Knobelf

Parallele Programmierung ZF

O-Notation
O(t) = (Work / span)
Work = work done O(linear)
Span = $t_1 / p + t_{inf}$ = O(lin.)/p + O(inf)

Parallelism
Distributed Mem. → no problem
Shared Mem. → data races etc.
- Simultanious Multithreading SMT
- Multicores
- Symmetric Multicore Processoring SMP
- Non Uniform Memory Access NUMA

Distributed Data
+ No coordination
− Messages for data access

Shared Data
- Mutual exclusion (Mutex)

Task Parallel
- Programmer defines Tasks • Generic

Data Parallel
- Set of Operations is applied simultaniously on individual items.
- Work partitioning by System
  → Programmer decides what, not how

Bad Interleaving
- "high level race condition"
- Problem with unfavorable execution

Data Races
- "low level race condition"
- If two threads access the same memory location → data inconsistency
- Not every instruction is executed in one step

avoid this: use one of the following three
- thread local memory
- Immutable memory → do not write/change (finals)
- Synchronized

Correctness
Mutual Exclusion
Freedom from Deadlock
Freedom from Starvation
Lock-free // Wait-free

	non Blocking	Blocking
all make progress	Wait-free	Starvation-free
1 makes progress	lock-free	Deadlock-free

Deadlock
Two or more processes are mutually blocked because each process waits for an other to proceed

Starvation
Repeated but unsuccessful atempt of recently unblocked process to continue its execution

Livelock
Competing processes are able to detect a deadlock but make no progress solving it.

Lock-free
At least one makes progress (⇒ not starvation free)

Wait-free
All make progress ⇒ Starvation free

avoid Deadlocks
create unique order, get one with smaller first.

Single Write Multiple read

Validation
- Find Elements, • load then,
- Check condition again, • If it holds, do... else repeat

Lazy list Remove
- Find nodes • lock them
- Mark the one to be deleted • redirect pointer

Lock-free-programming
- read data • modify data
- Compare and Set • if CAS returns false repeat

Producer and Consumer
- A producer does sth with the data and enqueues it to the array.
- A consumer takes it from there and continues working with it.

Datastructure
Circular array with enqueue and dequeue pointer
in = ... out = ...

Coordination
barrier , send / receive

Linearizability
(→ G) ⊂ (→ S) is required

If it is possible to arrange the events on the timeline, so that a given output is true

Invocation A: q.do(x)
Result A: q.result

History
is linearizable if possible to extend it to a legal sequential history
→ Sequence of invocations and Results

pending: if invocation does not process a result.
Complete Subhistory: if not pending

Sequential History
- Method calls of different threads do not interleave
- final pending is ok

Well formed History
- if proj. per thread are the same.

Equivalent if H|A == G|A and H|B == G|B

Legal History
- if proj. on every obj. is sequential
- Methods preceding, overlapping

Projections

Objects		Threads	
	A: q.do();		B: q.do();
	A: q.result();		B: q.result();
	B: q.do();		B: e.do();
			B: e.result();

Realtime Order
(→ G) ⊂ (→ S)
(→ G) = {a → c, b → c}
(→ S) = {a → b, a → c, b → c}

Sequential Consistency
If G can be extended to legal sequential History S for each thread
- Timelines können gegeneinander verschoben werden!

(→ G) ⊂ (→ S) is not required
Operations need to respect program order but not realtime order
- possible to reorder operations done by different threads

Power-supply-Wall CPU can't get enough power from PSU
Power-dissipation-wall CPU can't dissipate heat → overheating

Transactional Memory (Hazard Pointers)
Summerizes executions to transactions. If two transactions access to the same memory element. One transaction gets reversed.

Spinlock
- No notification mechanism.
- Computing resources wasted, overall performance degraded, particularly for long-lived contention
- Scheduling fairness / missing FIFO behaviour. Solved with Queue locks.

Backoff
Sleep or wait instead of while for waiting

Bakery Algorithm (Ticket-system) FIFO

Peterson Algorithm
- satisfies mutual exclusion • starvation free

Programming Model: What needs to be done (declarative)

Concurrent = gleichzeitig

Moore's law
- 2005: transistors ×2 for 2 Years
- Exponentially faster

Problem: Heat + Power

Single-Core	Multi-Core (Hardware)
+ Flexibility	+ Simpler controll HW
+ Performance	+ More data throughput
− Complex controll HW	+ More power efficient
− Expensive Power	− complex programming

Key Questions: always? why? better? sort by quality? Termination? Instructions unambiguous?

