Multithreading Grundlagen

Multi-Thread Programmierung

JVM Thread Modell

Java ist ein Single Process System. JVM ist ein Prozess im Bsys. **Main-Thread** wird beim Aufstarten der JVM anhand *main()* Methode erzeugt. JVM läuft, solange Threads laufen (Ausnahme Daemon Threads).

```
var myThread = new Thread(() -> { /* ... */ });
myThread.start();
```

Thread wird erst bei *start()* erzeugt. Führt *run()*-Methode des Runnable Interface aus. Thread endet beim Verlassen von *run()*.
Nicht-Determinismus: Threads laufen ohne Vorkehrungen beliebig verzahnt oder parallel.

Explizite Runnable-Implementation:

```
class SimpleLogic implements Runnable {
   @Override
   public void run() { /* ... */ }
}
var myThread = new Thread(new SimpleLogic()).start();
```

Sub-Klasse von Thread

```
class SimpleThread extends Thread {
   @Override
   public void run() { /* ... */ }
}
var myThread = new SimpleThread().start();
```

Thread Join

Warten auf Beendigung eines Threads. *t2.join()* blockiert, solange t2 läuft.

Thread Passivierung

Thraed.sleep(ms): Laufender Thread geht in Wartezustand, dann ready. **Thread.yield():** Gibt Prozesor frei, direkt ready.

Monitor Synchronisation

Java Synchronized Methoden

```
synchronized void f() { /* ... */ } // Object Lock
static synchronized void g() { /* ... */ } // Class Lock
```

Monitor

Objekt mit internem gegenseitigem Ausschluss. Nur 1 Thread operiert im Monitor. Alle äusseren Methoden synchronized. **Wait & Signal Mechanismus:** Threads können im Monitor auf Bedingung warten und wartende aufwecken (signal).

```
public synchronized void withdraw(int a) {
   while (amout > balance) { wait(); }
   balance -= a;
}
public synchronized void deposit(int a) {
   balance+= amount; notifyAll();
}
```

notify(): Bei Uniform Waiters & One-In-One-Out Bedingungen. **notifyAll():** Bei mehreren Bedingungen / One-In-Multiple-Out. **Pauschales wait & signal:** Wartende müssen selber schauen, ob sie ein Signal interessiert. **Signal and Continue:** Signalisierender Thread behält Monitor nach notify. Aufgeweckter Thread muss um Monitor-Eintritt kämpfen.

Spezifische Synchronisationsprimitiven

Arten

Faire Semaphoren: *new Semaphore(N, true)*, FIFO Warteschlange, langsamer.

```
private Semaphore upperL = new Semaphore(CAP, true);
private Semaphore lowerL = new Semaphore(0, true);
public void put(T item) throws InterruptedException {
   upperL.acquire();
   synchronized (queue) { queue.add(item); }
   lowerL.release();
}
public T get() throws InterruptedException {
   T item; lowerL.aquire();
   synchronized (queue) { item = queue.remove(); }
   upperL.release(); return item;
}
```

Multi-Acquire/Release: *acquire(N)*.

Lock & Condition

Monitor mit mehreren Wartelisten für verschiedene Bedingungen. **Lock-Objekt:** Sperre für Eintritt in Monitor. **Condition-Objekt:** Wait & Signal für bestimmte Bedingung.

```
private Lock monitor = new ReentrantLock(true); // fair
private Condition nonFull = monitor.newCondition();
private Condition nonEmpty = monitor.newCondition();
public void put(T item) throws InterruptedException {
   monitor.lock();
   try {
      while(queue.size() == Capacity) { nonFull.await(); }
      queue.add(item);
      nonEmpty.signal();
   } finally { monitor.unlock(); }
}
public T get() throws InterruptedException {
   monitor.lock();
   try {
      while(queue.size() == 0) { nonEmpty.await(); }
      T item = queue.remove();
      nonFull.signal();
      return item;
   } finally { monitor.unlock(); }
}
```

