Contents of Lecture 11

- More Transactional Memory
- Clojure
- GPUs and OpenCL
- Accelerators
- Spinlocks and busy wait
- Data prefetching with gcc, clang and xlc
- Special memory instructions on POWER
- Summary of the course and predictions
Many more L1 cache misses in TM code

Instruction path lengths = number of executed instructions
Single thread slowdown

- L1 cache misses penalize BG/Q Short
- OpenMP sometimes better due to better register allocation, i.e. luck
Speedups: sometimes good but none very efficient

Figure 5: Speedup over sequential for upto 64 threads.
The POWER architecture has supported TM since version 2.07, published May 10, 2013.

POWER8 implements version 2.07.

Every memory access is either transactional or non-transactional.

New instructions include (the . suffix means they set the condition codes in CR0):
- tbegin.
- tend.
- tabort.
- tsuspend. leave the transaction
- tresume. return to the transaction
- treclaim. used by kernel at context switches
- trechkpt. used by kernel to copy certain registers

Memory accesses executed between the tbegin. and tend. are transactional and all other are non-transactional.
The bit TDOOMED is set to 0 by tbegin. and to 1 at a failure.
Most registers (but not CR0 obviously) are saved at a tbegin. and are restored at a transaction failure.
A failure handler is run at a transaction failure
A transaction can fail either due to itself or due to another transaction
It fails by itself (called self-induced) if:
  - It executes tabort.
  - It has a too deep transaction nesting level at a new tbegin.
  - It has a too large footprint (written too much data)
  - It executes a disallowed instruction — such as doze or sleep
The failure handler should either retry the transaction or do the operation without a transaction (i.e. with locks)
There is no guarantee of any progress or fairness by the hardware
Transactions can be nested.

A `tend. with field A=1` ends all transactions of the thread and with `A=0` only ends the most recently started.

A failure of a nested transaction terminates all transactions!
A transaction conflicts with another transaction or a non-transactional access if they access the same cache block (i.e. memory block) and at least one is a store.

At least one of two conflicting transactions fail, i.e. are aborted.

Note the cache block granularity: since the cache block size is not defined by the architecture, software must be written accordingly.

The reason for transaction failure is provided in a register.
There are three states:

- Non-transactional: the normal state before any transaction is started
- Transactional: execution between a `tbegin` and a `tend`.
- Suspended: execution by the same thread but as a temporary escape of the transactional state. This is execution between a `tsuspend` and a `tresume`.

The purpose of the suspended state is for instance

- Inter-thread communication: cannot be rolled back!
- Other stores which should not be rolled back such as for debugging.
- Accesses to non-cachable memory.
Suspend confusions

x = 0;
tbegin
x = 1;
tsuspend
x += 1
tresume
tend

- The transaction fails
- What will the value of x be?
- IBM manual about using the processor suspended state:
  - Accessing storage locations in Suspended state that have been accessed transactionally has the potential to create apparent storage paradoxes.
  - It must be used with care.
POWER8 the POWER 2.07 architecture
As in BG/Q the L2 cache is used for speculative state
The L1 data cache is 64 KB and the L1 instruction cache is 32 KB
The cache block size is 128 bytes in all caches
All caches are 8-way associative
All up to 8 hardware threads in a core share the same L1 and L2 caches
Each core has its own L2 512 KB cache
The L3 cache is 8 MB and there is one per chip, i.e. one in power.cs.lth.se
Other HTM implementations

- Intel’s Transactional Synchronization Extensions (TSX) were used in some Haswell processors in 2013
- In 2014 a bug was detected and TSX was disabled with a microcode update on these chips
- TSX is a software API and the hardware details are not public
- Some Skylake processors support it
- Both AMD and ARM also have HTM
- IBM mainframes (around since 1952!) now also implement HTM, such as zEC12
- zEC12 has 6 cores clocked at 5.5 GHz
In 2015 IBM Research in Tokyo and Austin compared these on STAMP
Some "TM unfriendly" parts of STAMP were fixed
The number of used retries were tuned to each machine
All machines detect conflicts using the cache coherence protocol
On BG/Q programmers cannot write code to handle failed transactions
On BG/Q system calls are used to begin and commit transactions
The other machines use normal machine instructions
zEC12 has the largest cache lines, 256 bytes, and highest risk for false conflicts
zEC12 uses also the L1 cache to detect conflicts (i.e. in addition to the L2 cache)
Intel uses also the L1 cache to detect conflicts in addition to other resources
When the transaction capacity is exceeded, the transaction is aborted
The transaction capacity of Intel is not public but experiments revealed it to be 4 MB for reads and 22 KB for writes
POWER8 has a combined transaction capacity of 8 KB only
## Table 1. HTM implementations of Blue Gene/Q, zEC12, Intel Core i7-4770, and POWER8