Prove correctness
- Induction
- Invariant // assertion

JVM Java Virtual Machine
- Interprets compilation code
- Simulates CPU
- Translates compilation code to machine code

Performance loss with ||
- Context switches
- loss of locality
- CPU scheduling time vs. running time
- Additional overhead with synchronisation

Concurrent: requests for limited resources, manage access to "synchronisation" shared resources (possible on single cycle CPU)

Parallelism: Work on resources, use extra resources to "coordinate" solve a problem faster

Principle of locality easier to access local data

Sorts of Parallelism
- **Vectorisation** n times same in parallel
- **Pipelining** same thing but at different states
- **ILP:** Instruction level Parallelism
  - → Pipelining
  - → Super Scalar CPU (mult. Instr. per cycle)
  - → Out of order execution
  - → Speculative execution

Throughput: Input // Output datarate (1 / längste Zeiteinheit)

Latency: Time for one object to pass
- 1 Obj.: Add all steps
- n Obj.: Overall time / # Obj.

Ballanced Pipelines every unit → same timestep

Execution Time:
- T_1 = sequential Time
- $T_P \geq T_1 / P$

Speed-Up: $S_P = T_1 / T_P \leq P$

Efficiency: S_P / P

Amdahl's law problem size is constant (fixed amount of work)

$T_1 = W_{ser} + W_{par}$

$T_P \geq W_{ser} + W_{par}/P$, $T_P \geq$ crit. path

$W_{ser} = f \cdot T_1$, $W_{par} = (1-f)\, T_1$

$S_P \leq \dfrac{1}{f + \frac{1-f}{P} + a_P}$; f = non parallelizable serial fractions, a_P = Overhead

- + maximum speedup
- − limit on scalability
- − all non-parallel parts can cause problems!

Gustavson's law runtime is constant $T_1 = T_{inf}$

$W = P(1-f)\, T_1 + f\, T_1$

$S_P = f + P(1-f) = P - f\,(P-1)$

Reentrant lock (aquire lock it already owns)
```
lock.acquire
lock.release
... finally { lock.unlock() }
Condition
Final condition NotFull = lock.newCondition();
NotFull.await();
        signal();
        signalAll();
while (isFull) { await(NotFull); lock.unlock(); }
```

Synchronized (Expression) {...}
```
synch... (Obj.) {...}
synch... (this) {...}
synch... (lock) {...}
Synch... methodname () {...}
... methodname () {... synch... (int i) {...} }
```

Consensus
You cannot implement lock-free algorithms that require CAS with atomic registers.

Message Passing Interface (MPI)
- Hides Software / Hardware details
- Portable, flexible
- Implemented as a library

Synchronous Message Passing
- sender blocks until message is received

Asynchronous Message Passing
- sender does not block (fire-and-forget)
- placed into a buffer for receiver to get

Bitonic Sort
Parallel algorithm for sorting
- Sequential: $O(n \log^2 n)$
- Parallel: $O(\log^2 n)$

Read Modify Write

TAS Test-And-Set returns boolean
```
if (mem.ref[s] = 0) {
    mem[s] = 1;
    return true; }
else return false;
```
eg: while (!TAS(integer))...

CAS Compare-And-Swap Java: Boolean, Else: old value
```
atomic int;
i.compareAndSet(j)
if (old = oldval) {
    old = a;
    return true; }
else return false;
boolean attemptMark (V expected ref., boolean new Mark)
boolean compareAndSet (V expected ref., V new ref., boolean exp. Mark, boolean new Mark)
```
eg. while (!CAS(lock, 0, 1))

TTAS test Test And Set
```
public void lock() {
    do
        while (state.get()) {}
    while (! state.compareAndSet(false, true)) }
public void unlock() { state.set(false) }
```

Semaphores atomic counter (can be acquired if > 0)
```
acquire (s) {
    wait until s > 0;
    dec (s); }
release (s) { inc (s) }
```

Monitors allows mutual exclusion and wait(); synchronized in JAVA
```
wait();
notify();
notifyAll();
```

Volatile
```
private volatile int x = 0; ...
```
x is treated like a synchronized and written into memory directly

Oblivious
"Starts knowing nothing", makes the same comparisons regardless of the input.

Redundant
"Überflüssig", expendable comparisons

Interface
No code sharing

```
public interface Shape {
    public double area ();
    public double perimeter (); }

public class Rectangle implement Shape {
    public Rectangle (...) { }
    @Override
    public double area () {...}
    @Override
    public double perimeter () {...} }
```

Runable

```
public class MyRunnable implements Runnable {
    public MyRunnable...
    @Override
    public void run () {...} }

public class Use {
    Thread t1 = new Thread (new MyRunnable());
    t1.start ();
    Thread [ ] t = new Thread [5];
    t[0] = new Thread (new MyRunnable(...));
    t1.join (); }
```

Extend Thread

```
public class MyThread extends Thread {
    @Override
    public void run () {...} }

public class Use {
    MyThread t1 = new MyThread ();
    t1.start ();
    t1.join (); }
```

Tasks:
fast context switch, cheap to create, can create many of them to enable finegrained parallelism, matches with short lifetime of web requests

Threads:
more expensive to create, reuse a pool of threads (sized to match # cores), apply work stealing to achieve load balancing.