Read-Write Locks

Gegenseitiger Ausschluss ist unnötig streng für lesende Abschnitte. Erlaube parallele Lese-Zugriffe.

```
private Collection<String> names = new HashSet<>();
private ReadWriteLock rwLock = new ReentrantReadWriteLock();
public boolean exists(String pattern) { // Read−only accesses
   rwLock.readLock().lock();
   try {
      return names.steram().anyMatch(n -> n.matches(pattern));
   } finally { rwLock.readLock().unlock(); }
}
public void insert(String name) { // Write / Read accesses
   rwLock.writeLock().lock();
   try {
      names.add(name);
   } finally { rwLock.writeLock().unlock(); }
}
```

Count Down Latch

Synchronisationsprimitive mit Count Down Zähler. Threads können warten, bis Zähler <= 0 wird. **await():** warten bis Count Down 0 ist. **countDown():** Zähler - 1.

```
var ready = new CountDownLatch(N); // Warte auf N cars
var start = new CountDownLatch(1); // Einer gibt signal
// N Cars:
ready.countDown(); start.await();
// RaceControl:
ready.await(); start.countDown();
```

Cyclic Barrier

Treffpunkt für fixe Anzahl Threads. Anzahl treffender Threads muss vorgegeben sein. Ist wiederverwendbar (mehrere Runden).

```
var start = new CyclicBarrier(N); // Treffende Autos
// N Cars:
start.await(); // braucht kein Race Control mehr

var gameRound = new CyclicBarrier(N);
// N Players:
while(true) {
   gameRound.await(); // play concurrently with others
}

// Mit Austausch
Exchanger.exchange(something);
// Blockiert bis anderer Thread auch exchange() aufruft.
```

Gefahren der Nebenläufigkeit

Neue Arten von Programmierfehler, die es bei single-Threading nicht gibt. Können sporadisch oder selten auftreten. Sehr schwierig durch Tests zu finden.

Race Condition

Mehrere Threads greifen auf gemeinsame Ressourcen ohne genügend synchronisation zu. Mögliche falsche Resultate oder falsches Verhalten (Lost Updates). Ursache oft ein Data Race, nicht immer. Keine Atomarität bei Time of Check / Time of Use

Data Race

Unsynchronisierter Zugriff auf gleichen Speicher. Selbe Variable oder Array Element (min. 1 schreibender Zugriff).

Race Condition ohne Data Race

Critical Sections nicht geschützt. Data Races mit Synchronisation eliminiert, aber nicht genügend grosse synchronisierte Blöcke.

```
synchronized int getBalance() { return balance; }
synchronized void setBalance(int x) { balance = x; }
// Mehrere Threads, Kein Atomares Inc − Lost Update moeglich
accout.setBalance(account.getBalance() + 100);
```

Kombinationen

Alles Synchronisieren? Hilft nichts. Race Condition trotzdem möglich. Weitere Nebenläufigkeitsfehler. Synchronisationskosten sind relativ teuer.

Synchronisation: Verzichtbare Fälle

Immutability (Unveränderlicheit): Objekte mit nur lesendem Zugriff. **Confinement (Einsperrung):** Objekt gehört nur einem Thread zu einer Zeit.

Immutable Objects

Instanzvariablen alle *final*. Primitive Datentypen. Referenzen wiederum auf Immutable Objekte. Methoden mit nur lesendem Zugriff. Konstruktor initialisiert Instanzvariablen. Nach Konstruktor kann Objekt ohne Synchronisation von Threads verwendet werden.

Confinement

Struktur garantiert, dass auf ein Objekt nur durch einen Thread zur gleichen Zeit zugegriffen wird. **Thread Confinement:** Objekt gehört nur einem Thread und wird nur von demjenigen verwendet. **Object Confinement:** Objekt in anderem bereits synchronisierten Objekt eingekapselt.
Kapselungsbrüche: 1. Inneres Objekt ist aussen zugreifbar. 2. Rückgabe einer Referenz auf inneres Objekt. 3. Holder installiert selber Referenz ausserhalb. 4. Inneres Objekt gibt selber *this* raus.

