# 1 Multi-Threading Grundlagen

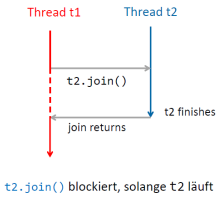
## 1.1 Thread Scheduling

**Processor Sharing:** Mehr Threads als Prozessoren und bei Wartebidingung Proz. an anderen bereiten Thread abgeben.
**Verzahnte Ausführung:** Prozessor führt Instruktionen von mehreren Threads in Teilsequenzen aus. → Quasiparallelität
**Synchron:** Warten auf Bedingung → Waiting Threads
**Asynchron:** Zeitablauf → Nach gewisser Zeit Proz. abgeben
**Kooperativ:** Threads müssen explizit beim Scheduler in Abständen Kontextwechsel synchron initiieren
**Preemptiv:** Scheduler kann per Timer-Interrupt den laufenden Thread asynchron unterbrechen

## 1.2 Multi Thread Programmierung

**JVM ist ein Prozess im Betriebssystem.** → Programmierer kann weitere Threads starten
Die JVM läuft, solange Threads laufen.
**Ausnahme:** Deamon Threads (z.B. Garbage Collector). Mit System.exit()/Runtime.exit() kann die JVM direkt terminiert werden (unsauber)

```
// A,B Ausgaben können durcheinander sein:
public class MultiThreadTest {
   public static void main (String[] args ) {
      var a = new Thread(Thread(() -> multiPrint("A"));
      var b = new Thread(Thread(() -> multiPrint("B"));
      a.start(); b.start();
      System.out.println("main finished");
   }
   static void multiPrint (String label) {
      for (int i = 0; i < 10; i++) {
         System.out.println(label + ": " + i);
      } }
// Explizite Runnable-Implementation:
class SimpleLogic implements Runnable {
   @Override
   public void run () {
      // thread behavior
   } }
var myThread = new Thread(new SimpleLogic());
myThread.start();
// Sub-Klasse von Thread
class SimpleThread extends Thread {
   @Override
   public void run ()
      // thread behavior
   } }
var myThread = new SimpleThread(); myThread.start();
```




t1 wartet, bis t2 fertig ist.
**Warten:**
Thread.sleep(milliseconds)
**Prozessor freigeben:**
Thread.yield()
**Von aussen Unterbrechen:**
myThread.interrupt()
**Aktuelle Thread-Instanz:**
Thread currentThread()
**Als Deamon markieren:**
void setDeamon(boolean on)

# 2 Thread synchronisation

**Threads teilen sich Adressraum und Heap.**

## 2.1 Race Condition

Wenn mehrere Threads das Selbe Objekt lesen und anhand vom Resultat dieses überschreiben. Time of Check and Time of Use Problem: Operationen sind nicht genügend Atomar: Ein andere Thread könnte dazwischen schreiben. → Visibility oder Lost Update problem

## 2.2 Synchronized (Monitor Lock)

Überholproblem und keine garantierte Reihenfolge
Das Keyword synchronized belegt einen Lock für das Objekt. → Nur ein Thread kann eine synchronized Methode in derselben Instanz zur gleichen Zeit ausführen. wait() gibt lock frei und wartet auf Signal. notify() signalisiert nur einen beliebigen (zufälligen), wartenden Thread. → Wartet vlt auf andere Bedingung und kann zu ewigem Warten führen
notifyAll() signalisiert alle wartenden Threads.
**Wichtig:**
- Signalisierender Thread behält Monitor
- Thread ist im inneren Warteraum, bis Signal kommt
- Bei notify() kommt Thread wieder in äusseren Warteraum

```
class BoundedBuffer <T> {
  private Queue<T> queue = new LinkedList<>();
  private int limit = 1; // or initialize in constructor
  public synchronized void put(T item) throws
        InterruptedException {
    while (queue.size () == limit) {
      wait(); // await non full
    } queue.add(item); notifyAll(); } // signal non empty
  public synchronized T get() throws
        InterruptedException {
    while (queue.size () == 0) {
      wait();// await non empty
    } var item = queue.remove(); notifyAll(); // non full
    return item; } }
// synchronized(object){ statements } ist eine explizite
      Angabe, auf welcher Instanz gelockt wird:
public void deposit(int amount) {
   synchronized(this) {this.balance += amount;} }
```

