[Java Thread] In Java können Threads mittels Vererbung hergestellt werden. Hierzu gibt es die Basisklasse **Thread**. Beim Ableiten von der Klasse muss nur **public void run()** überschrieben werden. Für den Start des Threads muss dann die Methode **start()** aufgerufen werden. EIN AUFRUF VON **run()** WÜRDE BLOCKIEREN!

```
class MyThread extends Thread {
  public void run() {
    do_something();
  }
}
//...
public static void main(String args[]) {
  (new MyThread()).start();
}
```

Des Weiteren gibt es das **Runnable** Interface, welches ähnlich dem Thread funktioniert. Es kann von einer Klasse implementiert werden. Auch hier muss die Methode **public void run()** überschrieben werden. Jedoch muss dann eine Instanz des neuen Runnables der Thread-Klasse im Konstruktor übergeben werden.

```
class MyRunnable implements Runnable {
  public void run() {
    do_something();
  }
}
//...
public static void main(String args[]) {
  (new Thread(new MyRunnable())).start();
}
```

Lambdas sind eine weitere Möglichkeit um einen Thread zu erstellen. Hier wird dem Konstruktor von **Thread** ein Lambda übergeben und der Thread gleich gestartet.

```
public static void main(String args[]) {
  (new Thread(() -> {
    do_something();
  })).start();
}
```

Eine vierte Möglichkeit wäre eine **anonyme innere Klasse**.

Eine spezielle Art von Threads sind **Daemon Threads**. Diese Threads verhindern das Beenden des Programms NICHT. Der **Garbage Collector** ist ein solcher Thread.

```
public static void main(String args[]) {
  // ...
  someThread.setDaemon(true);
  someThread.start();
  // wait for user input
  // otherwise program will stop
  System.in.read();
}
```

Sie brechen beim JVM-Ende unkontrolliert ab

notify nur bei:
- 1 Wartebedingung
- es DARF nur einer weiter machen
RW-Lock ==>
Es gibt mehrere Warteräume, pro Bedingung

Monitor:
[synchronized] Für einfachen (ineffizienten) **gegenseitigen Ausschluss** (mutual exclusion) gibt es das **synchronized** Schlüsselwort. Es muss vor dem Typ der Funktion angegeben werden. Der Auschluss via synchronized funktioniert mittels eines **Monitor-Locks** (siehe Monitor Object [POSA2]). Es kann immer nur ein Thread eine synchronized Methode auf einem Objekt ausführen. Alle anderen müssen warten, **selbst wenn sie eine andere synchronized Methode aufrufen**. **synchronized** kann auch auf Block-level verwendet werden. Die Synchronisierung mittels **synchronized** ist **reentrant**, das heisst ein Thread kann mehrere **synchronized** Methoden verschachtelt aufrufen.

```
public class MyClass {
  public synchronized void syncMethOne(){
    dangerous_things();
  }
  public synchronized void syncMethTwo(){
    dangerous_things();
  }
  public void partlySync(){
    non_dangerous_things();
    synchronized(this) {
      dangerous_things();
    }
  }
}
```

Da jedes Objekt ein **Monitor Lock** besitzt kann auf jedem Objekt ge**synchronized** werden (also auch auf Member-Objekten). Bedingungen sollten in **synchronized** Methoden mit **while** geprüft werden, da sonst die Gefahr besteht dass man fälschlicherweise weitermacht. Warten kann man mit **wait()** und notifizieren mit **notify** und **notifyAll()** wobei **notify() irgendeinen (!!!) wartenden Thread** weckt. FAUSTREGEL: Bei unterschiedlichen Wartebedingungen mit **notifyAll()** wecken.

Vorsicht: Sobald **wait()** benutzt wird, muss die **throws InterruptedException** verwendet werden.

[Semphore] sind "Zähler". Mit **acquire()** wird eine "Marke" entfernt, mit **release()** eine zurückgelegt. **acquire()** blockiert, falls gerade keine "Marke" frei ist. **Semaphore** können fair sein (**new Semaphore(n, true)**) und folgen dan dem FIFO-Prinzip.