Callable
interface with returntype

```
public class MyCallable implements Callable <Integer> {
    public MyCallable () {...}
    @Override
    public Integer call () { } }

public class Use {
    public static void main (String args[]) {
    Callable <Integer> thisCallable = new MyCallable ();
    ExecutorService pool = Executor.newFixedThreadpool;
    Future <Integer> result = pool.submit.thisCallable;
    int a = result.get ();
    pool.shutdown (); }
```

Fork / Join

```
public class fjTask extends RecursiveTask <V> {
    @Override
    public V compute () {
    if small enough do it yourself
    else
    RecursiveTask <V> l = new fjTask (...);
      "        "      r =    "
    l.fork ();
    V a = r.compute ();
    V b = l.join (); }
    return V; }

public class Use () {
    ... {
    ... static ForkJoinPool f1 = new ForkJoinPool ();
       Result = f1.invoke (new fjTask (input)); }
```

Reduce Lock Contention

- Reduce duration locks are held (smaller synchronized blocks)
- Reduce frequency of lock requests (lock splitting or stripping)
- Replace exclusive locks with coordination mechanisms that permit greater concurrency (R/W locks, non-bl. DS)
- Split up a lock into smaller locks (fine-grained locking)
- Avoid locks entirely (lock-free, replication, immutability)
- Transactional memory
- fairness model

Barrier Rendevouz for arbitrary number of threads
init (N), await (), reset ()

e.g.: Game of life, displaying the pixels in an image.

Threads t1
(share the same adress)
Terminates only if run() terminates

```
t1.start ();        // creates a new Thread
  .run ();          // same Thread do run()
  .join ();         // = finish
Backoff { .sleep ();  // use in while blocks
        { .notifyAll ();
  .setPriority ();  // 1-10
  .setName ("...");
  .getId ();
  .getState ();     // new, runnable, blocked, waiting, terminated
  .MAX_PRIORITY     // MIN_P...
  .interrupt ();    // can be ignored
  .isInterrupted (); // now?
  .interrupted ();  // Present
```

Threadpool
↳ Callable
use seq. cutoff 500 - 1000

Concurrent
+ Concurrent collection allows multiple threads to be reading / modifying it in parallel
- hard to make compound operations atomic, since you cannot use client-side locking

Problem: Load balancing **Solution:** Dynamic Sceduling

API design properties
declarative, no storage, functional, ~~easy~~ easy to parallelize
less flexibility than for-loop, operators must be stateless.

Data - Parallelism
+ maps, reductions, filter
- complex synchronizations, computation with complex computation patterns

Multithreaded Programs
Are hard to test because of nondeterminism from scheduling. Hard to track down bugs (subsequent runs)

Problems in programs using locks
- performance, lock contention
- Deadlocks
- Priority Inversion
- Livelock

```
class ReadersWritersLock {
    int writers = 0;
    int readers = 0;
    int writersWaiting = 0;

  • synchronized void AcquireRead () {
      while (writers > 0 || writersWaiting > 0) {
          try { wait (); }
          catch (InterruptedExeption e) {} }
      readers ++; }
  • synchronized void ReleaseRead () {
      readers --;
      notifyAll (); }

  • synchronized void AcquireWrite () {
      writersWaiting ++;
      while (writers > 0 || readers > 0 ) }
          try { wait (); }
          catch (InterruptedExeption e ) {} }
      writersWaiting --;
      writers ++; }

  • Synchronized void ReleaseWrite () }
      writers --;
      notifyAll (); }
```

```
=   set equal
==  equal ?
a.equals (b)      returns boolean

String a

a.indexOf (str);
 .length ();
 .substring (index 1, index 2);
 .toLowerCase ();
 .toUpperCase ();
 .equals (string);
 .equalsIgnorCase (string);
 .startsWith (str.);
 .endsWith (str.);
 .contains (str.);
```

```
public class NameClass {

    public NameClass (int i, ...) } {
        // Define "Creation Method"
    }
    NameClass NameObject =
        new NameClass (parameters);
}
```

```
public class Use {
    int a, b, c, x1, flowow ;

    public static void main (string args []) {
     [ if (...) {...}
     [ if else (...) {...}
     [ else {...}
     - System.out.println (a + "..." + "/N");
     - while (...) {}
     [ do {...}
     [ while {...}
      Scanner s = new Scanner (system.in);
      int d = s.nextInt;
      string e = s.nextLine ();
             e = s.lastString (); }

    public static int f (int i) {
        return i ; }
}
```

```
public class dog extends animal() {
    redefine ...
    @Override
    ... }

    for (int i = 0; i < 100; i++) { }

    switch (Data) {
        case 1 : ...
        case 2 : ...  }
```

```
import static java.lang.Math.* ;

    Math.max (last, first)   (min)

    System.out.println (...);
```

ECCO