Parallel: mehrere Apps gleichzeitig ausgeführt. (2 cores)
Concurrent: mehrere Apps machen «gleichzeitig» Fortschritt (1 core), werden in Wahrheit aufgeteilt und sequenziell ausgeführt (Context Switch).
Thread States: running waiting ready

Multi-Thread Programmierung (Java)

```
var myThread = new Thread(() -> { … });
myThread.start(); // startet Thread & JVM called run()
class SimpleLogic implements Runnable { public void run () {…} }
var myThread = newThread(new SimpleLogic());
```

Thread Start und Ende: Richtiger Thread wird erst bei start() erzeugt. Führt run() des Runnable Interface aus. Thread endet beim Verlassen von run(). Sub-Klasse von Thread:

```
class SimpleThread extends Thread { public void run() { … } }
var myThread = new SimpleThread();//Konstruktor mit Args möglich
myThread.start(); myThread.join(); // Warten auf Thread-Ende
```

Datenstrukturen
Thread safe (keine Data Races, Lock-Free, nur atomic Operations):
- Old Java 1.0 Collections (Vector, Stack, HashTable)
- java.util.concurrent (ConcurrentHashMap, ConcurrentLinkedQueue, CopyOnWriteArrayList). Updates während Iteration sind möglicherweise nicht sichtbar.

Nicht Thread safe:
- HashSet, TreeSet, ArrayList, LinkedList, HashMap, TreeMap

```
ConcurrentLinkedQueue<Integer> q=new ConcurrentLinkedqueue<>();
q.add(1); int element1 = q.poll();
```

Thread Passivierung: Static Methoden Thread-Classes

Thread.sleep(ms): Laufender Thread geht in Wartezustand. Nach Zeitablauf wird er wieder ready
Thread.yield(): Laufender Thread gibt Prozessor frei, Wird aber direkt wieder ready. Nicht mehr nötig, da moderne Hardware preemptive ist.
Preemptive Scheduling: Thread gibt Ressourcen frei, wenn fertig

Weitere Thread Methoden

`static Thread currentThread()`: gibt aktiven Thread zurück
`void setDeamon(boolean on)`: Thread als Daemon markieren (default ist false) → in Java JVM wird nicht gewartet bis Thread Terminiert ist (fire & forget)
myThread.interrupt(): Request zum Thread von aussen stoppen, Verhalten kann vom Coder geändert werden, verwenden wenn Cancel-Policy bekannt
getId(), getName(), isAlive(), getState()

Threads in Java

Heap enthält: von mehreren Threads gesehene Shared Ressourcen & Code
Memory Cost: Jeder Thread hat seinen eigenen Stack und seine eigenen Register, Full register-backup at preemption per thread
Thread States: blocked, new, runnable, terminated, timed_waiting, waiting

Threads in DotNet

Exceptions in Threads führen zu Abbruch von App.
Keine Fairness-Flags, keine Lock & Conditions
ReaderWriterLockSlim für Upgradable Read/Write

Synchronization

Immutability: (Unveränderlichkeit): Objekte mit nur lesendem Zugriff
Confinement: (Einsperrung): Objekt gehört nur einem Thread zur Zeit synchronized: Methode / Objekt erhalten Mutual Exclusion (mutex) Lock
static synchronized: Ganze Klasse wird gelockt
Im Hintergrund wird bei synchronized ein Monitor-lock verwendet.
happens-before relationship: Am Ende von einem synchronized Block ist garantiert, dass alle Änderungen auf das Objekt für alle Threads sichtbar sind.

Kein synchronized notwendig

Object Confinement: Objekt ist nur "eingehüllt" in synchronized Struktur und braucht somit keine innere synchronized Handhabung, **Read-only Objects**

Thread Safety Java und DotNet

DotNet: Collection **nicht** ThreadSafe, ausser unter namespace System.Collections.Concurrent: *BlockingCollection<>, ConcurrentDictionary<TKey,TValue>, ConcurrentQueue<>, ConcurrentStack<>*

Monitor

Jedes Object hat ein Lock, nur 1 Thread kann es acquiren, Methoden oder Objects (mit synchronized(object) { … }) können synchronized werden
sleep() & yield() geben Lock nicht frei, stattdessen wait() verwenden
Fairness: Nicht zwingend fair, Überholproblem und keine garantierte Reihenfolge trotz synchronized (kein FIFO)
Uniform Waiters: Alle Threads warten auf die gleiche Bedingung.
One-In/One-Out: Bedingung gilt jeweils nur für einen. Nur ein einziger wartender Thread kann weitermachen
Negativ: Effizienz (viele Bedingungen bei notifyAll), keine shared Locks, unfair

Signal & Continue: Signalisierender Thread behält Monitor:
nach **Notify()/NotifyAll()** läuft er im Monitor weiter. Erst nach Ende von synchronized Block können andere Threads den Monitor lock kriegen. Aufgeweckter Thread kommt in äusseren Warteraum, muss neu um Monitor-Eintritt kämpfen
notify() gebrauchen wenn: Nur eine semantische Bedingung existert (Uniform Waiters) oder nur einzelner wartender Thread kann fortfahren (One-In/One-Out). notify() wecket Thread aus inner waiting room auf.
notifyAll() z.B. bei One-In, Multiple-Out

Spurious Wakeup: Fälschliches Aufwecken eines Threads in vereinzelten Betriebssystemen (z.B. in POSIX Thread API spezifiziert). Schlechtes Design: OS Implementierung vereinfacht, dafür Benutzung kompliziert

Waiting Rooms: Jeder Monitor hat einen eigenen einzigen outer waiting room. Pro Bedingung gibt es je einen inneren waiting room. Im outer waiting room befinden sich die Threads, die bereit zur Ausführung sind, im inner waiting room befinden sich die, die auf eine Bedingung warten.

Monitor Beispiel:

```
class BoundBuffer<T> {
  private Queue<T> queue = new LinkedList<>();
  private int limit = 1;
  public synchronized void put(T item)throws InterruptedException{
      while(queue.size() == limit) { wait(); }
      queue.add(item);
      notifyAll(); }
  public synchronized T get() throws InterruptedException {
      while(queue.size() == 0) {  wait(); }
      var item = queue.remove();
      notifyAll();
      return item; } }
```

In Dot Net Monitor: FIFO Queue, kein spurious wakeup, Signal & Continue Problem ebenfalls, pulse() / pulseAll() analog Java

```
private decimal balance;
private object syncObj = new();
public void Withdraw(decimal amount) {
  lock(syncObj) {
    while (amount > balance) { Monitor.Wait(syncObj); }
      balance -= amount;
  } }
public void Deposit(decimal amount) {
  lock(syncObj) {
    balance += amount; Monitor.PulseAll(syncObj); //notifyAll
  }}
```

Semaphore

Allgemeine Semaphore (Zähler zwischen 0 bis N)

```
new Semaphore(N)
```

- Bis zu N Threads können gleichzeitig akquiriert haben
- Für limitierte Ressourcen, Quotas, Service Throttling etc.