Thread Safety

Klassen / Methoden, die intern synchronisiert sind. Keine Race Conditions innerhalb dieses Codes. Kritischer Abschnitt nur pro Methode erfüllt. **Aber:** Kein kritischer Abschnitt über mehrere Methodenaufrufe. Andere Nebenläufigkeitsfehler möglich.

Java Collections - Thread Safety

Alte Java 1.0 Collections (Vector, Stack, Hashtable): **JA**. Moderne Collections (HashSet, TreeSet, ArrayList, etc.): **NEIN**. Concurrent Collections (ConcurrentHashMap, etc.): **JA**.

Verstecktes Multi-Threading

Finalizers: Laufen über separaten Finalizer-Thread. **Timers:** Handler durch separaten Thread ausgeführt (ausser GUI). **Externe Libraries & Frameworks:** z.B. Abarbeitung von Web-Service Aufrufen.

Deadlock

Beide Threads sperren sich gegenseitig aus:

Spezialfall: Livelocks

Threads haben sich gegenseitig permanent blockiert. Führen aber noch Warteinstruktionen aus. Verbrauchen CPU während Deadlock.

```
// Thread 1
b = false; while (!a) { } b = true;
// Thread 2
a = false; while (!b) { } a = true;
```

Deadlock Erkennung

Deadlock = Zyklus im Betriebsmittelgraph
Deadlock Voraussetzungen: Geschachtelte Locks, Zyklische Warteabhänigkeiten

Deadlock Vermeidung

Lineare Sperrordnung der Ressourcen einführen. Nur geschachtelt in aufsteigender Reihenfolge sperren. Eliminiert zyklische Warteabhänigkeiten.
Grobgranulare Locks wählen. Wenn lineare Sperrordnung nicht möglich/sinvoll ist. Sperre gesamte Bank bei Kontenzugriff. Eliminiert Schachtelung von Locks.

Starvation

Ein Thread kriegt nie die Chance, auf eine Ressource zuzugreifen, obwohl sie immer wieder frei wird. Andere Threads überholen andauernd. Liveness/Fairness Problem.

```
do { // Starvation moeglich
   success = account.widthdraw(100);
} while (!success);
```

Vermeidung

Faire Synchronisationskonstrukte (bei Semaphore, Lock & Condition, ReadwriteLock möglich). Java Monitor hat ein Fairness Problem (Starvation anfällig).

Parallelität Korrektheitskriterien

Safety: Keine Race Conditions, Keine Deadlocks. **Liveness:** Keine Starvation.

Thread Pool

Work-Stealing: Vorteil: Effizienz duch weniger Contention. Nachteil: Fairnessprobleme bei unausgeglichenen Verteilungen.

Vorteile

Beschränkte Anzahl von Threads: Viele Threads verlangsamen das System oder überschreiten Speicherlimit. **Recycling der Threads:** Spare Thread-Erzeugung und Freigabe. **Höhere Abstraktion:** Trenne Task-Beschreibung von Task-Ausführung. **Anzahl Threads pro System konfigurierbar:** #Worker Threads = #Prozessoren + #I/O-Aufrufe

Einschränkung

Daemon: Laufen nicht garantiert zu Ende bei Fire & Forget. Tasks dürfen nicht aufeinander Warten, sonst Deadlock. **Run to Completion:** Task muss zu Ende laufen, bevor Worker Thread anderen Task ausführen kann. **Ausnahme:** geschachtelte Tasks.

Java Fork-Join-Pool

```
var threadPool = new ForkJoinPool();
Future<Integer> future = threadPool.submit(() -> { });
Int result = future.get(); // Blockiert bis Task beendet
```

Rekursive Task Erstellung

Anzahl Tasks: Java: Arraylänge, THRESHOLD. .NET: # Freien Worker Threads falls <Arraylänge

```
class CountTask extends RecursiveTask<Integer> {
   // Constructor
   @Override
   protected Integer compute() {
      // no / single element => return result
      // Calculate lower, middle, upper
      var left = new CountTask(lower, middle);
      var right = new CountTask(middle, upper);
      left.fork(); right.fork();
      return right.join() + left.join(); } }
// Ausfuehrung
var threadPool = new ForkJoinPool();
int res = threadPool.invoke(new CountTask(2, N)); // blockiert
```

Fork Join Pool Internals

Automatischer Parallelitätsgrad: Default: #Worker Threads = #Prozessoren Dynamisches Hinzufügen / Wegnehmen von Threads
Common Pool: Verhindert Engpässe durch zu viele Thread Pools.