Semaphoren können auch negativ initialisiert werden

[Locks & Conditions] können verwendet werden um ähnliche Funktionalität wie beim **Monitor** zu erreichen, mit dem grossen Unterschied, dass hier die Wartebedingungen gezielt "notifiziert" (via **signal()**) werden können. Jedem **Lock** können mehrere **Conditions** zugeordnet sein, auf welche Threads via **await()** warten können.

```
public class MyClass {
  // fair lock
  private Lock mon = new ReentrantLock(true);
  // Conditions
  private Condition con1 = mon.newCondition();
  private Condition con2 = mon.newCondition();

  public void put(){
    mon.lock()
    try {
      while(canPut == false) {
        con1.await();
      }       // await auf Condition gibt Lock
      // do put          zwischenzeitig ab
      con2.signal();
    } finally { mon.unlock(); }
  }

  public void get(){
    mon.lock()
    try {
      while(canGet == false) {
        con2.await();
      }
      // do put
      con1.signal();
    } finally { mon.unlock(); }
  }
}
```

[Read-Write-Locks] erlauben feingranulares Sperren. Da Sperren nicht nötig ist wenn alle Threads **nur lesen**, können beliebig viele Threads gleichzeitig ein **Read-Lock** halten, jedoch **nur einer** ein **Write-Lock**. **Read-Write-Locks** können NICHT geupgraded werden.

In Java: Write-Lock nehmen falls upgradebar sein muss

[CountdownLatch] bietet einen "Einweg-Synchronisationspunkt". Bei der Initialisierung muss angegeben werden, auf wieviele Threads gewartet werden soll. Threads verwenden **countDown()** auf dem Latch um den Zähler zu dekrementieren. Mit **await()** wird gewartet bis der Latch auf 0 ist. Der Latch kann nicht wiederverwendet werden.

[CyclicBarrier] ist ähnlich wie **CountdownLatch**, kann aber wiederverwendet werden. Sie wird bei **await()** dekrementiert; **getParties()** ermittelt die Anzahl Teilnahmer der Barriere.

Sie wird nach dem Freigeben wieder neu errichtet

[Phaser] stellt eine verallgemeinerte **CyclicBarrier** dar. Mit **arriveAndAwaitAdvance()** wird auf die Freigabe gewartet. Threads können sich mit **register()** am Phaser an- und mit **arriveAndDeregister** abmelden.

[Exchanger] ermöglicht es, zwischen zwei Threads Objekte auszutauschen. **exchange(obj1)** wartet, bis der andere Thread auch **exchange(obj2)** aufgerufen hat.

[Race Conditions] werden in **low-level (Data Race)** und **high-level (Semantic Race)** unterteilt. **Low-level** Races treten auf wenn unsynchronisiert auf den gleichen Speicher (Variable, Array-Element, ...) zugegriffen wird. **High-level** Races sind Race-Conditions in der Programmlogik. Sie treten auf, wenn die Critical Sections nicht ausreichend geschützt sind.

Auf Synchronisierung kann verzichtet werden, wenn entweder **Unveränderlichkeit** (z.B Java final) gegeben ist oder veränderliche Objekte in breits synchronisierte Objekte eingesperrt sind (**Confinement**).

[Collections] aus **java.util** sind grundsätzlich **nicht threadsafe**. Es gibt jedoch threadsafe Collections in **java.util.concurrent**.

[Deadlocks] treten auf wenn sich Threads gegenseitig sperren. Folgende Bedingungen müssen eintreten damit es einen Deadlock gibt: **Geschachtelte Locks** UND **Zyklische Warteabhängigkeiten** UND **Gegenseitiger Ausschluss** UND **Kein Timeout**. Deadlocks lassen sich durch Einführen einer **linearen Sperrordnung** oder **grobgranularer Sperrung** lösen.

Live-Locks sind Deadlocks, bei denen CPU verbraucht wird

[Starvation] tritt ein wenn einem Thread immer wieder die Möglichkeit zu arbeiten "weggeschnappt" wird. Dies lässt sich durch **faire** Synchronisationsprimitiven lösen.

