the number of cores, i.e., $s > n$. Thus, our timestamping algorithm only guarantees sequential execution, whereas the optimal might execute all transactions in parallel.

Are there contention managers that guarantee more than sequential execution, if a lot of parallelism is possible? If we have a powerful adversary, that can change the required resources after an abort, the analysis is tight. Though we restrict to deterministic algorithms here, the theorem also holds for randomized contention managers.

Theorem 13.6. *Suppose n transactions start at the same time and the adversary is allowed to alter the resource requirement of any transaction (only) after an abort, then the competitive ratio of any deterministic contention manager is $\Omega(n)$.*

Proof. Assume we have n resources. Suppose all transactions consist of two operations, such that conflicts arise, which force the contention manager to abort one of the two transactions T_{2i-1}, T_{2i} for every $i < n/2$. More precisely, transaction T_{2i-1} writes to resource R_{2i-1} and to R_{2i} afterwards. Transaction T_{2i} writes to resource R_{2i} and to R_{2i-1} afterwards. Clearly, any contention manager has to abort $n/2$ transactions. Now the adversary tells each transaction which did not finish to adjust its resource requirements and write to resource R_0 as their first operation. Thus, for any deterministic contention manager the $n/2$ aborted transactions must execute sequentially and the makespan of the algorithm becomes $\Omega(n)$.

The optimal strategy first schedules all transactions that were aborted and in turn aborts the others. Since the now aborted transactions do not change their resource requirements, they can be scheduled in parallel. Hence the optimal makespan is 4, yielding a competitive ratio of $\Omega(n)$. □

Remarks:

- The prove can be generalized to show that the ratio is $\Omega(s)$ if s resources are present, matching the previous upper bound.
- But what if the adversary is not so powerful, i.e., a transaction has a fixed set of needed resources?

  The analysis of algorithm timestamp is still tight. Consider the dining philosophers problem: Suppose all transactions have length n and transaction i requires its first resource R_i at time 1 and its second R_{i+1} (except T_n, which only needs R_n) at time $n - i$. Thus, each transaction T_i potentially conflicts with transaction T_{i-1} and transaction T_{i+1}. Let transaction i have the i^{th} oldest timestamp. At time $n - i$ transaction $i + 1$ with $i \geq 1$ will get aborted by transaction i and only transaction 1 will commit at time n. After every abort transaction i restarts 1 time unit before transaction $i - 1$. Since transaction $i - 1$ acquires its second resource $i - 1$ time units before its termination, transaction $i - 1$ will abort transaction i at least $i - 1$ times. After $i - 1$ aborts transaction i may commit. The total time until the algorithm is done is bounded by the time transaction n stays in the system, i.e., at least $\sum_{i=1}^{n}(n - i) = \Omega(n^2)$. An optimal schedule requires only $O(n)$ time: First schedule all transactions with even indices, then the ones with odd indices.
- Let us try to approximate the work done differently. The transaction, which has performed more work should win the conflict. A transaction counts the number of accessed resources, starting from 0 after every restart. The transaction which has acquired more resources, wins the conflict. In case both have accessed the same number of resources, the transaction having locked the resource may proceed and the other has to wait.

  Analysis: Deadlock possible: Transaction A and B start concurrently. Transaction A writes to R_1 as its first operation and to R_2 as its second operation. Transaction B writes to the resources in opposite order.

Randomized contention managers

Though the lower bound of the previous section (Theorem 13.6) is valid for both deterministic and randomized schemes, let us look at a randomized approach:

Each transaction chooses a random priority in $[1, n]$. In case of a conflict, the transaction with lower priority gets aborted. (If both conflicting transactions have the same priority, both abort.)

Additionally, if a transaction A was aborted by transaction B, it waits until transaction B committed or aborted, then transaction A restarts and draws a new priority.

Analysis: Assume the adversary cannot change the resource requirements, otherwise we cannot show more than a competitive ratio of n, see Theorem 13.6. Assume if two transactions A and B (potentially) conflict (i.e., write to the same resource), then they require the resource for at least a fraction c of their running time. We assume a transaction T potentially conflicts with d_T other transactions. Therefore, if a transaction has highest priority among these d_T transactions, it will abort all others and commit successfully. The chance that for a transaction T a conflicting transaction chooses the same random number is $(1 - 1/n)^{d_T} > (1 - 1/n)^n \approx 1/e$. The chance that a transaction chooses the largest random number and no other transaction chose this number is thus at least $1/d_T \cdot 1/e$. Thus, for any constant $c \geq 1$, after choosing $e \cdot d_T \cdot c \cdot \ln n$ random numbers the chance that transaction T has commited successfully is

$$1 - \left(1 - \frac{1}{e \cdot d_T}\right)^{e \cdot d_T \cdot c \cdot \ln n} \approx 1 - e^{-c \ln n} = 1 - \frac{1}{n^c}.$$

Assuming that the longest transaction takes time t_{max}, within that time a transaction either commits or aborts and chooses a new random number. The time to choose $e \cdot t_{max} \cdot c \cdot \ln n$ numbers is thus at most $e \cdot t_{max} \cdot d_T \cdot c \cdot \ln n = O(t_{max} \cdot d_T \cdot \ln n)$. Therefore, with high probability each transaction makes progress within a finite amount of time, i.e., our algorithm ensures wait freedom. Furthermore, the competitive ratio of our randomized contention manger for the previously considered dining philosophers problem is w.h.p. only $O(\ln n)$, since any transaction only conflicts with two other transactions.