<table>
<thead>
<tr>
<th>Processor type</th>
<th>Blue Gene/Q</th>
<th>zEC12</th>
<th>Intel Core i7-4770</th>
<th>POWER8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conflict-detection granularity</td>
<td>8 - 128 bytes</td>
<td>256 bytes</td>
<td>64 bytes</td>
<td>128 bytes</td>
</tr>
<tr>
<td>Transactional-load capacity</td>
<td>20 MB (1.25 MB per core)</td>
<td>1 MB</td>
<td>4 MB</td>
<td>8 KB</td>
</tr>
<tr>
<td>Transactional-store capacity</td>
<td>20 MB (1.25 MB per core)</td>
<td>8 KB</td>
<td>22 KB</td>
<td>8 KB</td>
</tr>
<tr>
<td>L1 data cache</td>
<td>16 KB, 8-way</td>
<td>96 KB, 6-way</td>
<td>32 KB, 8-way</td>
<td>64 KB</td>
</tr>
<tr>
<td>L2 data cache</td>
<td>32 MB, 16-way, (shared by 16 cores)</td>
<td>1 MB, 8-way</td>
<td>256 KB</td>
<td>512 KB, 8-way</td>
</tr>
<tr>
<td>SMT level</td>
<td>4</td>
<td>None</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>Kinds of abort reasons</td>
<td>-</td>
<td>14</td>
<td>6</td>
<td>11</td>
</tr>
</tbody>
</table>
### Evaluation

<table>
<thead>
<tr>
<th>CPU</th>
<th>cores</th>
<th>SMT</th>
<th>clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Core 7-4770</td>
<td>4</td>
<td>2</td>
<td>3.4 GHz</td>
</tr>
<tr>
<td>IBM zEC12</td>
<td>16</td>
<td>1</td>
<td>5.5 GHz</td>
</tr>
<tr>
<td>IBM BG/Q</td>
<td>16</td>
<td>4</td>
<td>1.6 GHz</td>
</tr>
<tr>
<td>IBM POWER8</td>
<td>6</td>
<td>8</td>
<td>4.1 GHz</td>
</tr>
</tbody>
</table>

- The POWER8 is available with 6 to 12 cores
- To make comparisons fair, only four cores were used in any machine
- The performance metric is speedup over single thread for the same machine
- Thus no comparison of execution times of IBM vs Intel since the machines are so different
- The purpose is to understand what to improve in future TM implementations
4 core speedups vs itself

![Chart showing speedup comparison between different processors for various applications.](chart.png)
Intel’s hardware prefetching is counted as transaction accesses
This causes a higher number of data conflicts than the IBM machines
This is significant for kmeans-low
The IBM researchers disabled hardware prefetch on Intel to verify this
Then informed Intel which confirmed the findings
POWER8 had more capacity failures so the capacity should be increased
From the paper: The zEC12 suffers from mysterious transaction aborts that degrade its performance
The failure codes where not documented and happened in odd situations
This is a better explanation than saying ”did not do it due to lack of time”!
Transaction failures

![Bar chart showing transaction abort ratio for different scenarios and applications.](image-url)
Clojure

- Clojure is a Lisp language with support for software transactional memory
- To use STM you need to use a special type, called ref
- This has the advantage that only such objects need to be logged

```
(def start-balance 1000) ; #define
(defrecord account [balance]) ; ; class
(def pointer (ref (->account start-balance))) ; ''new''

(println (deref pointer)) ; *pointer
(println @pointer) ; *pointer

(update @pointer :balance + 5) ; uses ''new''

(println @pointer)
```

- The update creates a new object with balance 1005
- It does not modify pointer or the first account
ref-set