[Thread-Pools] besitzen eine **Task Queue** in welcher Tasks eingereiht und dann von einem freien **Worker Thread** bearbeitet werden. In Java erzeugt man **Thread Pools** mit der Factory Klasse **Executors**. Zur Auswahl stehen **newFixedThreadPool(nofThreads)**, **newChachedThreadPool()** (automatische Thread Zahl) und **newWorkStealingPool()**. Gesondert gibt es noch den **ForkJoinPool**, welcher es erlaubt, rekursive Tasks zu formulieren. **Thread Pools** haben den Vorteil, dass nicht unnötig viele Threads erzeugt werden. Einschränkung: Tasks dürfen nicht von einander abhängig sein (**Deadlock**-Gefahr!). Nur der **ForkJoinPool** setzt **Daemon Threads** ein, alle anderen Pools müssen mit **myPool.shutdown()** beendet werden. Tasks werden mittels des **Callable** Interfaces implementiert:

```
class CalcTask implements Callable<Integer> {
  @Override
  public Integer call() throws Exception {
    int val = 42;
    // long running stuff
    return val;
  }
}
ExecutorService pool =
  Executors.newFixedThreadPool(2);
Future<Integer> fut1, fut2;
// ...
fut1 = pool.submit(new CalcTask());
fut2 = pool.submit(new CalcTask());
// ...     Thread Pools: neu schreiben
int res1 = fut1.get();
int res2 = fut2.get();
// ...
pool.shutdown();
```

```
class MSTask extends RecursiveAction {
  private static final int THRESHOLD = 1;
  private int[] array;
  private int p1, p2;

  public MSTask(int[] array,int p1, int p2) {
    this.p1 = p1; this.p2 = p2;
    this.array = array;
  }

  @Override
  public void compute() {
    if (parameter > THRESHOLD) {
      int mid = (p1+p2)/2;
      MSTask t1 = new MSTask(array, p1, mid);
      MSTask t2 = new MSTask(array, mid, p2);
      invokeAll(t1, t2);
      merge(array, p1, mid, p2);
    } else {
      mergeSort(array, p1, p2);
    }
  }
}

ForkJoinPool pool = new ForkJoinPool();
pool.invoke(new MSTask(array, 0, array.length));
```

CompletableFuture statt Future

[Work Stealing] ist ein Verfahren das bei Thread Pools eingesetzt wird bei welchem es eine **globale FIFO** Queue und für jeden Worker Thread eine **locale LIFO** Queue gibt. Tasks in der globalen Queue werden (per Default) "vernachlässigt" aber der Work Stealing Pool kann auch auf FIFO umgestellt werden.

[Asynchrone Aufrufe] erlauben das Auslagern von langen Operationen auf einen anderen Thread oder Pool. Es gibt zwei Ansätze: Caller-zentrisch (**pull**) und Callee-zentrisch (**push**). Pull setzt auf **Future** Objekte während push auf **Completion Callbacks** setzt. Für Completion Callbacks eignet sich als Interface, z.B. **java.util.function.Consumer**.

```
interface Consumer<T> {
  void accept(T result);
}
// ...
void asyncOp(int in, Consumer<Integer> cb) {
  pool.submit(() -> {
    Integer res = longOperation(in);
    cb.accept(res);
  });
}
// ...
asyncOp(42, res -> {
  System.out.println(res);
});
```

[Continuations] bieten in Java 8 eine Möglichkeit, eine Folgeoperation an einen Task anzuhängen. Dazu verwendet man **CompletableFuture<T>**:

```
CompletableFuture<Long> fut =
  CompletableFuture.supplyAsync(() -> longOp());
//...
fut.thenAccept(res -> System.out.println(res));
```

Die Continuation läuft auf dem Initiator **NUR WENN** die Future das Ergebnis schon hat, sonst auf einem beliebigen Worker Thread. Mittels **allOf()** und **any()** kann eine Continuation an mehrere Futures gebunden werden:

```
// wait for ALL Futures to arrive
CompletableFuture.allOf(fut1, fut2)
  .thenAccept(cont);
// wait for ANY Future to arrive
CompletableFuture.any(fut1, fut2)
  .thenAccept(cont);
```

[.NET] Threads "funktionieren" ähnlich wie Java Threads, nur ohne Vererbung sondern mit **Delegates** und mit dem Unterschied, dass **uncaught Exceptions** per Default zum **Programmabbruch** führen. Hier ein Threading-Beispiel mit einem Lambda:

```
Thread myThread = new Thread(() => {
  for(int i = 0; i < 100; ++i) {
    Console.WriteLine("Step {0}", i);
  }
});                                    Text
// ...
myThread.Start();
myThread.Join();
```

Im Gegensatz zu Java 8 können in .NET Lambdas auch **non-final** Variablen aus dem umgebenden Scope (via Referenzen) zugegriffen werden (AUCH SCHREIBEND!!!) was die Chance auf **low-level** Races erhöht.