Binäre Semaphore (Zähler nur 0 oder 1)

```
new Semaphore(1)
```

Für mutual exclusion (1=offen, 0=geschlossen)

acquire(): Bezieht freie Ressource, wartet, wenn keine verfügbar (Zähler <= 0), sonst Zähler dekrementieren, Passeren, am besten vor `try { … }`
acquire(int permits): (Zähler -= permits) Blockiert bis Zähler >= permits ist.
release(): Gibt Ressource frei und benachrichtigt Wartende, Vrijgave
release(int permits): Zähler += permits, am besten in `finally`

```
class BoundBuffer<T> {
/* lowerLimit= # in Queue, upperLimit=verfügbare Kapazität Queue
upperLimit.counter + lowerLimit.counter = Kapazität Buffer */
  private Queue<T> queue = new LinkedList<>();
  private Semaphore upperLimit = new Semaphore(CAPACITY, true);
  private Semaphore lowerLimit = new Semaphore(0, true);
  private Semaphore mutex = new Semaphore (1, true);
  public void put(T item) throws InterruptedException {
    upperLimit.acquire();
    mutex.acquire(); queue.add(item); mutex.release();
    oder monitor: synchronized(queue) { queue.add(item); }
    lowerLimit.release(); }
  public T get() throws InterruptedException {
    lowerLimit.acquire();
    mutex.acquire(); T item = queue.remove(); mutex.release();
    oder monitor: synchronized(queue) { queue.remove(); }
    upperLimit.release();
    return item; }}
```

Faire Semaphore mit `new Semaphore(N, true);`

- Benutzt FIFO-Warteschlange für Fairness
- Langsamer als unfaire Variante

Lock & Conditions (Java)

Monitor mit mehreren Wartelisten für verschiedene Bedingungen.
Lock-Objekt: Sperre für Eintritt in Monitor, äusserer Warteraum
Condition-Obj: Wait&Signal für bestehende Bedingung, Thread bleibt im inneren Warteraum
Fairness: `new ReentrantLock(true)` → fair. default unfair

```
private Queue<T> queue = new LinkedList<>();
private Lock monitor = new ReentrantLock(true);
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 Lock

Ermöglicht gleichzeitige Lesezugriffe, da diese kein Lock brauchen, Read/Write, Write/Write, Write/Read sind nicht erlaubt

```
private Collection<String> names = new HashSet<>();
private ReadWriteLock rwLock = new ReentrantReadWriteLock(true);
// true -> fairer Lock
public boolean exists(String pat) {
  rwLock.readLock().lock(); //shared lock (andere reads erlaubt)
  try {
    return names.stream().anyMatch(n->n.matches(pat));
  } finally { rwLock.readLock().unlock(); } }
public void insert(String name) {
  rwLock.writeLock().lock(); // exclusive lock
  try { names.add(name); }
  finally { rwLock.writeLock().unlock(); } }
```

Count Down Latch

- Synchronisationsprimitive mit Count Down Zähler
- Threads können warten, bis Zähler <= 0 wird
- Threads können runterzählen
- Latches sind nur einmalig verwendbar, kein countUp() wegen Race Cond.: man wüsste nicht ob schon alle await()'s durchgekommen sind beim Aufruf

await(): warten bis Countdown = 0 ist
countDown(): Zähler um 1 dekrementieren

```
var ready = new CountDownLatch(N); // Warte auf N cars
var start = new CountDownLatch(1); // Einer gibt Signal
```

N Cars:	Race Control:
ready.countDown();	ready.await();
start.await();	start.countDown();

Cyclic Barrier – Treffpunkt für fixe Anzahl an Threads

Anzahl Threads muss im Konstruktor angegeben werden. Keine RaceControl benötigt. Reset auf ursprüngliche Anzahl Threads, nachdem Barriere hochging.

```
CyclicBarrier barrier = new CyclicBarrier(2); // 2 Biker
class Biker extends Thread {
  private final int nr; public Biker(int nr) { this.nr = nr; }
  public void run() {
    System.out.println(nr + "started");
    System.out.println(nr+"reachedCommonPoint&WaitsForOthers");
    barrier.await(); System.out.println(nr+"continueJourney");}}
var biker1 = new Biker(1); var biker2 = new Biker(2);
biker1.start(); biker2.start(); biker1.join(); biker2.join();
```

Exchanger

2 Threads warten aufeinander (wie new CyclicBarrier(2)) & tauschen Objekt aus

```
var exchanger = new Exchanger<Integer>();
for (int i = 0; i < 2; i++) { new Thread(() -> {
  for (int in = 0; in < 5; in++) {
    try { int out = exchanger.exchange(in);
      System.out.println("I got " + out);
    } catch (InterruptedException e) { … } } }).start(); }
```

Semaphore - Lock & Condition: L&C hat mehrere waiting rooms
Semaphore - CountdownLatch: semaphore: rauf und runter, blockiert bei 0; **CountdownLatch:** Einweg zu 0, blockiert bei >0 und lässt los bei 0
CountDownLatch – CyclicBarrier: CyclicBarrier is reusable, CD⊗ not

Concurrency Hazards

Data Race (spezifische Race Condition)

Nebenläufiger unsynchronisierter Zugriff mit mindestens einem **schreibenden** Zugriff (Read-Write, Write-Read, Write-Write). Variable wird gelesen, ist aber nicht korrekt (anderer Thread hat sein eigenes Resultat noch nicht in shared memory geladen). Gelöst durch `volatile` → Java garantiert, dass beim Lesen von `volatile` Variablen der aktuellste Wert vom shared memory gelesen wird.

Race Condition (Grund für Lost Update)

Timing oder Reihenfolge bestimmen Korrektheit vom Code. Passiert, wenn Operation nicht atomic ist, z.B. `count--`; anderer Thread könnte zwischen Auslesen und Schreiben von count diesen schon verändert haben. Konflikt passiert, da die **Funktion** (nicht wie bei Data Race das Problem mit local & shared memory) einen Wert ausliest und zwischenspeichert. → Lösung: `atomic`
Race Condition ohne Data Race: z.B. bei verschachtelten synchronized Funktionen. Oder: Thread A soll Thread B signalisieren, dass er starten soll. Wenn A failed zu signalisieren, wartet B für immer.

Lost Updates

Wert wird bei nicht synchronisierten Schreibzugriffen überschrieben.

Deadlock

Gegenseitiges aussperren von Threads.
Zyklus im Graph:

```
public synchronized void transfer(BankAccount to, int amount) {
    balance -= amount;
    to.deposit(amount); } // implizit geschachtelter lock
public synchronized void deposit(int amount) {
    balance += amount; }//Thread 1 a.transfer, 2 b.transfer
a.transfer(b,2); //same synchronized(a){synchronized(b){.}}
b.transfer(a,5); //same synchronized(b){synchronized(a){.}}
```

Livelock

Threads haben Sich gegenseitig permanent blockiert. Führen aber noch Warteinstruktionen aus. Verbrauchen CPU während Deadlock. (Schleifen)

Thread 1	Thread 2

b = false;
while (!a) { }
b = true;
	a = false;
	while (!b) { }
	a = true;

Lösung Lock Hierachy: Locks bekommen ein Level, man kann einen Lock nur acquiren, wenn der Lock vom Level darüber acquired ist
Lösung wenn Hierarchie keinen Sinn macht: weniger fein locken, z.B. die ganze Bank anstatt einzelner Konten locken
Fairness-Problem: 100% Fairness: alle Threads kommen gleichmässig vorwärts. 0% Fairness: nur 1 Thread läuft und andere (fast) nie. Starvation:

Starvation

Thread erhält nie die Chance, um auf eine Ressource zuzugreifen, obwohl Ressource keinem Deadlock unterliegt. Andere Threads können ihn dauernd überholen und ihm die Ressource wegschnappen.
Verhinderung: faire Synchronisationskonstrukte, faire Semaphoren, Lock & Condition, Read-Write Lock. Threads können Prioritäten zugewiesen werden, aber Scheduling ist vom OS abhängig.
Code-Beispiel mit Starvation:

```
class Switch {
  private AtomicBoolean on = new AtomicBoolean(true);
  public void toggle() {
    bool state = on.get();
    while (!on.compareAndSet(state, !state)) {} } }
```

Wenn ein anderer Thread bei toggle zuvorkommt, muss der Thread mindestens bis zum übernächsten toggle() warten. Beliebig viele Versuche bei toggle() sind zudem möglich, da andere Threads wiederholt zuvorkommen können.

Spin Lock-Deadlock

Ein Spin Lock passiert, wenn Threads in einer Schlaufe unendlich lange prüfen, ob das Lock verfügbar ist.

```
SpinLock a, b = new SpinLock();
Thread 1: a.acquire();b.acquire();b.release();a.release();
Thread 2: b.acquire();a.acquire();a.release();b.release();
```

→ Zyklisch blockiert

Spin Lock: Nicht korrekt ohne atomares Lesen & Schreiben, Fix:

```
public class SpinLock {
  private final AtomicBoolean lock = new AtomicBoolean(false);
  public void acquire() { while (lock.getAndSet(true)) { }}
  public void release() { lock.set(false); }}
```

Thread Pool

ForkJoinPool

Aufruf über ForkJoinPool (`invoke` und `invokeAll` sind blocking):

```
var threadPool = new ForkJoinPool();
int result = threadPool.invoke(new CountTask(2, N));
```

auch möglich: `threadPool.invokeAll(array);`
oder mit `.commonPool` (je nach Systemconfig nicht alle CPU gebraucht!)

```
int result = new CountTask(2, N).invoke();
```

Java Future

```
var threadPool = new ForkJoinPool();
Future<Integer> future = threadPool.submit(() -> {
  int value = ...; /* long calculation */ return value; });
```

get() wird nie aufgerufen → fire & forget, get() blockiert bis Task beendet ist:

```
T result = future.get(); Oder canceln probieren: future.cancel()
```

Work Stealing

Work Stealing ist eine Strategie, bei der untätige Threads in einem Thread-Pool Aufgaben aus den Warteschlangen der beschäftigten Threads "stehlen". Somit wird die idle time minimiert.

Daemon Workers

Normalerweise wartet JVM auf Threads bevor sie terminiert. Wenn Thread ein Daemon ist, bricht JVM diese Threads unkontrolliert ab, sobald der main Thread endet. Daemon Effekt kaputt mit `.join()` bei jedem Thread.

```
Thread daemonThread = new Thread();
daemonThread.setDaemon(true);
```

Java Thread Pool manuelle Implementation

```
public class ThreadPool {
  private BlockingQueue<Runnable> queue; //custom implementation
  private int size;
  public ThreadPool(int size) {
    if (size < 1) {
      throw new IllegalArgumentException("Must be > 0"); }
    this.size = size;
    queue = new BlockingQueue<>(size);
    for (int i = 0; i < size; i++) {
      Thread thread = new Thread(() -> {
        while (true) {
          try {
            Runnable task = queue.remove();
            task.run();
          } catch (InterruptedException e) {
            break; // Thread interrupted, exit the loop } } } );
      thread.start(); } }
  public void submit(Runnable task) { queue.add(task); }
  public void shutdown() {
    // Interrupt worker threads damit Programm terminiert
    // Optimal: zuerst prüfen, ob Threads idle sind
    for (Thread thread : Thread.getAllStackTraces().keySet()) {
      if (thread.getState() != Thread.State.TERMINATED) {
        thread.interrupt();} } } }
//Auch möglich um Tasks abzuwarten
ForkJoinPool.commonPool().awaitTermination(10,TimeUnit.SECONDS);
```

Rekursive Tasks Java (in ForkJoinPool)

```
class CountTask extends RecursiveTask<Integer> {
  public CountTask(int lower, int upper) {
    this.lower = lower; this.upper = upper; }
  protected Integer compute() {
    if (lower == upper) { return 0; }
    if (lower + 1 == upper) { return isPrime(lower) ? 1 : 0; }
    int middle = (lower + upper) / 2;
    var left = new CountTask(lower, middle);
    var right = new CountTask(middle, upper);
    left.fork(); right.fork();
    return right.join() + left.join(); } }
var threadPool = new ForkJoinPool();
int result = threadPool.invoke(new CountTask(2, N));
Oder: int result = new CountTask(2, N).invoke();
```

```
class SignumTask extends RecursiveAction {
  private final int[] array;
  private final int left, right;
  private final static int THRESHOLD = 1; // appropriate value
  // Constructor, assign private fields
  protected void compute(){
    if(right - left > THRESHOLD) {
      // Call recursive Tasks (Divide & Conquer):
      int middle = (left+right)/2;
      invokeAll(
        new SignumTask(array, left, middle),
        new SignumTask(array, middle, right)
      ); // kann auch Liste (Collection) entgegennehmen
    } else { // Sequential Code:
      for(int i = left, i < right; i++){
        array[i] = signum(array[i]); } } } }
// Aufruf für so viele Threads wie array lang ist (für T=1):
new SignumTask(array, 0, array.length).invoke();
```

Keine Überparallelisierung

```
if (upper - lower > THRESHOLD) {…} else { sequential code }
```

Anzahl Tasks - Keine Überparallelisierung
Java: für THRESHOLD=20 und array mit Länge 100:
100→(50,50)→(25,25,25,25)→(12,13,12,13,12,13,12,13) => 15 Tasks
➡ Abhängig von Array-Länge und THRESHOLD
.Net: Anzahl (freier) logischer Prozessoren (Range Partitioning)
➡ Loop partitioning abhängig von Anzahl freier Worker Threads

C# .Net Task Parallel Libray (TPL)

```
// Nicht parallelisierbar, wenn auf gleiches Element in Loop +=
// ausgeführt wird → Data Race
Parallel.For(0, array.length, (row) => {
  Parallel.For(0, array[0].length, (col) => {
    array[row][col] = (row + 1) * (col + 1); }); });
// Inkrementation um 2:
Parallel.For(0, array.Length / 2, idx => {
  int i = 2 * idx;
}); // oder
Parallel.For(0, array.Length, i => SomeTask(i));
Parallel.ForEach(list, file => Convert(file));
```

```
ThreadPool.SetMaxThreads()
// Task starten / abwarten
Task<int> task = Task.Run(() => {...return total;});
task.Wait(); // blockiert (falls keine Rückgabe)
task.Result; // blockiert und liefert Resultat
// kann parallel ausgeführt werden, implizite Barrier am Ende
Parallel.Invoke(() => res1 = MergeSort(1, m),
  () => res2 = MergeSort(m,r));
// PLINQ: default Reihenfolge wird nicht wegen .AsParallel()
beibehalten, deshalb .AsOrdered(), um Reihenfolge zu behalten
inputList.AsParallel().AsOrdered().Select(x =>
IsPrime(x)).ToList();
```

Asynchronität

Java – CompletableFuture

Pro parallele Action muss ein CompletableFuture eingesetzt werden.
Main Thread wartet nicht auf CompletableFutures die noch am Laufen sind.
Async Threads laufen im ForkJoinPool.commonPool()

Führt asynchrone Action aus, liefert Rückgabe: `CompletableFuture<String> f = CompletableFuture.supplyAsync(() -> "Hello");`
`f.runAsync(() -> doIt())` führt asynchrone Action aus, ohne Rückgabe.