- ref-set is used to modify a ref

```clojure
(def start-balance 1000)
(defrecord account [balance])
(def pointer (ref (->account start-balance)))

(println @pointer)

(ref-set pointer (update @pointer :balance + 5))

(println @pointer)
```

- Good start but not sufficient
- We can only modify a ref in a transaction
Use dosync

(def start-balance 1000)
(defrecord account [balance])
(def pointer (ref (->account start-balance)))

(println @pointer)

(dosync (ref-set pointer (update @pointer :balance + 5))
  (ref-set pointer (update @pointer :balance + 6))
  (ref-set pointer (update @pointer :balance + 7)))

(println @pointer)

The balance becomes 1018
The term GPU was coined by Nvidia in 1999 when their GeForce 256 was launched. It was a one chip processor for transformations, lighting, and triangle setup and clipping. Performance was 10 million polygons per second. Their competitor ATI launched Radeon 9700 in 2002. These early GPUs were not programmable but had APIs used by the host computer. Modern GPUs are programmable e.g. in OpenCL — based on C.
Differences between CPUs and GPUs

- CPUs have higher clock frequency compared with GPUs
- GPUs have many many more cores (e.g. hundreds or thousands)
- GPUs are good at SIMD processing
- GPUs cannot run OSes — no interrupt mechanism
- CPUs are better at control (such as if-then-else)
- CPU instruction sets are published while those of GPUs are not
- GPU instructions can change between chip generations
- Intel makes integrated GPUs — on the same chip as the CPUs
- In Intel Core M, the GPU use about 60 % of the chip area
- CPUs and GPUs can complement each other
OpenCL originates from Apple and is an Apple trademark
Can e.g. be used with a C/C++ program on a host computer
The purpose of OpenCL is to
  - Let software exploit all parallel hardware on a machine: CPUs, GPUs, DSPs etc
  - Provide a language that vendors agree on
  - Virtual memory works in compute devices such as GPUs — with OpenCL 2.0
  - Make software more portable

How can this be achieved?
Portability despite different ISAs

- Due to different instruction set architectures, OpenCL programs are compiled at runtime!
- This costs some execution time (obviously) but achieves portability
- So each vendor provides an OpenCL-compiler and a runtime library.
- OpenCL is developed through the Khronos Group industry consortia.
Based on C99

- No pointers to functions
- No bit-fields
- No variable-length arrays
- No structs
- New keyword `__kernel`
- A kernel is a function which should be executed by some compute engine
OpenCL Hardware Model

- Host — CPU
- Compute device — e.g. a GPU
- A compute unit — part of a compute device
- A processing element — part of a compute unit
Host code + device code
Host tells device to transfer data to/from memory
Host tells device to execute code
An application consists of serial (only host) and parallel parts (run at devices)
Work items

- An N-dimensional domain is defined
- A **kernel** is executed on each point in the domain
- A kernel is a short function

```c
__kernel void mul(
    __global const float* a,
    __global const float* b,
    __global float* c)
{
    int id;

    id = get_global_id(0);
    c[id] = a[id] * b[id];
}
```
Execution Model

- A work-item is executed by a kernel function
- A *context* is the environment where a work-item is executed
  - device
  - device memory
  - command queue
- A *command queue* is used by the host to submit work to a device
Memory

- Private memory — per work item
- Local memory — shared within a work group
- Global/Constant memory — visible to all work groups
- Host memory — system RAM
- Programmer is responsible for transferring data
OpenCL language features

- Scalar types from C
  - `image2d_t`, `image3d_t`, `sampler_t`
- Various vector types
- Memory fence — as in C11
- Memory barrier — all wait here
Recent research: accelerators

- GPUs can be seen as accelerators for certain computations
- There are many computations for which neither CPUs nor GPUs provide sufficient performance for the cost
- Examples which "always" need better performance:
  - Medicine: individual radiation therapy simulations: it is desired to go from weeks to hours
  - Medicine: image analysis, public health, pharmaceuticals
  - Finance: trading, order books, fraud detection in milliseconds
  - Manufacturing: CAD and 3D modeling
  - Data analytics: pattern recognition
- One option is to buy a really fast supercomputer
- Another is to build specialized hardware: an accelerator which is good at basically one algorithm
Using accelerators