[Monitor] ist in .NET über **lock(someObj)** verfügbar, welches ähnlich wie **synchronized** vor einen Block gehängt wird. Best Practice: sperre ein Hilfsobjekt und nicht das eigene Objekt. Zum Warten und Benachrichten werden **Monitor.Wait(lock)**, **Monitor.Pulse()** und **Monitor.PulseAll()** verwendet. Der .NET Monitor ist fair und **Pulse()** weckt immer den Thread, der schon am längsten wartet.

```
class Account {
  // ...
  private object syncObject = new object();

  public void syncOp1() {
    //...
    lock(syncObject) {
      dangerous_things();
      Monitor.PulseAll();
    }
  }
}
```

[Andere Primitiven] in .NET sind analog zu Java, ausser den fehlenden **Locks & Conditions** und den fehlenden Fairness Flags. Dazu kommen **ReadWriteLockSlim** (upgradable), **Mutex** (binärer Semaphor) und speziellere. **Ausser** den Collections in **System.Collections.Concurrent** sind alle Collection NICHT Threadsafe.

[TPL] (Task Parallel Library) ist ein **Work Stealing Thread Pool** mit verschiedenen Abstraktionsebenen (Task Parallelism, Data Parallelism, Asynchronous Programming with Continuations). Die **TPL** erkennt geschachtelte Tasks selber und es sind keine speziellen Vorkehrungen zu treffen. Des Weiteren erzeugt die **TPL** selber neue Threads wenn sie merkt, dass alle Threads blockiert sind. ACHTUNG: **TPL** Threads sind Background Threads (analog Java Daemon Threads). Via **ThreadPool.SetMaxThreads(n)** kann die maximale Anzahl Threads festgelegt werden -> **Deadlock**-Gefahr

Hill Climbing misst Durchsatz und variiert Anzahl Worker-Threads

```
Task task1 = Task.Factory.StartNew(() => {
  // do stuff
});

// wait for task to finish
task1.Wait();
```

Task.Run() V6 S. 15/18

```
// tasks can have return values
Task<int> task2 = Task.Factory.StartNew(() => {
  int res = 42;
  return res;
});

// this blocks until task2 is finished
Console.WriteLine(task2.Result);
```

[Exceptions] in Tasks werden seit .NET 4.5 stillschweigend ingoriert und müssen explizit via den Event **TaskScheduler.UnobservedTaskException** aboniert werden.

[Datenparallelität] kann mittels Bordmitteln erreicht werden. Voraussetzung ist, dass die Unabhängigkeit der Daten gegeben ist! Nach dem Aufruf wird auf die Beendigung ALLER Tasks gewartet.

```
Parallel.Invoke(
  () => Console.WriteLine("foo"),
  () => Console.WriteLine("bar")
);

Parallel.ForEach(list,
  entry => DoStuff(entry)
);

Parallel.For(0, arr.Length,
  idx => MoreStuff(arr[idx])
);
```

[async/await] erlauben das teil-asynchrone Ausführen von Funktionen. Der Compiler zerlegt die Funktion in zwei Hälften: bis zum ersten **await** wird die Funktion vom Caller synchron ausgeführt, Der Rest wird auf einem **TPL**-Thread erledigt. **await** darf nur in Funktionen vorkommen die mit **async** gekennzeichtet sind und Funktionen die mit **async** gekennzeichnet sind MÜSSEN ein **await** enthalten! Das was nach dem await kommt ist die **Continuation**. Ist der Aufrufer ein **UI-Thread** dann wird die Continuation auf den **UI-Thread** dispatched, andernfalls auf einen **TPL-Thread**. ACHTUNG: **async/await** kann zu einem Threadwechsel INNERHALB eines Funktionsaufrufs führen.

```
class MainClass {
  public static async void DoStuff() {
    await Task.Delay(1000);
    Console.WriteLine("Knights!");
  }

  public static void Main(string[] args) {
    Console.WriteLine("foo");
    DoStuff();
    // In the meantime 'Knights!' gets
    // written to the console
    Thread.Sleep(5000);
    Console.WriteLine("bar");
  }
}
```

Beispiel für UI-Thread-Dispatching:

```
threadPool.submit( () -> {
  longOperation(); // in different thread
  SwingUtilities.invokeLater(() -> {
    // in UI thread again b/c label
    someLabel.setText("Done!");
  });
});
```

Das **[Java Memory Model]** garantiert Atomicity für **Lese-/Schreiboperationen** bis 32-Bit (mit **volatile** auch für long und double) und auf Referenzen. Visibility ist bei Locks **Release**/**Acquire** garantiert (Änderungen **vor** Release werden bei Acquire sichtbar). Bei **volatile** Variablen werden alle vorhergehenden Änderungen beim Zugriff sichtbar. **final**-Variablen werden nach dem Ende des Konstruktors sichtbar. Sichtbarkeit ist auch bei Thread-Start und -Join sowie Task-Start und -Ende garantiert.