Asynchrone Programmierung

Unnötige Synchronität: Langlaufende Rechnungen, I/O Aufrufe.
Asynchroner Aufruf: Aufrufer soll während der Operation weitermachen.

```
// Klassisch
Future<long> future = threadPool.submit(() -> { });
   // other work
process(future.get());
// Modern, runAsync() falls kein Rueckgabewert
CompletableFuture<long> future =
                CompletableFuture.supplyAsync(() -> { });
   // other work
process(future.get());
```

Ende des async Aufrufs

Caller-zentrisch (Pull): Caller warted auf Task-Ende und holt sich Resultat, Future abfragen. **Callee-zentrisch (Push):** Async Operation informiert direkt über Resultat. Completion Callback.

Continuation

Folgeaufgabe an asynchrone Aufgabe anhängen.

```
// thenApply() fuer Continuation mit Rueckgabe
future.thenAccept(res -> System.out.println(res));
```

Ausführung: durch beliebigen Thread, durch Initiator, wenn Future bereits Resultat hat. **Asynchrone Continuations:** *thenAcceptAsync()* bzw. *thenApplyAsync()*.

Multi-Continuation

```
// .allOf(futures).thenRun(() −> syso());
CompletableFuture.allOf(f1, f2).thenAcceptAsync(() -> { }));
CompletableFuture.any(f1, f2).thenAcceptAsync(() -> { }));
```

Fire and Forget

Task starten, ohne das Ende abzuwarten. Submitter ruft kein *get()* oder *join()* auf.

```
CompletableFuture.runAsync(() -> { });
```

Daemon Workers: Workers Threads in Fork-Join-Pools sind Daemon. Anwendung kann vor Task-Ende stoppen.
Ingorierte Exceptions: Exceptions in Fire & Forget Task werden ignoriert.

Task Parallel Library

.NET Threads

Keine Vererbung: Delegate bei Konstruktor. **Exception in Thread:** Abbruch des Programs.

```
var myThread = new Thread(() => { /* ... */ });
myThread.start(); /* ... */ myThread.join();
```

C# Lambda kann umgebende Variablen zugreifen (auch schreibend).

Monitor in .NET

FIFO Warteschlange, Pulse informiert längst Wartenden. *Wait()* in Schlaufe. *PulseAll()* bi mehreren Bedingungen oder Erfüllungen mehrerer Threads. Synchronisation mit HilfsObj als Best Practice.

```
private object syncObject = new(); // Monitor auf HilfsObj
public void Widthdraw(decimal amount) {
   lock (syncObject) {
      while(amout > balance) { Monitor.Wait(syncObject); }
      balance -= amount; } }
public void Deposit(decimal amount) {
   lock (syncObject) {
      balance += amount;
      Monitor.PulseAll(syncObject); } }
```

.NET Synchronisationsprimitiven

Fehlen: kein Fairnessflag, kein Lock & Condition. **Zusätzlich:** ReadwriteLockSlim für Upgradeable Read/Write. Semaphoren auch auf OS-Stufe nutzbar. Mutex (binärer Semaphor). **Collections nicht Thread-safe:** Ausser *System.Collections.Concurrent*

.NET Task Parallel Library (TPL)

Work Stealing Thread Pool. **Verschiedene Abstraktionsstufen:** Task Parallelization: Explizite Tasks starten und warten. Data Parallelization: Parallele Statements und Queries. Asynchrone Programmierung: mit Continuation Style.

Thread Injection

TPL fügt zur Laufzeit neue Worker Threads hinzu. **Hill Climbing Algorithmus:** Misst Durchsatz & variiert Anzahl Worker Threads. **Kein Deadlock** bei Task Abhängigkeiten, aber ineffizient, nicht dafür gemacht.