- Algorithms, or parts of algorithms, implemented directly in hardware can be much more efficient.
- For some applications the performance demands can make ASICs or FPGAs less expensive than buying a huge supercomputer.
- An accelerator which is not using the cache coherence protocol of the rest of the machine needs OS kernel assistance to copy data.
- Writing programs which use such accelerators typically is a bit messy.
- Communication between the main program and the accelerator can be time consuming so that the resulting performance might not be sufficient.
A new industry platform to make accelerators easier to use

Companies: e.g. AMD, Google, HP, IBM, Mellanox, Nvidia, Samsung, SK Hynix, Toshiba, Western Digital, Xilinx, and others

Universities: e.g. ETZ in Zürich and the University of Tokyo

The idea is to make external hardware accelerators cache coherent so they can live in the same address space as normal threads

This hardware can look like a thread to the rest of the program: it can take your mutex and change memory

A trivial example: we could build an FPGA which performs

\[ in(w) \leftarrow use(w) \cup (out(w) - def(w)) \]

in one very parallel operation
In the Cell architecture, the main program copies data to a SIMD compute engine (called SPU)
- After a while a result is copied back
- The SPU lives in a different address space and cannot e.g. follow pointers in a linked list
- Since a CAPI device is part of the same address space it can do that
- It has its own cache (if it needs one)
The CAPP has a directory of all cache lines held by the CAPI device.
In this case POWER8 but other CPU manufacturers may do similar.
Test-And-Set Machine Instruction

Definition expressed in the C language

```c
int test_and_set(int* ptr)
{
    int old_value;

    old_value = *ptr;
    *ptr = 1;
    return old_value;
}
```

- This machine instruction is executed *atomically* even on a multiprocessor
- Example assembler syntax: `ts ra, rb, rd`. Old value is written to rd.
The assumption is we use the machine for one important application
Used on multiprocessors when a lock is needed for a short time.
Uses busy-wait while waiting for the lock — a context-switch takes more time than the expected waiting time.

```c
void lock(int *ptr)
{
    while (test_and_set(ptr)) ; /* busy-wait */
}

void unlock(int* ptr)
{
    *ptr = 0;
}
```
Disadvantages of our Spinlock

- It generates a lot of bus-traffic as we will see soon.
- It is not fair. A processor with poor luck will not get the lock.
- But for a small-scale multiprocessor, say with two processors, it is excellent.
Performance of our Spin-Lock Implementation for Two Processors

- The processor which happen to execute its test-and-set instruction first will get the lock, store a one in memory and execute its critical section.
- The other processor will sit in the while loop and read a one, store back a one, and continue in the while loop.
- The first time the second processor executes the test-and-set instruction, it will fetch and then invalidate the copy stored in the first processor’s cache.
- The second and other iterations will only result in reads and writes to the private cache. No bus-traffic is generated.
- When the lock is released the first processor will write a zero, the second will get a cache miss and eventually get the lock.
Suppose now instead there are multiple processors which need the lock.

Suppose \( P_0 \) has the lock, and that \( P_1 \) and \( P_2 \) also need it.

Both \( P_1 \) and \( P_2 \) will invalidate the others’ copy every loop iteration.

Both will suffer cache misses, although that in itself does not matter much here (since the processor has nothing else to do anyway) but the problem is bus-traffic.

The cache-block where the lock is will ping-pong between the caches of \( P_2 \) and \( P_3 \) and thereby occupying the bus with the effect of slowing down other processor’s memory accesses. How can we solve this problem?
A Better Spinlock Implementation

- We would like the waiting processor to only affect their own private cache until the owner of the lock releases the lock.
- This can be achieved by adding another loop:

```c
void lock(int *ptr)
{
    while (test_and_set(ptr))
        while (*ptr) /* loop on copy in private cache. */
            ;
}
```

- Now no bus-traffic is generated by the waiting processors, however, every waiting processor will generate a write just after the unlock.
With an atomic **fetch-and-increment** machine instruction we can get rid of the write-traffic which occurs when a spinlock is released.

The idea is simply to model a normal queue.

Two atomic variables are used: `atomic_uint ticket_number, now_serving`.

When starting to wait, a processor fetches and increments `ticket_number` and waits until `now_serving == ticket_number`.

When releasing the lock, `now_serving` is incremented.