[volatile in JAVA] garantiert, dass kein Reordering über einen Zugriff (lesend oder schreibend) auf eine derart markierte Variable hinaus statt findet. Das Ordering vor und nach **volatile** folgt innerhalb eines Threads der "As-If-Serial"-Semantik. Das heisst, der Compiler darf optimieren, falls die Semantik INNERHALB des Threads gleich bleibt. Zwischen Threads ist Ordering nur bei Zugriffen auf **volatile**-Variablen und bei Synchronisationsbefehlen garantiert. **volatile** Zugriffe führen nicht zu locking.

[Atomare Operationen] erlauben das atomare Ändern einer Variable mit einer zusätzlichen Operation (meistens return des alten Werts). Es gibt für Atomare Klassen für Boolean, Integer, Long und Referenzen (auch für Array-Elemente). Seit Java 8 gibt es auf diesen sogar Operationen welche ein "Expression-Lambda" nehmen. **Atomare Operationen** garantieren Ordering und Visibility. Beispiele für atomare Operationen sind **getAndSet(newVal)**, **getAndAdd(delta)** und **updateAndGet(lambda)**.

```
public class SLock {
  private AtomicBoolean locked =
    new AtomicBoolean(false);

  public void acquire() {
    // spin if already locked
    // else set locked to true
    while(locked.getAndSet(true)) {
      Thread.yield();
    }
  }

  public void release() {
    // set false and make visible
    locked.set(false);
  }
}
```

[compareAndSet(old, new)] erlaubt das atomare Prüfen einer Variable auf einen bestimmten Wert und ersetzt sie bei Übereinstimmung mit **new**. Rückgabewert zeigt ob die Ersetzung stattgefunden hat. ACHTUNG: Hier kann man in das **ABA-Problem** laufen.

```
do {
  oldV = var.get();
  newV = calcChanges(oldV);
} while (!var.compareAndSet(oldV, newV));
```

Das **[ABA-Problem]** beschreibt, was passiert wenn ein anderer Thread dazwischen schreibt und sich der zu vergleichende Wert scheinbar nicht ändert. Dies ist besonders bei lockfreien Datenstrukturen wie Stacks und Listen ungünstig.

[Das .NET Memory Model] unterscheided sich vom Java Memory Model darin, dass long und double auch mit **volatile** nicht atomar sind. Auch die Visibility ist nicht explizit definiert, da sie durch das Ordering gegeben ist. Beim Ordering ist es wichtig, dass **volatile** in .NET nur ein "Partial Fence" ist.

[volatile in .NET] verfolgt beim Lesen die sogenannte **Acquire-Semantik**, was bedeutet, dass alles was **VOR dem LESEN** einer volatile Variable nach "unten" optimiert werden darf. Beim Schreiben wird die **Release-Semantik** angewandt: alles was **NACH dem SCHREIBEN** kommt, darf nach oben optimiert werden. Da diese Semantik of ungenügend ist, gibt es mit **Thread.MemoryBarrier()** einen "Full-Fence".

=> besser Bilder verwenden

[GPU Vokabular]: Eine GPU besteht aus mehreren **Streaming Multiprocessors (SM)**. Jeder SM besteht aus mehreren **Streaming Processors (SP)**. In CUDA sind Threads in **Blöcke** zusammengefasst. Jeder SM kann mehrere Blöcke beherbergen, jeder Block ist intern in **Warp**s zu je 32 Threads zerlegt.

[SIMD] ist die Abkürzung für "Single Instruction Multiple Data". Dies entspricht dem Paradigma der Verktorparallelität. GPUs sind inherent für SIMD-Applikationen geeignet, denn innerhalb eines SM führen alle Cores die gleiche Instruktion auf unterschiedlichen Daten aus. Es gibt auch in CPU begrenzt mächtige SIMD-Instruktionen.