Task Parallelisierung

```
Task task = Task.Run(() => { }); /* ... */ task.Wait();
// Task mit Rueckgabe
Task<int> task = Task.Run(() => { return 0; });
Console.Write(task.Result); // Blockiert
// Geschachtelte Tasks
Task.Run(() => {
   var left = Task.Run(() => { });
   var right = Task.Run(() => { });
   int res = left.Result + right.Result; });
```

Parallele Statements

Menge an Statements potentiell parallel ausführen. Als Task starten. Barriere der Tasks am Ende.

```
Parallel.Invoke(
   () => MergeSort(1, m); () => MergeSort(m,1);
);
```

Parallele Loop

Schlaufen-Bodies potentiell parallel ausführen. Gruppierung der Bodies in Tasks. Berriere dieser Tasks am Ende.

```
Parallel.ForEach(list, file => Convert(file));
Parallel.For(0, array.Length, i => Calc(array[i]));
```

Parallele Loop Partitionierung

Schlaufe mit vielen sehr kurzen Bodies ist ineffizient. TPL gruppiert automatisch mehrere Bodies zu Task. Aufteilung gemäss verfügbaren Worker Threads.
Explizite Partitionierung: Vorteil: Weniger Body-Delegates, Nachteil: Künstliche Unterschleife.

```
Parallel.ForEach(Partitioner.Create(0, array.Length),
   (range, _) => {
      for (int i = range.Item1; i < range.Item2; i++) {
         Calc(array[i]);
      } })
```

Asynchrone Programmierung mit TPL

Task Continuation

```
task1.ContinueWith(task2).ContinueWith(task3);
// Multi−Continuation
Task.WhenAll(task1, task2).ContinueWith(continuation);
```

```
Task.WhenAny(task1, task2).ContinueWith(continuation);
```

GUI and Threading

GUI Frameworks erlauben nur Single-Threading.
UI Thread: Loop zur Ausführung der Ereignisse aus einer Queue.

UI Thread Confinement

Wiso basieren GUI-Frameworks auf Single-Thread Modell? **Synchronisationskosten:** Locking in allen Komponenten und Methoden relativ teuer. **Deadlock-Risiko:** Bei zyklischen geschachtelten Aufrufen (z.B. MVC).

Swing: Dispatching an UI Thread

Komponentenzugriffe an UI Thread delegieren.

```
// Benutzung der Klasse SwingUtilities
static void invokeLater(Runnable doRun); // Async
static void invokeAndWait(Runnable doRun); // Synchron
// Example
button.addActionListener(event-> {
   new Thread(() -> {
      var text = readHugeFile();
      SwingUtilities.invokeLater(() ->
            { textArea.setText(text); }); }).start(); });
```

.NET UI Threading Modell

Gleiches Prinzip wie Java. **UI Thread:** Aufrufer von *Application.Run()*.
UI Event Dispatching: WPF: *control.Dispatcher.InvokeAsync(action)*
WinForm: *control.BeginInvoke(delegate)*

Async / Await

Async Methode läuft teilweise synchron, teilweise asynchron. Aufrufer führt Methode solange synchron aus bis ein *await* anliegt. Compiler zerlegt Methode in Abschnitte. Abschnitt nach Await läuft später nach Task-Ende (Continuation). Methode läuft synchron bis *await*, springt dann zurück zum Aufrufer.
Verschiedene Ausführungen: Fall 1: Aufrufer ist normaler Thread, Abschnitt wird durch TPL Worker-Thread ausgeführt. Fall 2: Aufrufer ist UI-Thread, Abschnitt wird als Event vom UI-Thread ausgeführt.

```
public async Task<int> LongOperationAsync() { }
Task<int> task = LongOperationAsync(); /* ... */
int res = await task; // Warte auf Beendigung
```

Async Rückgabetypen: *void*: fire-and-forget. *Task*: Kein Rückgabetyp erlaubt warten. *Task<T>* Rückgabetyp T.

```
async Task<string> ConcatAsync(string url1, string url2) {
   HttpClient client = new HttpClient();
   Task<string> d1 = client.GetStringAsync(url1);
   Task<string> d2 = client.GetStringAsync(url2);
   string site1 = await d1; string site2 = await d2;
   return site1 + site2;
}
```

Memory Models

Lock-Freie Programmierung: Korrekte nebenläufige Interaktionen ohne Locks. Garantien des Speichermodels nutzen.