All processors are spinning on the private copy of `now_serving` which all refetch when it is incremented.
For large multiprocessors, it is undesirable to generate such massive bus-contention due to reads and writes as in spinlocks and ticket-based locks.

At the expense of memory, an array of locks, one for each processor, can be used:

```c
typedef struct {
    atomic_uint* array[P]  /* Pointers to spinlocks. */
    atomic_uint    next;  /* modulo P. */
} lock_t;
```

When starting to wait, the next field is fetched and incremented modulo the number of processors. Then if the spinlock pointed to is locked, the processor waits until that lock will be released. Only one processor is notified about a release.
A cache block is the amount of data which is fetched from memory at a read miss, and is invalidated at a write.

If all locks would be stored in adjacent memory locations, one processor’s write to its lock would invalidate all other cached copies of that block, ie, all the other locks. Performance disaster.

The solution is to store each spinlock in a separate cache block.

This applies to normal usage of spinlocks as well.

When processors write to the same cache block without actually communicating data between each other (ie, reading what another wrote) we suffer from what is called false sharing.
Cache Misses

- Cold miss: first time a variable is accessed.
- Capacity miss: miss due to cache size regardless of associativity.
- Conflict miss: miss due to limited associativity.
- True sharing miss: essential miss since it communicates data.
- False sharing miss: non-essential miss. We could have ignored the invalidation (this is an interesting research problem).
### Example of a False Sharing Miss

<table>
<thead>
<tr>
<th>Access</th>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Load 0</td>
<td></td>
<td>Cold miss</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Load 1</td>
<td>Cold miss</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>Store 1</td>
<td>Invalidation</td>
</tr>
<tr>
<td>4</td>
<td>Load 0</td>
<td></td>
<td>False sharing miss</td>
</tr>
</tbody>
</table>

### Effects of larger cache block size

- Increased benefit from spatial locality (prefetching within block)
- The larger risk of suffering from false sharing.
**Example of a True Sharing Miss**

<table>
<thead>
<tr>
<th>Access</th>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Load 0</td>
<td></td>
<td>Cold miss</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Load 1</td>
<td>Cold miss</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>Store 1</td>
<td>Invalidation</td>
</tr>
<tr>
<td>4</td>
<td>Load 0</td>
<td></td>
<td>True sharing miss</td>
</tr>
<tr>
<td>5</td>
<td>Load 1</td>
<td></td>
<td>Reads a new value</td>
</tr>
</tbody>
</table>

- While we *cannot* know it at the time of Access 4, that miss is a true sharing miss (which we realise at Access 5).
Data prefetching means fetching a memory block to the cache. The cache coherence protocol is used and the directory is updated. Either the CPU or the programmer can initiate prefetching. Hardware does this by noting when there are cache misses to subsequent blocks.

- Waiting for memory can be reduced or eliminated for some data
  - Prefetching too early may replace data which is still needed
  - Prefetching unused data wastes traffic
Programmers can use special prefetch instructions which are similar to load but do not put the data in a register.

A load or store to an invalid address gets a hardware exception and kills the program (segmentation fault).

Prefetch instructions are allowed to ask for invalid addresses.

The effect is simply to ignore the prefetch request.

There are usually two versions:

- prefetch for reading, or shared mode prefetch
- prefetch for writing, or exclusive mode prefetch

Some compilers use prefetch during optimization.
void __builtin_prefetch(const void* addr, int wr, int loc);

- The cache block of addr is prefetched
- The wr parameter specifies reading (0) or writing (1)
- The loc parameter should have a value from 0 to 3
- Zero means no temporal locality and three high temporal locality
Software inserted prefetch

- Advantages with prefetch instructions
  - can prefetch with more precision
  - can prefetch in exclusive mode to get ownership to help sequential consistency, and reducing network traffic

- Disadvantages with prefetch instructions
  - instruction overhead
  - bigger risk of register spilling (register allocation fails)
  - it is often very difficult to know if prefetch really is needed
  - In T. Mowry’s PhD thesis at Stanford compiler controlled prefetch was found to be useful for numeric codes
  - For codes with lists and trees it is much much more difficult to use prefetch since the address to prefetch (a few pointer dereferences away) is unknown
  - For library codes such as memcpy which copies an array it is impossible to know at compile time if prefetch is a good idea
The Postgres database is about one million lines of C code