# 3 Spezifische Synchronisationsprimitiven

## 3.1 Semaphor

**Ziel:** Vergabe einer beschränkten Anzahl freier Ressourcen. Semaphor ist also ein Objekt mit Zähler.
**acquire():** Bezieht freie Ressource. Wartet, falls keine verfügbar (Zähler <= 0). Dekrementiert Zähler wenn erfolgreich.
**release():** Ressource freigeben. Zähler inkrementieren.
**Verwendung:** new Semaphore(N); oder new Semaphore(N, true); → true für Fairness (FIFO-Prinzip), aber langsamer

```
// Mutex Semaphore als alternative zu synchronized
class BoundedBuffer <T> {
  private Queue<T> queue = new LinkedList<>();
  private Semaphore upperLimit=new Semaphore(cap, true);
  private Semaphore lowerLimit = new Semaphore(0, true);
  // private Semaphore mutex = new Semaphore(1, true);
  public void put(T item) throws InterruptedException {
    upperLimit.acquire();
    synchronized (queue) {queue.add (item);}
    // mutex.acquire(); queue.add(item); mutex.release();
    lowerLimit.release(); }
  public T get() throws InterruptedException {
    T item; lowerLimit.acquire();
    synchronized (queue) { item = queue.remove(); }
    // mutex.acquire(); T item = queue.remove(); mutex.
        release();
    upperLimit.release(); return item; } }
```

## 3.2 Lock & Condition

**Ziel:** Monitor mit mehreren Wartelisten für verschiedene Bedingungen. Braucht kein synchronized, wait, notify, notifyAll

```
class BoundedBuffer<T> {
  private Queue<T> queue = new LinkedList<>();
  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(); } }
```

## 3.3 Count Down Latch

**Ziel:** Synchronisationsprimitive mit Count Down Zähler. Threads können warten, bis Zähler <= 0 ist. countDown(), um Zähler zu dekrementieren aber kein countUp().
await() um auf Signal zu warten.

## 3.4 Cyclic Barrier

**Ziel:** Treffpunkt für fixe Anzahl Threads. Ist wiederverwendbar. Hat nur Funktion countDown()

```
// CountDown Latch
var ready = new CountDownLatch(N); //Warte auf N Threads
var start = new CountDownLatch(1); // Einer gibt Signal
// Cyclic Barrier
var start = new CyclicBarrier(N); // Kein start nötig
```

# 4 Gefahren der Nebenläufigkeit

## 4.1 Race Condition

**Problem:** Ungenügend synchronisierte Zugriffe auf gemeinsame Ressourcen. → Falsche Resultate oder falsches Verhalten möglich. (meistens wegen Data-Race)
**Ohne Data Race weil:** Critical Sections nicht geschützt.
**Kombinationen:**
Data Race + Race Condition: Fehlerhaftes Programmverhalten
Data Race ohne Race Condition: Korrektes Programmverhalten, aber formal falsch
Race Condition ohne Data Race: Fehlerhaftes Prog.verhalten
Weder noch: Richtig
**Wann kann man auf Synchronisation verzichten?**
**Immutability** (Unveränderlichkeit): Instanzvariablen sind alle final, Methoden mit nur Lesezugriff → Konstruktor initialisiert die Instanzvariablen.
**Confinement** (Einsperrung): Objekt gehört nur einem Thread zu einer Zeit. Oder Objekt ist in anderem bereits synchronisiertem Objekt eingekapselt

## 4.2 Deadlocks

**Problem:** Gegenseitiges Aussperren von Threads. Zwingend 2 Lock Instanzen
**Deadlock Vermeidung:**
- Lineare Sperrordnung der Ressourcen einführen → Nur geschachtelt in aufsteigender reihenfolge sperren
- Grobgranulare Locks wählen → Wenn lineare nicht möglich

## 4.3 Starvation

**Problem:** Kontinuierliche Fortschrittsbehinderung von Threads wegen Fairness-Problem. → Ist ein Liveness/Fairness Problem
**Starvation Vermeidung:**
- Länger wartende Threads haben Vortritt
- Fairness einschalten mit Semaphore, Lock & Condition, Read-Write Lock
- Java Monitor hat ein Fairness-Problem

# 5 Thread Pools

## 5.1 Konzept und Funktionsweise

**Tasks:** Implementieren potentiell paralelle Arbeitspakete. Auszuführende Tasks werden in Warteschlange eingereiht.
**Thread Pool:** Beschränke Anzahl von Worker-Threads. Holen Tasks aus der Queue und führen sie aus.