Ursachen für Probleme

Weak Consistency: Speicherzugriffe werden in verschiedenen Reihenfolgen aus verschiedenen Threads gesehen. Ausnahme: Synchronisationen/Speicherbarrieren **Optimierungen:** Compiler, Laufzeitsystem und CPUs, Instruktionen werden umgeordnet/wegoptimiert.

Java Memory Model

Atomicity

Einzelnes Lesen / Schreiben ist atomar für: Primitive Datentypen, Obj-Referenzen, long und double nur mit *volatile* Keyword.
Unteilbarkeit ≠ Sichtbarkeit: Nach Write sieht anderer Thread vlt. noch alten Wert.

Visibility

Garantierte Sichtbarkeit zwischen Threads: Locks Release & Acquire, Volatile Variable, Thread/Task-Start und Join, Initialisierung von final Variablen.

Ordering

Program Order: 'as-if-serial', Sequentielles Verhalten jedes Threads bleibt erhalten. (Andere Threads dürfen es anders sehen) **Synchronization Order (Total Order):** Synchronizationsbefehle werden zueinander nie umgeordnet. **Happens-Before Relation (Partial Order):** Alles andere kann umgeordnet werden, ausser es gibt garantierte Sichtbarkeit unter den Threads.

Java Synchronization Order

```
// Keine Umordnung in Java, weil alles volatile
volatile boolean a = false, b = false;
a = true; while(!b) { } // Thread 1
b = true; while(!a) { } // Thread 2
// Nicht korrekt, da nicht atomar
private volatile boolean locked = false;
public void aqcuire() {
   while(locked) { }
   locked = true;
}
// Spin-Lock mit atomarer Operation
private AtomicBoolean locked = new AtomicBoolean(false);
public void acquire() {
   while (locked.getAndSet(true)) { }
}
public void release() { locked.set(false); }
// Atomares Compare and Set, setzt Update falls Wert gleich expect
boolean compareAndSet(boolean expect, boolean upadte);
// Optimistische Synchronization
do {
   oldValue = var.get(); newValue = calcChange(oldValue);
} while (!var.compareAndSet(oldValue, newValue));
```

.NET Memory Model

Unterschied zu Java: Atomicity: long/double auch ohne volatile atomar. Visibility: Nicht definiert, implizit durch Ordering. Ordering: nur Half und Full Fences.
Atomare Instruktionen: *Interlocked* Klasse

Volatile Half Fences

Volatile Write: Vorangehende Zugriffe bleiben davor. (Release Semantik) **Volatile Read:** Nachfolgende Zugriffe bleiben danach. (Acquire Semantik)

Full Fence: Memory Berrier

```
Thread.MemoryBarrier(); // Verbietet Umordnung in beide Richtungen
```

Actor Model

Async: nicht Blockierend.
Substantiell anderes Programmierkonzept. **Aktive Objekte:** haben nebenläufiges Innenleben. **Kommunikation:** Objekte senden und empfangen Nachrichten. **Kein Shared Memory:** Nur Austausch von Nachrichten.

Vorteile: Actor, CSP

Inhäerente Nebenläufigkeit: Alle Obj (Actors) laufen nebenläufig. Maschine kann Grad an Nebenläufigkeit ausnutzen. **Keine Race Conditions:** Kein Shared Memory. Nachrichtenaustausch synchronisiert implizit. **Gute Verteilbarkeit:** Kein Shared Memory. Nachrichtenaustausch für Netz prädestiniert.