```
f.thenApply[Async](s->s+" World");//get() liefert "Hello World"
f.thenAccept[Async](s->print(s + " World"));//get() liefert void
```

Damit Fehler nicht verloren gehen: `.join()` oder `future.exceptionally(e -> { return <<default string>>; });`

For Each Async – um mehrere CompletableFutures abzuwarten:

```
CompletableFuture<Void> analyzeAsync(String website) {
  List<CompletableFuture<Void>> futures = new ArrayList<>();
  for(String link : extractLinks(website)) {
    futures.add(CompletableFuture.runAsync(() -> {
      // check link
    }); }
  return CompletableFuture.allOf(futures.toArray(new //or .any()
CompletableFuture[0]));
//real array now }} Aufruf (blocking): analyzeAsync(...).get();
```

DotNet

Caller-centric (Caller wartet auf Task-Ende): `int result = task.Result;`
Callee-centric: (Task übergibt Resultat dem Nachfolger, Task Continuation): `Task.Run(t1).WhenAll/WhenAny(t1, t2).ContinueWith(t3).Wait()`

Aufrufer von async-Methode ist nicht zwingend während gesamter Ausführung blockiert. **Synchrone** Ausführung bis zu `await` von aufrufendem Thread, danach wird `async` Methode parallel ausgeführt, die aufrufende Methode wird suspended bis der awaited Task completed ist, Code nach await = Continuation
Rückgabetypen: void, Task (erlaubt Warten auf Ende), Task<T>
Compiler: await nur in async (Error), async muss await enthalten (Warnung).

GUI Thread Modell

Non-Blocking UI (.NET)
Caller ist normaler Thread: TPL-Worker-Thread führt Continuation aus
Caller ist UI-Thread: UI-Thread führt Continuation als Event aus

```
var url = textField.Text;
Task.Run(async ()=> {
  var text = await DoItAsync(url);
  Dispatcher.InvokeAsync(() => {label.Content = text}) });
// oder mit async/await (Namenskonvention in Violett):
async Task<Void> DoItAsync() {
return await DownloadAsync(url); }
```

Non-Blocking UI (Java)

```
button.addActionListener(event -> {
    var url = textField.getText();
    CompletableFuture.runAsync(() -> {
        var text = download(url);
        SwingUtilities.invokeLater(() -> {
            textArea.setText(text); }); }); );
```

Rekursiver Aufruf (um Reihenfolge zu behalten):

```
button.addActionListener(event -> log(list));
void log(List<String> pending) {
    if (pending.size() == 0) {
        statusLabel.setText("done");
    } else {
        String message = pending.remove(0);
        ForkJoinPool.commonPool().submit(() -> {
            logToServer(message);
            SwingUtilities.invokeLater(() -> {
                statusLabel.setText(message + " logged");
                log(pending);  }); }); } }
```

UI Thread: blau, Worker Thread: rot
Wartender UI Aufruf: `SwingUtilities.invokeAndWait(() -> ...);`
invokeAndWait(): der aufrufende Thread wird blockiert
invokeLater(): der aufrufende Thread läuft parallel weiter

Memory Model

Lock-Freie Programmierung

Korrekte nebenläufige Interaktionen ohne Locks
Garantien des Speichermodells nutzen. Ziel: Effiziente Synchronisation

Probleme (Java Weak Memory Model)

Weak Consistency: Instruktionen werden in verschiedenen Reihenfolgen von verschiedenen Threads ausgeführt. Ausnahme: Synchronisationen/Speicherbarrieren (Memory Fences)
Grund: Optimierungen von Compiler, Laufzeitsystem und CPUs. Instruktionen werden umgeordnet/wegoptimiert

Java Memory Model: Minimale spezifizierte Garantien:

Atomicity (Unteilbarkeit)
Separates Lesen / Schreiben ist atomar für primitive Datentypen bis 32 Bit, Objekt-Referenzen, long und bei double nur mit `volatile` Keyword atomar
Visibility (Sichtbarkeit)
Locks Release & Acquire: Änderungen vor Release sind bei acquire sichtbar
`volatile`: Variable wird im main memory anstatt lokal beim Thread gespeichert. Beim `volatile` write ist garantiert, dass alles Vorherige zum Lesen sichtbar ist. Beim `volatile` read ist garantiert, dass alle vorherigen Writes sichtbar sind. `volatile` Lese- und Schreibzugriffe werden bei Java nicht umgeordnet (starke Ordnung). Race Conditions können immer noch entstehen wenn nicht atomar. Keine Data Races mehr.
`atomic`: Operationen werden nicht von anderen Threads unterbrochen
final Variablen: Nach Ende von Konstruktor sichtbar
Thread Start/Ende: Start happen-before alle Thread-Actions, alle Thread-Actions happen-before join()-Return in irgendeinem anderen Thread

AtomicInteger

```
private AtomicInteger count = new AtomicInteger();
addAndGet(n); // Addiert n und gibt neuen Wert zurück
getAndAdd(n); // Gibt alten Wert zurück und addiert n
getAndIncrement(); // Gibt alten Wert zurück und inkrementiert
incrementAndGet(); // Inkrementiert und gibt neuen Wert zurück
```

Atomares compareAndSet

- Setzt `update`, wenn Wert gleich `expected` ist (atomar)
- Retourniert true bei erfolgreichem Update (also wenn AtomicInteger den Wert von `expected` enthielt)

```
boolean compareAndSet(int expected, int update)
```

AtomicBoolean

```
AtomicBoolean atomicBoolean = new AtomicBoolean(true);
boolean oldValue = atomicBoolean.getAndSet(false);
boolean wasExpected = atomicBoolean.compareAndSet(false, true);
```

Ordering (Reihenfolge)

Reihenfolge bei Sichtbarkeit, Partielle Ordnung: Unlock garantiert vor Lock, volatile Write garantiert vor volatile Read
Zusätzlich, Totale Ordnung: Operationen im `synchronized` Block werden nicht reordered. Lock/Unlock, volatile-Zugriffe, Thread-Start/Join

Volatile in Java

Livelock Beispiel fix: Keine Umordnungen in Java, wegen volatile:

```
volatile boolean a = false, b = false;
```

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

Dot Net Memory Model

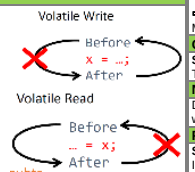
Unterschiede zu Java: kein **Atomicity** für long/double durch volatile (Atomare Instruktionen durch **Interlocked** class (sowie double/long)). **Visibility** nicht definiert, aber implizit durch Ordering. **Ordering:** nur Half und Full Fences.

Half Fence: Zugriffe vor `volatile` **write** bleiben davor.

Zugriffe nach `volatile` **read** bleiben danach.

Gar keine Umordnung über **Memory (Full) Fence:** `Thread.MemoryBarrier();` → in Kombination mit `volatile` Variablen!

(.NET) → bei write *danach* und bei read *davorsetzen*.



Unterschied volatile Java VS Dotnet

Atomarität: In Java sind volatile double/long atomar, in .NET nicht.

Sichtbarkeit: In .NET nicht definiert. **Ordnung:** volatile/volatile sind in Java stark geordnet, in .NET sind die in eine Richtung umordenbar. In .NET gibt es eine definierte partielle Ordnung bei volatile Zugriffen in jedem Fall; bei Java ist dies nur bei einer Schreiben-Lesen Beziehung zwischen Threads.

Performance Scaling

GPU: High Throughput, CPU: Low Latency

Performance

Moore's Law: alle zwei Jahre die doppelte Menge an Transistoren.
Atomgrenzen fordern neue Wege -> Parallelisierung mit mehr Cores

Performance ist definiert durch Speicherbandbreite, Rechenbandbreite und Latenzzeit=latency (Gesamtzeit = Speicherzugriff + Rechenzeit)

Compute bound: nicht genügend Prozessorleistung. Prozessor ist bottleneck. Memory ist kein bottleneck vom Setting her 😉.

Memory bound: mehr Zeit für den Speicherzugriff im Vergleich zu Berechnungen.

Conclusion: Software, die compute-bound ist, wird bevorzugt, da es möglich ist, diese Zeit zu verkürzen, indem die Berechnungen optimiert werden. Dies funktioniert nicht für memory-bound Software.

Arithmetic Intensity

Arithmetische Intensität ($FLOPs/Byte$): z.B. 32Bit Int's
Code: $z[i] = x[i] + y[i]$ → 4Byte + 4Byte + 4Byte und ein "+" = 1 FLOP
= 1/12 FLOPs/Byte, Höher ist besser für eine effiziente parallele Nutzung.

Throughput (operations/sec) beschreibt, wie viele Instruktionen / Operationen in bestimmter Zeit gemacht werden können (Frequenz).

Latency Zeit, welche für eine einzelne Instruktion / Operation von Anfang bis Ende benötigt wird (Einheit: sec).

T Zeit für eine Instruktion ist: $t_{\text{Memory Access}} + t_{\text{Computation}}$

Compute Bound bedeutet, dass $t_{\text{Memory Access}} < t_{\text{Computation}}$ gilt.

Memory Bound bedeutet, dass $t_{\text{Memory Access}} > t_{\text{Computation}}$ gilt.

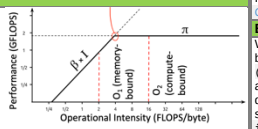
Wir können dies auch mit der arithmetischen Intensität ausdrücken:

AI: $\frac{\text{operations}}{\text{bytes}} = \frac{\text{Bandwidth}_{\text{Computation}}}{\text{Bandwidth}_{\text{Memory}}}$, compute bound: $\frac{\#\text{ops}}{\#\text{bytes}} > \frac{\text{BW}_{\text{compute}}}{\text{BW}_{\text{memory}}}$

Je **höher** arithmetischen Intensität ist, desto **besser** können wir moderne Parallelisierung durch CPU / GPU ausnutzen.

Roofline Model

Zieht locality, bandwidth, parallelization paradigms, multicore, etc. in Betracht und zeigt ob memory- oder compute-bound
Nach dem Ridge Point: Leistung ist compute-bound (minimale Arithmetic Intencity (AI), um beste Leistung zu erreichen).



FLOPS=Floating Point Operations per Second
β=Bandwidth, I=Arithmetic Intensity, π=Peak performance

⇨ Ridge point is at operational, i.e. Arithmetic Intensity: $I = \frac{\pi}{\beta}$

⇨ Attainable Performance P with given β and I: $P = \min(\pi, \beta * I)$ (flops/s)

⇨ Higher Performance: $\frac{\pi_{faster}}{\beta_{slower}}$=AI values how long the faster processor stays faster (intersection point)

GPUs are useful for: Matrix & Vektor, Rendering, Hash-Cracking

Weak Scaling (Gustafson's Law)

Die Anzahl Prozessoren und die Problemgrösse werden gleichzeitig skaliert → perfect scaling: gleiche workload pro Prozessor.

Speedup: $s + p * N = s + (1 - s) * N$

Example: We have 64 Processors, 5% of the program is serial.
scaled weak speedup $= 0.05 + (1 - 0.05) * 64 = 60.85$

Strong Scaling (Amdahl's Law)

Die Anzahl Prozessoren wird erhöht, während das Problem gleich bleibt → weniger Arbeit pro Prozessor.

T = total time ($T = (1 - p)\,T + Tp$)
p = part of the program can be parallelized, s = part serial, N = # of Cores

$$T_N = \frac{T * P}{N} + (1 - P) * T$$

This law ignores the parallel overhead such as task startup time, interprocess interaction, idling due load imbalance / synch, etc.

Speedup: $\frac{T}{\frac{T*p}{N}+(1-p)*T} = \frac{1}{1-p+\frac{p}{N}} = \frac{1}{s+\frac{p}{N}}$, $s = (1 - p)$

⇨ Max speedup: $N = \infty$ *infinity*

Efficiency: $\frac{T}{N*T_N}$

Example: 90% can be parallel using 8 processors:
$p = 0.9, s = 0.1, N = 8$ → $Speedup = \frac{1}{0.1+\frac{0.9}{8}} = 4.7$

Berechnungen & Formeln

Die **Maximale Thread Anzahl** lässt sich aus den Informationen vom Device Query (`cudaGetDeviceProperties()`) berechnen:

```
Device 0: "NVIDIA RTX A4000"
  Multiprocessor count                 48
  Maximum threads per multiprocessor   1536
  Maximum threads per block            1024
```

Woraus wir nun ableiten können: $\text{Limit}_{\text{Total}} = \min(L_A, L_B) = XXX$ (siehe unten)
⇨ 200x900pixels → 1 Thread pro Pixel: 180'000 threads
⇨ **Block Dimension** (quadratisch): 32×32 → (1 Block = 1024 = 32×32)

⇨ **Grid Dimension:** $Ceil\left(\frac{200}{32}\right) \times Ceil\left(\frac{900}{32}\right) = 7 \times 29$

⇨ **idle threads (=zu viel gestartet):** $7 * 29 * 1024 - (200 * 900 pixels) = 27'872$
Max Threads= $L_A = \text{Num}_{SM} * \text{Max}_{\text{ThreadsPerBlock}} = 48 * 1024 = 49'152$

GPU Structure

SM: **S**treaming **M**ultiprocessors beinhalten Streaming Processors(=1 GPU core)
Thread wird auf Core, Thread Block auf 1 SM & Grid auf GPU ausgeführt

NUMA (Non-Uniform Memory Access)

Daten müssen von Host (CPU) nach Device (GPU) übertragen werden und wieder zurück nach Berechnung.

Flynn's Classical Taxonomy (SISD, SIMD, MISD, MIMD)

SISD: Single Instruction Single Data. Sequenzielle Ausführung von Instruktionen
SIMD: Single Instruction Multiple Data (auf SM ausgeführt)
MISD: Multiple Instruction Single Data. Mehrere Nodes führen verschiedene Operationen auf den gleichen Daten aus. **MIMD (perfect)**: Multiple Instruction Multiple Data. Tasks, die von verschiedenen Prozessoren ausgeführt werden, können zu unterschiedlichen Zeiten beginnen oder enden.

CUDA (Computer Unified Device Architecture) (GPU)

Device Query: Infos über GPU: Max N of Threads per Block, Max Block Dim, Max Grid Size, Bus Width, Warp Size

Whole block runs on 1 SM. All threads in a block must complete.

The more SM in a GPU, the more blocks can be executed in parallel.

The SM also perform **Context Switching**.

Jeder Thread hat ein **private memory**, jeder block hat ein **shared memory** und ein grid hat ein **global memory**.

Für Variablen- und Methodendeklarationen:

- `__global__`: Läuft auf Device wird vom Host aufgerufen
- `__device__`: Läuft auf Device und wird vom Device aufgerufen (global)
- `__device__ __shared__`: device aber nur im Block shared
- `__device__ __const__`: global constant
- `__host__`: Läuft auf Host und wird auch vom Host aufgerufen
- `__shared__`: Memory zwischen Threads in Block

Memory Access in den device Global Memory dauert viel länger als in den Shared Memory (`__shared__`) eines SM (ca. Faktor 125). Aus diesem Grund ist das Shared Memory oftmals vorzuziehen.

Memory Coalescing Eine Coalesced Memory Transaction findet statt, wenn alle Threads in einem Warp nachfolgende Speicherstellen aus dem global memory laden. Denn dann fasst die Hardware die langsamen Zugriffe auf das global memory zusammen (burst) und ist somit schneller. Sieht oftmals so aus:

```
data[blockDim.x * blockIdx.x + threadIdx.x]
```

Synchronisierung in CUDA kann mit `__syncthreads()` erreicht werden, was **alle Threads in einem Block** zum Synchronisieren zwingt. Kann nur in einem `if-else` Block verwendet werden, wenn **alle** Threads die gleiche Abzweigung nehmen (ansonsten Fehlverhalten).

Warps: Hardware gruppiert Threads, die dieselbe Anweisung ausführen, in Warps. Ein Warp sind 32 Threads innerhalb eines Thread-Blocks (SIMD).

Data Partitioning

threadIdx.x/y/z: Thread index innerhalb Block
blockIdx.x/y/z: Block Index innerhalb Grid
blockDim.x/y/z: Block size, **gridDim.x/y/z**: Grid size

Für Matrizen braucht man `dim3`, Block size für 1024 = dim3 (32, 32)

Indexierung in memory:

```
C[i * M + j] = sum; // equivalent to C[i, j]
```

Boundary Checks

Wenn wir den boilerplate Code brauchen, um die aufgerundete Gridsize zu bestimmen, dann erstellen wir
`(gridSize.x*gridSize.y*blockSize.x*blockSize.y) % N` mehr Threads, als wir eigentlich wollen. Somit könnten andere Threads beeinflusst werden. Um das zu beheben, muss die Grösse des Problems (z.B. Arraygrösse) bekannt sein und nach einem Overflow geprüft werden:

```
if (i < N) {  C[i] = A[i] + B[i];  }
```

Thread Divergence

Happens when threads of the same warp take different branches in the code. E.g., for the signum task we have 3 branches (x>0,x<0,x=0) – if elements of a warp have different signs, different branches are being taken. Then, threads of different branches have to wait because alternating the instructions of different branches are being executed. For 3 branches the worst case is 3 times slower.

Number of Threads

```
VectorAddKernel<<<dim3(8,4,2), dim3(16, 16)>>>(d_A, d_B, d_C);
```
Startet $8 * 4 * 2 * 16 * 16 = 16'384$ Threads

```
VectorAddKernel<<<4, 1024>>>(A, B, C, 4096);
```
Startet Grid mit 4 Blocks, welche je 1024 Threads haben, $4 \cdot 1024 = 4096$ Threads, **1D**: 4 = dim3(4, 1, 1), **2D**: dim3(4,2) = dim3(4, 2, 1), **3D**: dim3(4, 2, 2)

Error Handling

```
void handleCudaError(cudaError error) {
  if (error != cudaSuccess) {
    fprintf(stderr, "CUDA: %s! \n", cudaGetErrorString(error));
    exit(EXIT_FAILURE); } }
handleCudaError(cudaMemcpy(...)); // Bei normalen CUDA Befehlen
SomeKernel<<<...>>>(...);
handleCudaError(cudaGetLastError()); // Nach Kernel Operation
```

Unified Memory

Automatischer Speicher Transfer vom Hauptspeicher zum GPU, neue Regeln:

```
cudaMallocManaged(&A, size);
SomeKernel<<<...>>>(A);
cudaDeviceSynchronize(); cudaFree(A);
```

Code

PairwiseSum

```
// gridSize = 1024; blockSize=(len/2+blockSize-1)/blockSize;
// Uses strided coalescing
__global__ void pairwiseSum(int* array, int length){
    int i = 2 * (blockIdx.x * blockDim.x + threadIdx.x);
    if (i < length - 1) {
        array[i] += array[i + 1];
        array[i + 1] = 0;  } }
```

VectorAddition Code

```
__global__ void vecAdd(int *a, int *b, int *c, int n) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  // Make sure we do not go out of bounds (boundary check):
  if (idx < n) { c[idx] = a[idx] + b[idx]; }
}
int main(int argc, char * argv[]) { // CPU (Host), N=1000000
  int blockSize = 1024; // max
  int gridSize = (N + blockSize - 1) / blockSize);
  size_t size = N * sizeof(double);
  *d_a, *d_b, *d_c; // *h_b, *d_b, ...
  cudaMalloc(&d_a, size); // same for d_b and d_c
  cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
  vecAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);
  cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
  cudaFree(d_a); //same for d_b and d_c
  return 0;  }
```

Multi Kernel Grid – Matrix Multiplication