[CUDA] ist die "Computer Unified Device Architecture" von NVIDIA. CUDA arbeitet mit sogenannten **Kernels** welche auf der GPU laufen. Jeder Kernel bekommt Informationen darüber, auf welchem Block er läuft (**blockIdx.x**), welcher Thread innerhalb des Blocks er ist (**threadIdx.x**) und wie gross ein Block ist (**blockDim.x**). Auch die **y**- und **z**-Dimensionen sind nutzbar. Als Programmierer muss man die Datenaufteilung selber modellieren. **Divergenz** heisst, dass im selben Warp unterschiedliche Instruktionen vorhanden sind (z.B. via if/else). Dies bewirkt, dass je für die "if"- und "else"-Elemente ein Cycle gebraucht wird –> Performance-Probleme.

```
// vector addition kernel
__global__
void vecAKern(float *A, float *B,
              float *C, int N) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  // bounds check
  if(i < N) {
    C[i] = A[i] + B[i];
  }
}
```

Das **[CUDA Memory Management]** kennt die drei wichtigen Funktionen **cudaMalloc**, **cudaFree** und **cudaMemcpy** welche ähnlich wie ihre C-Geschwister funktionieren. Mit **cudaMalloc** kann Speicher im **Device Global Memory** alloziert werden. **cudaFree** gibt allozierten Speicher auf dem Device wieder frei und **cudaMemcpy** wird verwendet um Daten vom Host zum Device zu kopieren.

Maximale Thread-Zahl:
Gegeben: Threads per Block: 1024, Matrix size: 256*512, Anzahl SMs: 3, Max. Resident Threads/SM: 2048, Max. Resident Blocks/SM: 16
Limit A: 3 * 2048 (Max. Res. Threads/SM) **tiefste Zahl = Limit**
Limit B: 3 * 16 * 1024 (Max. Res. Blocks)
Limit C: 256 * 512 (# Threads überhaupt, wegen Matrix Size)

Das "CUDA-Grundgerüst" ohne Error Handling sieht in etwa so aus:

```
void CudaVAdd(float* A, float* B,
              float* C, int N) {
  size_t size = N * sizeof(float);
  float *d_A, *d_B, *d_C;

  cudaMalloc(&d_A, size);
  cudaMalloc(&d_B, size);
  cudaMalloc(&d_C, size);

  cudaMemcpy(d_A, A, size,
             cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, B, size,
             cudaMemcpyHostToDevice);

  // for 2/3 dimensions: dim3 gDim(32, 32);
  int bDim = 512; //block size
  int gDim = (N + bDim - 1) / bDim;
  vecAKern<<<gDim, bDim>>>(d_A, d_B, d_C, N);

  cudaMemcpy(C, d_C, size,
             cudaMemcpyDeviceToHost);

  cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
}
```

[CUDA Function Keywords] sind **__global__**(läuft auf Device wird von Host aufgerufen), **__device__**(läuft auf Device und wird von Device aufgerufen) und **__host__**(läuft auf Host und wird auch vom Host aufgerufen).

[Die Launch-Configuration] muss dynamisch bestimmt werden und sich am Problem und den Fähigkeiten des Device (ermittelt mit **cudaGetDeviceProperties()**) orientieren. Aus Effizienzgründen sollte die Blockgrösse ein Vielfaches von 32 sein (remember: 32 Threads per Warp, multiple Warps per Block). Die Blockgrösse sollte auch nicht zu gross gewählt werden, um die Anzahl der unnützen Threads zu minimieren. Die SM sollten voll ausgeschöpft werden (also ans Limit von Resident Blocks und Resident Threads gehen). Grosse Blöcke haben den Vorteil, dass die Threads interagieren können.

[Memory Access] in den **Device Global Memory** ist viel teuerer (ca. Faktor 125) als Zugriff in den **Shared Memory(__shared__)** eines SM. Um **Memory Coalescing**(das Zusammenfassen mehrerer Ladevorgänge zu einem) zu verwenden sollten Speicherzugriffe möglichst folgende Gestalt haben:

```
data[(Ausdruck ohne threadIdx.x) + threadIdx.x]
```

[Synchronisierung] in CUDA kann mit **__syncthreads()** erreicht werden, was alle Threads in einem Block zum Synchronisieren zwingt.

[Effiziente Matrix Multiplikation]:

```
__shared__ float Asub[TILE_SIZE][TILE_SIZE];
__shared__ float Bsub[TILE_SIZE][TILE_SIZE];

int tx = threadIdx.x, ty = threadIdx.y;
int col = blockIdx.x * TILE_SIZE + tx;
int row = blockIdx.y * TILE_SIZE + ty;

for (int tile = 0; tile < nofTiles; tile++) {

  Asub[ty][tx] = A[row*K + tile*TILE_SIZE + tx];
  Bsub[ty][tx] = B[(tile*TILE_SIZE + ty)*M + col];

  __syncthreads();

  for (int ksub = 0; ksub < TILE_SIZE; ksub++) {
    sum += Asub[ty][ksub] * Bsub[ksub][tx];
  }
  __syncthreads();
}
```

[Cluster] weisen spezielle Eigenschafen auf. Erstens gibt es nur **innerhalb eines Nodes** Shared Memory und nicht über die Grenzen eines Nodes hinweg. Zweitens sind Cluster durch die grosse Anzahl von General Purpose CPUs sehr gut für "allgemeine" Rechenaufgaben geeignet. Im Allgemeinen wird der Austausch von Informationen in Clustern mit **Dateien** oder über **Sockets** bewerkstelligt.