Akka

Actor Model für JVM. Actors haben privaten Zustand, kann aber auf JVM nicht enforced werden. Eine Mailbox pro Actor: Ein Buffer für alle ankommenden Nachrichten. Asynchrones Senden.
Empfangsverhalten: Reaktion auf ankommende Nachricht. Behandlungsmethode wird ausgeführt. Effekte per Behandlung (Ändere Zustand, Sende Nachrichten, Erzeuge neue Actors). Intern sequentiell, nur eine Nachricht auf einmal behandelbar.

```
public class Printer extends UntypedActor {
   public void onReceive(final Object message) {
      if(message instanceof Integer) { } } }
// Erzeugen und Senden
ActorSystem system = ActorSystem.create("System");
ActorRef p = system.actorOf(Props.create(Printer.class));
for(int i = 0; i < 100; i++) {
   printer.tell(i, ActorRef.noSender()); // simple async
}
System.shutdown();
```

Anwendung: Alternative zu Threads, Transaction-Processing, Backend für Service, Kommunikations-Hub.
Actor Hierarchies: Passend zu URL Adressierungsschema. Erzeuger ist Parent. *ActorSelection* selektiert Teilbaum, broadcast möglich.

GPU Parallelisierung

512, 1024, 3584, 5760 Cores. Sehr spezifische langsamere Prozessoren.

GPU Aufbau

SP (Streaming Processor): 8-192 SPs pro SM **SM (Streaming Multiprocessor):** z.B. 1-30 SM **SIMD:** SM ist prinzipiell SIMD (Single Instruction Multiple Data), Vektorparallelität Alle Cores führen gleiche Instruktion aus, Einzelne können sie auch nicht ausführen.

GPU vs. CPU

GPU: hohe Datenparallelität, wenig Verzweigungen, Kein beliebiges Warten, Kleine Caches pro Core. Ziel: Hoher Gesamtdurchsatz
CPU: Gegenteil, Ziel: Niedrige Latenz pro Thread

NUMA Modell

Non-Uniform-Memory-Access: Kein gemeinsamer Hauptspeicher zwischen GPU und CPU. Explizites Übertragen.

CUDA - Computer Unified Device Architecture

CUDA Blocks: Threads sind in Blöcke gruppiert. Blöcke sind im gleichen SM. Threads können innerhalb Block interagieren. **Ausführungsmodell:** Thread: virt. Skalarprozessor. Block: virt. Multiprozessor. Blöcke müssen unabhängig sein. Run To Completion. Beliebige Ausführungsreihenfolge. Grad der Parallelität durch GPU bestimmt. Automatische Skalierung. **Ablauf:** 1. Auf GPU allozieren *cudaMalloc*, 2. Daten au GPU transferieren *cudaMemcpy*, 3. Kernel ausführen, 4. Rücktransfer, 5. Auf GPU deallozieren

Datenaufteilung

threadIdx.x: ThreadId im Block. **blockIdx.x:** Nummer des Blocks. **blockDim.x:** Blockgrösse. Programmierende modellieren Datenaufteilung selber.

Boundary Check

Falls Mehr Threads als zu bearbeitende Daten. Threads mit $i \geq N$ dürfen nicht auf Daten zugreifen.

Unified Memory

Automatischer Transfer CPU - GPU.

Speicherstufen

Shared Memory: Per SM. Schnell ca 4 Zyklen. Nur zwischen Threads innerhalb Block sichtbar. Paar KB. *__shared__ float x;* **Global Memory:** Main Memory in GPU. Langsam, ca. 400-600 Zyklen. Allen Threads sichtbar. Mehrere GB. *cudaMalloc()*

Block Barriere

```
__syncThreads();
```

In if/else nur falls für alle Threads eines Blocks.

Warp

Block wird intern in Warps zerlegt (je 32 Threads). **Ausführung:** SIMD. SM kann alle Warps eines Blocks beherbergen. Nur wenige laufen echt parallel (1 bis 24).

Divergenz

Unterschiedliche Verzweigungen im selben Warp. *if/switch/while/-do/for*. SM führt Intruktionen der einen Verzweigung durch, dann Instruktionen einer anderen Verzweigung. **Performance Problem** Schlechter Fall: Divergenz innerhlab derselben Warp. Guter Fall: Gleiche Verzweigung im Warp.