```
__global__
void matrixMultKernel(float *A, float *B, float *C) {
  __shared__ float Asub[TILE_SIZE][TILE_SIZE]; // shared memory,
  __shared__ float Bsub[TILE_SIZE][TILE_SIZE]; // i.e. "cache"
  int tx = threadIdx.x, ty = threadIdx.y;
  int col = blockIdx.x * TILE_SIZE + tx;
  int row = blockIdx.y * TILE_SIZE + ty;
  int nofTiles = (A_COLS + TILE_SIZE - 1) / TILE_SIZE;
  float sum = 0.0;
  for (int t = 0; t < nofTiles; t++) {
    if (row < A_ROWS && t * TILE_SIZE + tx < A_COLS) {
      int Aidx = row * A_COLS + t * TILE_SIZE + tx;
      Asub[ty][tx] = A[Aidx]; }
    if (col < B_COLS && t * TILE_SIZE + ty < B_ROWS) {
      int Bidx = (t * TILE_SIZE + ty) * B_COLS + col;
      Bsub[ty][tx] = B[Bidx];  }
    __syncthreads(); //matrices are done before calculating sum
    if (row < C_ROWS && col < C_COLS) {
      for (int k = 0; k < TILE_SIZE; k++) {
        if (t * TILE_SIZE + k < A_COLS) {
          sum += Asub[ty][k] * Bsub[k][tx]; } } }
    __syncthreads(); //calculations are done before writing to C
    if (row < C_ROWS && col < C_COLS) {
      C[row * C_COLS + col] = sum; // 2D → 1D } } }
// Aufruf -> Resultate nach A, B, C
dim3 threadsPerBlock(TILE_SIZE, TILE_SIZE);
dim3 blocksPerGrid((C_COLS + TILE_SIZE - 1) / TILE_SIZE, (C_ROWS
+ TILE_SIZE - 1) / TILE_SIZE); // this ceils the # of blocks
matrixMultKernel<<<blocksPerGrid, threadsPerBlock>>>(A, B, C);
// Tile size: should fit in the GPU's shared memory or cache
```

Transpone Matrix

```
__global__
void transpose(int* matrix, int nRows, int nCols, int* result){
  int row = blockIdx.x * blockDim.x + threadIdx.x;
  int col = blockIdx.y * blockDim.y + threadIdx.y;
  if (row < nRows && col < nCols) {
    // result[col][row] = matrix[row][col]:
    result[col * nRows + row] = matrix[row * nCols + col]; } }
// Aufruf
dim3 blockSize(32, 32);
dim3 gridSize((N + 31) / 32, (M + 31) / 32);
transpose<<<gridSize,blockSize>>>(d_matrix, N, M, d_result);
// Lesen ist hier langsamer, da kein coalescing.
// Optimierung mit Shared Memory (Tiled Transposing): Jeweils
// ein Tile mit Coalescing vom Device Memory ins Shared Memory
// einlesen. __syncThreads(). Danach vom Shared Memory das
// transponierte Element mit Coalescing zurück in Device Memory
// schreiben.
```

Cluster-Parallelisierung (multiple computers, i.e. nodes)

Parallel Computer Memory Architectures

No matter how fast the processing unit is, if the memory cannot maintain & provide instructions & data at a sufficient speed, there will be no improvement in performance.

SPMD: Single program multiple data, nutzt message passing / hybrid coding
MPMD: multiple program, multiple data
Shared Memory (Achtung: Race Conditions möglich)
Shared memory verringert die Zugriffszeit für Speicherzugriffe. Ausserdem ist shared memory geteilt zwischen allen Threads eines Blocks.
Uniform Memory Access (UMA)
All processors in the UMA model have a shared memory. Bottleneck because only one CPU can access Memory at a time.
Non-Uniform Memory Access (NUMA)
Under NUMA, a processor can access its own local memory faster than non-local memory (memory local to another processor or memory shared between processors). Bottleneck improved because every CPU has its own memory.
Distributed Memory
Distributed memory systems require a communication network to connect inter-processor memory. Developer has to define explicitly how and when data is communicated. No global address space across all processors. Send via MPI.
Hybrid Distributed-Shared Memory
All processors in a machine can share the memory and can request data from other computers

Clusters

A **High-Performance Cluster (HPC)** uses a hybrid memory model and can run a program on multiple nodes. A node is a standalone computer. Nodes are networked together to form a supercomputer.
- No Shared Memory (= NUMA) between Nodes
- Shared Memory for Cores inside a Node (Symmetric Multiprocessing)

Message Passing Interface (MPI)

Based on Actor/CSP principle. Difference: MPI starts all processes on start. Similarities: isolated memory (NUMA) and async. inter-node communication. When more than one processor wants to exchange data, then they need to send messages and communicate with each other. All variables inside a process are private. All processes start and terminate synchronously. Process works in their (virtually) isolated address space. No shared memory between nodes (NUMA). But there is shared memory for all cores inside a node. Synchronization with barriers possible.

Communication Modes

Point to point: One sender, one receiver. Relies on matching send/receive.
Collective communications:
Communication between groups of processes can be desired.
Broadcast:
MPI_Scatter: Broadcast, aber Prozesse erhalten jeweils anderes Array-Element
MPI_Gather: Reduce, Root concatenated empfangene Daten aber in Array

MPI Functions

MPI_Comm_rank(MPI_Comm communicator, int* rank): Each process inside of a communicator is assigned an incremental rank starting from zero. Is used for identification purpose.
MPI_Comm_size(MPI_Comm communicator, int* size): returns the size of a communicator (number of processes in group).
MPI_COMM_WORLD (constructed for us by MPI) encloses all of the processes in the job, so this returns the total number of processes that were requested for the job.
MPI_Send(array, LENGTH, MPI_INT, receiverRank, tag, MPI_COMM_WORLD); // based on `int array[LENGTH];`
MPI_Recv(array, LENGTH, MPI_INT, senderRank, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
MPI_Barrier(MPI_COMM_WORLD): Warte, dass alle Prozesse Barriere erreichen
MPI_Reduce(&value, &total, 1, MPI_INT, MPI_SUM, receiverRank, MPI_COMM_WORLD) Nur ein Prozess (receiverRank) sieht das Gesamtresultat. Effizienter als Allreduce, weil kein Broadcast. Rank 0 schickt ebenfalls ein Resultat auch wenn er der Receiver ist! Alle Prozesse müssen das MPI_Reduce ausführen (ausser bei receiverRank ist die Ausführung optional), sonst terminiert das Programm nicht.
MPI_Allreduce(&value, &total, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD): Aggregation von Teilresultate zwischen Prozessen. Jeder erhält das Gesamtresultat als Rückgabe-wert. Implizite Barriere/Broadcast über alle Prozesse.

MPI Reduce (in detail)

```
MPI_Reduce(void* send_data, void* recv_data, int count,
MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
```

MPI Operations: MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD (product, Multiplies all elements), MPI_LAND (logical ***and***), MPI_LOR (logical ***or***), MPI_BAND (bitwise ***and***), MPI_BOR (bitwise ***or***), MPI_MAXLOC (maximum value + rank of process), MPI_MINLOC (min value + rank of process)
Datentypen: MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG, MPI_LONG_LONG, MPI_UNSIGNED, MPI_FLOAT, MPI_DOUBLE

The communicator is a group of MPI processes and allows communication between those. Own groups definable. Different ranks and sizes per group.

Code Example: Monte Carlo Pi Simulation

Randomisierte Berechnung von Pi. Generiere zufällige Punkte aus Fläche. Schaue, ob sie im Einheitskreis liegen. Trefferrate gibt eine Annäherung an Pi.

Sequentiell:

```
long count_hits(long trials) {
  long hits = 0, i;
  for (i = 0; i < trials; i++) {
    double x = (double)rand()/RAND_MAX;
    double y = (double)rand()/RAND_MAX;
    if (x * x + y * y <= 1) { hits++; } } return hits; }
```

MPI:
Ausführen: `mpiexec -n <<size>> <<file path>> [params]`

```
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
srand(rank * 4711); //every process has random seed
// every process calculates a fraction:
long hits = count_hits(TRIALS / size);
long total;
MPI_Reduce(&hits, &total, 1, MPI_LONG, MPI_SUM, 0,
MPI_COMM_WORLD); // process 0 receives result
if (rank == 0) {
  double pi = 4 * ((double)total / TRIALS); }
```

Code Example: Max of an array with SEND and RECEIVE

```
int main(int argc, char *argv[]) {
  MPI_Init(&argc, &argv);
  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  int max = 0;
  if (rank == 0) {
    int length;
    int* array = read_array_file(&length);
    int part = length / size; // elements per process
    for (int to = 1; to < size; to++) {
      int offset = to * part;
      if (to == size - 1) {
        part = length - offset; }
      // Send length of partition for each process first
      MPI_Send(&part, 1, MPI_INT, to, 0, MPI_COMM_WORLD);
      MPI_Send(array+offset, part,MPI_INT,to,0,MPI_COMM_WORLD);}
    // Process with rank 0 also calculates a partition
    for (int i = 0; i < part; i++) {
      if (array[i] > max) {
        max = array[i]; } }
  } else {
    int part;
    MPI_Recv(&part,1,MPI_INT,0,0,MPI_STATUS_IGNORE);
    int array[length];
    MPI_Recv(&array,length,MPI_INT,0,0,MPI_STATUS_IGNORE);
    for (int i = 0; i < length; i++) {
      if (array[i] > max) {
        max = array[i]; } } }
  int result;
  MPI_Reduce(&max,&result,1,MPI_INT,MPI_MAX,0,MPI_COMM_WORLD);
  if (rank == 0) { printf("max: %i\n", result); }
  MPI_Finalize(); }
```

Multi-Threading HPC in MPI with OpenMP (cores on CPU)

So far, we've used CUDA for parallelization on the GPU and we've used MPI to distribute tasks to multiple CPUs. OpenMP on the other hand is used to distribute tasks on a single CPU to its cores. All threads have access to a shared memory.
OpenMP program starts as single thread which is the master thread. It executes sequentially. Per default are numbers of threads set by the number of cores.

```
export OMP_NUM_THREADS=10 // # of threads as env variable
omp_set_num_threads(nProcessors) // # of threads set in program
gcc -fopenmp helloOpenMP.c // Compilation of program
```

Code Example: simple Fork and Join

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char* argv[]) {
  const int np = omp_get_max_threads();
  printf("OpenMP with threads %d\n",np );
  #pragma omp parallel
  {
    const int np = omp_get_num_threads();
    printf("Hello from thread %d\n", omp_get_thread_num()); }
  return 0; }
```

Code Example: For Loop

Iterations will be distributed across available threads. Each thread processes one iteration at a time. Execution returns to the initial thread.

```
#pragma omp parallel for
for (i = 0; i < n ; i++) {
    printf("Iteration %d, thread %d\n",i,omp_get_thread_num());}
```

Memory Model Code

Example for private A and shared B variable (after loops ends, A will be removed from memory):

```
int A, B;
#pragma omp parallel for private (A) shared (B)
for (...)
```

Alternative for a private A variable:

```
#pragma omp parallel
int A = 0;
#pragma omp for
for (...)
```

Private Variables:
→ Variables declared inside a parallel region or with private(…) clause.
→ Loop iteration variables of parallel loops (`#pragma omp parallel for`).
Shared Variables:
→ Variables declared outside a parallel region.
→ Variables explicitly declared as shared with the shared(…) clause.
→ Static variables declared inside a parallel region (retain their value between parallel regions).

Code Example: Critical Section

Really slow and goes back to being sequentially. Overkill for this kind of task.

```
int sum = 0;
#pragma omp parallel for
for (int i = 0; i < n ; i++) {
  #pragma omp critical
  { // only one thread at a time executes this section
    sum += i; } }
```

Code Example: Lightweight Mutex

```
int sum = 0; int i;
#pragma omp parallel for
for (i = 0; i < n ; i++) {
  #pragma omp atomic
  {    sum += i;  } }
```

Code Example: Reduction across threads

This is the best solution for summarization with multiple Threads.

```
int sum = 0;
#pragma omp parallel for reduction(+: sum)
  for (int i = 0; i < n ; i++) {  sum += i;  }
```

Code Example approximating PI with Multithreading & Cluster

Main Method:

```
  MPI_Init(&argc, &argv);
  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  long numThreadTosses = numTosses/(numThreads * size);
  if (rank == 0) {
    printf("Not trials: %i, nof processes: %i,
      nof Threads %d\n", numTosses, size, numThreads); }
  srand(rank * 4711);
  long hits = count_hits(numThreadTosses);
  MPI_Barrier(MPI_COMM_WORLD);
  long total;
  MPI_Reduce(&hits, &total, 1, MPI_LONG, MPI_SUM, 0, MPI_COMM_WORLD);
```

Count_hits Methode:

```
long count_hits(long trials) {
  long hits = 0;  long i;  double x,y;
  #pragma omp parallel
  {
    #pragma omp for reduction(+:hits) private(x,y)
    for (i = 0; i < trials; i++) {
      double x = random_double();
      double y = random_double();
      if (x * x + y * y <= 1) { hits++; }  }  }
  return hits; }
```

Vector Parallelism (SIMD)

Not vectorizable: read after write & write after write

SIMD Implementations

MMX (Intel) (64bit), **3DNow! (AMD), SSE**(128bit), **AVX** (256bit), **AVX-512**
• Different generations and implementation of the same concept
• Supports most of the primitive data-types (Short, Int, Float, Double, Byte, Bit)

Java Vector API

Code unten zeigt vectorized Array-Addition, JVM macht aber Auto-Vectorisation, somit hat normaler sequentieller Code die gleiche Performance

```
private static final VectorSpecies<Integer> SPECIES =
IntVector.SPECIES_PREFERRED; //statt Int alle Datentypen möglich
public static int[] vectorComputation(int[] a, int[] b) {
 var c = new int[a.length];
 int upperBound = SPECIES.loopBound(a.length);
 int i = 0;
 for (; i < upperBound; i += SPECIES.length()) {
 var va = IntVector.fromArray(SPECIES, a, i);
 var vb = IntVector.fromArray(SPECIES, b, i);
 var vc = va.add(vb);
 vc.intoArray(c, i); }
 for (; i < a.length; i++) { /* Cleanup loop für nicht in Lane
passende Array-Membern(z.B. bei AVX-512 Platz für 16 32bit Ints)
 c[i] = a[i] + b[i]; } return c; }
```

Monitor Checklist: synchronized, notifyAll(), throws InterruptedException
throws InterruptedException bei Monitor.wait(), Semaphore.acquire(), Thread.run()