Very mature code

Very optimized

Martin Lindblom and Fredrik Strandin used cleverly inserted prefetch to improve the performance.

https://lup.lub.lu.se/student-papers/search/publication/4939915

https://jonasskeppstedt.net/theses
Download statistics

Over 1000 downloads!!
A paper by Michel Dubois and me evaluated the following:

- Treat L2 cache misses as lightweight exceptions — there soon will not be much to do for the processor to do anyway.
- Such exceptions do not involve the OS kernel but simply jump to a special place in the program.
- For certain references in certain loops, the compiler has created an exception handler which will program a prefetch engine.

The exception handler is a part of the function’s control flow graph so it has access to all local variables which are register allocated both for the function and the exception handler.

- Therefore the exception handler can compute what (address and how much) to prefetch while the L2 cache miss is being serviced.
- Prefetch is done in shared or exclusive mode
- The instruction overhead of always prefetching is removed.

https://dl.acm.org/citation.cfm?id=347333
Four data streams can be prefetched concurrently
The basic instruction is \texttt{dst} — data stream touch
One of the instruction fields is a two bit stream selector
Other parameters:
- Prefetch unit size $S$ in 16-byte blocks from 1 to 32
- Number of units to prefetch
- Distance $D$ in bytes between two units, called the stride
Data stream touch POWER instructions

- `dst` for read and fetch into cache
- `dstst` for store and fetch into cache
- `dsttt` for read and fetch into special buffer
- `dststt` for store and fetch into special buffer
- `dss` stop a certain data stream prefetch engine
- `dssall` stop all data stream prefetch engines
- Implement exception based prefetch engine using:
  - Performance monitor exceptions
  - An adapted Linux kernel to reduce cost of exception
  - POWER prefetch engines
dcbz memory instruction on POWER

- Data cache block zero
- This instruction sets a cache block to zero
- Why can that be better than using a for loop?

```c
for (i = 0; i < 16; i += 1)
    a[i] = 0;
```

- Of course fewer instructions but are there other advantages?
- Assume the block is not in the cache
for (i = 0; i < 16; i += 1)
    a[i] = 0;

- At the first write, the cache requests the complete block
- The cache does not know the entire block will be overwritten (if it will)
- Copying the data from memory to the cache consumes network traffic
We have seen:

- Different parallel architectures
- Different programming languages
- Different paradigms make multicore programming more or less convenient
- If we want to maximize performance, we need to partition the data and balance the computations
- The language is probably not the most important decision (but see productivity below)
- Sequential consistency is a basic memory model
- Relaxed memory models allow more performance but more care
Multicore programming productivity

- Depending on the application and language, try a parallelizing compiler, or use OpenMP to incrementally parallelize a program.
- Try transactional memory and if that is sufficiently good, fine, but you need to consider what to do if there are numerous transaction restarts.
- Start with a simple solution with locks and condition variables.
- Use tools to detect data races.
- Use tools to detect risks for deadlocks.
- If the performance is insufficient, investigate atomic variables.
- Regardless of the language and synchronization primitives:
  - The best speed improvement is from a non-working to a working program (J. Ousterhout, professor at Stanford).
  - Ensure good load balancing.
  - Try to make data private to avoid locks.
  - Minimize cache misses.
  - Avoid false sharing – especially with transactional memory.
Rest of the course

- I will finish Lab 6 on: Rust, Clojure, HTM, OpenMP and parallelizing compilers
- I will publish, a growing, list of typical exam questions
- I will enable Forsete on power.cs.lth.se and a highscore list
- I will grade the exam and I expect you will do very well on it :)
- Deadline for the competition is November 30, at 23:59
- The project is the labs and the voluntary competition: no extra handin
About the exam

- See reading advice on course page
- There will be questions related to the labs and lecture notes (obviously)
- I will not ask about details such as language syntax
- There will be no questions where you will write code (or pseudo code)
- The questions are about "principles" and some "facts"
- What is meant by release and acquire belongs to "principles"
- I will not ask you to do dependence analysis but may ask about why or why not a certain loop might be executed in parallel. (2p)
- I may ask one simple question about history (2p).
- Max 60 p, grade = \left\lfloor \frac{\text{score}}{10} \right\rfloor