[MPI] (Message Passing Interface basiert auf dem Actor- bzw. dem CSP-Modell und ist sehr gut für heterogene Umgebungen geeignet. MPI ist ein Industriestandard einer Bibliothek für Cluster-Anwendungen. MPI ist sogenanntes **SPMD** (Single Program Multiple Data), da jeder Node das gleiche Programm mit anderen Daten ausführt. Ein Beispiel in C:

```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char* argv[]){
  MPI_Init(&argc, &argv);
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  printf("Process:%i", rank);
  MPI_Finalize();
  return 0;
}
```

Und eines in .NET:

```
using MPI;
using System;
class Program {
  public static void Main(string[] args) {
    using (new MPI.Environment(ref args)) {
      int rank = Communicator.world.Rank;
      Console.WriteLine("Process:{0}", rank);
    }
  }
}
```

[Kommunikation] in MPI kann leicht über **Communicator.world** erfolgen. Um direkt einem Node eine Nachricht zu senden, verwendet man **world.Send(value, receiverRank, messageTag)**. Analog funktioniert das Empfangen mit **world.Receive(senderRank, messageTag, out value)**. Nachrichten an alle können mit **world.Broadcast(ref value, senderRank)** abgesetzt UND empfangen werden.**Synchronisierung** kann mit **Communicator.world.Barrier()** erreicht werden. Alle Nodes warten darauf, dass alle die Barriere erreichen. Ergebnisse können mittels **Allreduce(T value, Op<T>)**(Broadcast) oder **Reduce(T value, Op<T>, int rank)**(ohne Brodcast) reduziert werden. Es existieren noch weitere Kommunikationsmöglichkeiten wie **outputArray = Alltoall(inputArray)**, **value = Scatter(array, rank)** oder **outputArray = Gather(value, rank)**.

[.NET PLINQ] via "Extension Methods":

```
salesEurope.
  Union(salesAsia).
  Union(salesAmerica).
GroupBy(item => item.Article,
        item => item.Volume).
Select(category => new {Key = category.Key,
                        Value = category.Sum()}).
Where(category => category.Value >= 1000);
```

oder mit der eingebetteten Syntax:

```
from entry in
  salesEurope.AsParallel().
  Union(salesAsia.AsParallel()).
  Union(salesAmerica.AsParallel())
group
  entry by entry.Article into category
  let sum = category.Sum(e => e.Volume)
where
  sum >= 1000
select
  new { category.Key, sum };
```

In PLINQ ist die Reihenfolge der Resultate beliebig (Ordering ist opt-in **AsOrdered()**).

[Reactive Programming mit Rx.NET] erlaubt das Reagieren auf eine aktive Quelle (**push**). Das System basiert auf dem Observer Pattern [GoF]. Um eine Pipeline zu bauen, benötigt man ein **Subject** das sowohl **Observer** als auch **Observable** ist:

```
var subject = new Subject<string>();

subject.Subscribe(Console.WriteLine);

subject.OnNext("A");
subject.OnNext("B");
subject.OnNext("C");

subject.OnCompleted();
```

Normalerweise haben Subjects kein Gedächtnis, es gibt jedoch **ReplaySubject**, **BehaviorSubject** und **AsyncSubject**.

[Software Transactional Memory (STM)] nimmt Ideen aus der Datenbankwelt und versucht damit das Problem des **Shared Mutable State** ohne Locks und Starvation anzugehen. Es gibt auch Implementationen in Hardware. Meistens wird **Optimistic Concurrency Control**(Rollback bei Konflikt) als Umsetzung verwendet. In der Java Welt gibt es ScalaSTM, ein deskriptiver Ansatz ("was", nicht "wie") und somit ein relativ einfaches Programmiermodell. Die Implementierung ist jedoch sehr komplex und "teuer".