Memory Coalescing

Zugriffsmuster für Performance optimieren. **Burst:** Falls Threads auf 32-Byte-Bereiche zugreifen. Sonst teure Einzel-Zugriffe. (je 400 Zyklen pro Global Memory)
Tiles einlesen

Cluster Parallelisierung

Synchron: Send & Recv sind blockierend.
mpiexe -n 100 programm.exe
Motivation: möglichst hohe Beschleunigung. Viele CPU Cores statt nur viele GPU Cores. **Verteiltes Programmiermodel:** Programm auf mehreren Nodes ausführen. Kein Shared Memory (NUMA) zwischen nodes, nur für Cores im Node (SMP). **Message Passing Interface (MPI):** Basiert auf Actor/CSP Prinzip.

```
MPI_Init(&argc, &argv); // MPI Initialisierung
MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Prozess Identifikation
MPI_Finalize(); // MPI Finalisierung
MPI_Send(&val, 1, MPI_INT, recRank, tag, MPI_COMM_WORLD);
MPI_Recv(&val, 1, MPI_INT, senderRank, tag, MPI_COMM_WORLD,
                              MPI_STATUS_IGNORE);
MPI_Barrier(MPI_COMM_WORLD); // Wartet auf alle Prozesse
MPI_Allreduce(&val, &total, 1, MPI_INT, MPI_SUM,
   MPI_COMM_WORLD); // Aggregation von Teilresultaten
MPI_Reduce(&val, &total, 1, MPI_INT, MPI_SUM, recRank,
   MPI_COMM_WORLD); // Effizienter, kein Broadcast
```

SPMD: Single Program Multiple Data. MPI Programm wird in mehrere Prozesse gestart. Prozesse können untereinander kommunizieren. **Communicator:** Gruppe von MPI-Prozessen. Communicator World: Alle Prozesse einer MPI-Programmausführung.

Concurrency in Python

GIL: Global Interpreter Lock. Nur ein Thread kann Python Byte-Code ausführen. Kein Speedup für CPU-Bount Operationen möglich. Data Races durch Reordering dennoch möglich, Visibility nicht garantiert. Kein definiertes Memory Model.

```
from threading import Thread
from multiprocessing import Process
if __name__ == '__main__':
   t = Thread( # p = Process
      target=fibonacci,
      args=(10,)
   ).start();
   t.join();
# Shared Memory
res = Value('i', -1, lock=False); // Typ, InitialWert, lock
```

Pools

Threads: *concurrent.futures.thread.ThreadPoolExecutor* **Prozesse:** *concurrent.futures.process.ProcessPoolExecutor / multiprocessing.Pool*

```
if __name__ == '__main__':
   with ProcessPoolExecutor() as pool:
      future = pool.submit(fib, 10) # or: map(fib, [1,2,3])
      print(future.res()); # Blockiert bis Task-Ende
```

Asnychrone Programmierung (asyncio)

Coroutine-Functions: werden erst beim *await* ausgeführt. Keine parallele Ausführung. Ausnahme: Coroutine wird als Task verpackt.

```
async def sub_routine(n):
   await asyncio.sleep(n);
if __name__ == 'main__':
   asyncio.run(sub_routine(1))
```

JavaScript Concurrency

Grundsätzlich Single-Threaded mit einem Event-Loop. Kein Schutz vor Race-Condition.

```
function delay(ms) {
   let promise = new Promise((resolve, reject) => {
   setTimeout(() => resolve(), ms); }); }
async function countTo(n) {
   for (let = 1; i <= n; i++) {
      await delay(1000); } }
```

Web-Worker

Entsprechen einem Thread im Browser. Datenaustausch primär über Messaging. Werden mit Quelldatei gestartet. Langlebiger Prozess mit eigenem Event-Loop.

```
onmessage = event => { // Worker definieren
   const n = event.data; const res = fib(n);
   postMessage(res); }
const worker = new Worker('w.js'); // Worker verwenden
worker.onmessage = event => {
   console.log(event.data); }
worker.postMessage(42);
```