Das **[Actor Model]** versucht das Problem zu lösen, dass herkömmliche Sprachen nicht für Nebenläufigkeit entworfen wurden. Dass Threads oft nur **Second-Class**-Features sind und Speicher per Default nicht **threadsafe** ist, macht es extrem schwierig **KORREKTE** nebenläufige Programme zu schreiben.

[Active Objects] (siehe auch [POSA2]) sind Objekte welche ein "Eigenleben" führen. Im **Actor Model** sind alle Actors **Active Objects**. **Active Objects** weist grosse Ähnlichkeiten zu CSP (Communicating Sequencial Processes) auf.

[Akkas Actor Konzept] besteht aus Actors welche nebenläufig zueinander laufen. Sie verfügen jeweils über eine **Mail-Box** um Nachrichten zu empfangen. Beim Empfangen einer Nachricht wird eine spezielle Empfangsmethode (**onReceive**) im Actor ausgeführt.

```
public class NumberPrinter extends UntypedActor {
  public void onReceive(final Object message) {
    if (message instanceof Integer) {
      System.out.print(message);
    }
  }
}
ActorSystem system =
  ActorSystem.create("System");
ActorRef printer =
  system.actorOf(Props.create(
                   NumberPrinter.class));
for (int i = 0; i < 100; i++) {
  printer.tell(i, ActorRef.noSender());
}

system.shutdown();
```

[ActorRef] speichert eine **Referenz** auf eine Instanz eines Actors. Dadurch wird verhindert, dass direkt auf Variablen und Methoden des Actors zugegriffen wird. Falls der Actor neu gestartet werden muss, **behält er seine Adresse**. **ActorRef**s können in Nachrichten verschickt werden.

[Remoting] wird dadurch vereinfacht, dass Nachrichten **immutable** sind. Ein Lookup für einen Actor kann mittels **system.actorSelection(urlString)** durchgeführt werden. Das Ergebnis (eine **ActorSelection**) kann 0-n Aktoren umfassen und zu einer **ActorRef** aufgelöst werden.

[Messaging] in Akka ist grundsätzlich asynchron. Es kann jedoch **mittels Futures** synchron auf eine Antwort gewartet werden. Messages müssen **Serializable** sein, dürfen nur **final** Felder haben und nicht über Methoden mit Seiteneffekten verfügen. Collections müssen in **Collections.unmodifiableList** verpackt werden.

```
// synchronous responses
Future<Object> result =
  Patterns.ask(actorRef, msg, timeout);
// ...
result.get();

// custom messages (Java)
public class Booking {
  final String name;

  public Booking(String name) {
    this.name = name;
  }

  public String getName() {
    return name;
  }
}

// the same (!!!) in Scala
class Booking(val name: String)
```

[Das Akka Laufzeitsystem] setzt typischerweise auf **ForkJoinPools** auf, jedoch wird aus Effizienzgründen nicht ein Thread pro Actor verwendet. **Synchrones Senden und Empfangen** von Nachrichten führt zu Warteabhängigkeiten, was wiederum zu **Deadlocks** führen kann. Deshalb wird von **synchroner Kommunikation abgeraten**.

[Akka Supervision] bezeichnet das Überwachen von Actors durch andere Actors. Eltern überwachen per Default immer ihre Kinder. Bei einer Exception wird der Supervisor informiert und muss dand entscheiden, wie es weiter geht. Der Supervisor hat die Wahl zwischen vier Möglichkeiten: mit **Resume** kann er dem Kind sagen es soll weitermachen, **Restart** startet das Kind neu, **Stop** beendet das Kind und **Escalate** (siehe Fault Tolerance Patterns) meldet seinem **eigenen Supervisor**, dass er nicht weiss, wie er reagieren soll.

Der **[System Shutdown]** erfolgt durch das Stoppen der Actors. Mittels **getContext().stop(actorRef)** wird einem Actor mitgeteilt, dass er **nach Bearbeitung der aktuellen Message** anhalten soll. Mit **actor.tell(PoisonPill.getInstance(), sender)** wird eine Terminierungsnachricht eingereiht, deren Bearbeitung den Actor stoppt. Als "last measure" dient **actor.tell(Kill.getInstance(), sender)**, was eine **Supervision Behandlung** auslöst.

Verteilung: Actor (Comm. via Messages, no shared state), Reactive
No Race Conditions: Actor (no shared mem., Comm synch'd), Reactive, STM (autom. isoliert)
No deadlocks: Reactive (Flow ist async, non-blocking, STM (kein Warten, nur Transakt.)