## 5.2 Vorteile und Einschränkungen

**Vorteile:**
- Beschränkte Threadanzahl (Zu viele verlangsamen System)
- Recycling der Threads (Thread-Erzeugung und Freigabe)
- Höhere Abstraktion (Trenne TaskBeschreibung/Ausführung)
- Anzahl Threads pro System konfigurierbar

**Einschränkungen:**
- Tasks dürfen nicht aufeinander warten (Deadlock)
- Task muss zu Ende laufen, bevor Worker Thread anderen Task ausführen kann (Ausnahme: Geschachtelte Tasks)

## 5.3 Fork & Join Pool

**Future Konzept:** Repräsentiert ein zukünftiges Resultat. Proxy wartet auf Resultat, muss das Ende der Berechnung abwarten.

```
// Future Verwendung
var threadPool = new ForkJoinPool();
Future<Integer> future = threadPool.submit(() -> {
   return value; }); // Tasks können Rückgabe haben
int result = future.get() // Blockiert, bis Task beendet
// Einfaches Beispiel:
var left = threadPool.submit(() -> count(leftPart));
var right = threadPool.submit(() -> count(rightPart));
result = left.get() + right.get();
```

### 5.3.1 Rekursive Tasks

Tasks können Untertasks starten und abwarten. Erben von RecursiveTask<T>
- T compute(): Task Implementierung
- fork(): Starte als Sub-Task in einem anderen Task
- T join(): Warte auf Task-Ende und frage Resultat ab
- T invoke(): Ein Sub-Task starten und abwarten
- invokeAll(): Mehrere Sub-Tasks starten und abwarten

```
class PairwiseSum extends RecursiveAction {
  private final int[] array;
  private final int lower, upper;
  private static final int THRESHOLD = 1;//configurable
  public PairwiseSum(int[] array, int lower, int upper){
    this.array = array;
    this.lower = lower;
    this.upper = upper; }
  @Override protected void compute() {
    if (upper - lower > THRESHOLD) {
      int middle = (lower + upper) / 2;
      invokeAll(
        new PairwiseSum(array, lower, middle),
        new PairwiseSum(array, middle, upper) );
    } else {
      for (int i = lower; i < upper; i++) {
        array[2 * i] += array[2 * i + 1];
        array[2 * i + 1] = 0; } } } }
```

## 5.4 Asynchrone Programmierung

**Ziel:** Aufrufer soll während der Operation weiterarbeiten. → Operation in Thread oder Thread Pool auslagern

```
// Reihenfolge der Downloads und GUI erhalten bleiben
void download(List<URL> links, OutputStream output) {
  if (links.isEmpty()) { statusLabel.setText("Done"); }
  var url = links.get(0);
  statusLabel.setText("Downloading " + url);
    if (cancelBox.isSelected()) {
      statusLabel.setText("Cancelled!");
      return; }
    CompletableFuture.runAsync(() -> {
      url.openStream().transferTo(output);
      var remaining = links.subList(1, links.size());
      SwingUtilities.invokeLater(() -> download(remaining
          , output)); }); }
// Asynchron mit CompletableFuture
CompletableFuture<File> zipAsync(File[] files) {
  statusLabel.setText("zip started");
  return CompletableFuture.supplyAsync(() -> {
    File[] temp = new File[files.length];
    for (int i = 0; i < files.length; i++) {
      temp[i] = compress(files[i]); }
    SwingUtilities.invokeAndWait(() -> statusLabel.
        setText("all files compressed"));
    File output = archive(temp);
    SwingUtilities.invokeAndWait(() -> statusLabel.
        setText("zip completed"));
    return output; }); }
// Parallel und asynchron
void scanAsync(List<File> list, String pattern) {
  List<CompletableFuture<Void>>futures=new ArrList<>();
  for (File file : list) {
    futures.add(CompletableFuture.runAsync(() -> {
      if (search(file, pattern)) {
        print("Found " + file); } })); } //SwingUtilities
          mit invokeAndWait falls auf GUI schreiben
  CompletableFuture.allOf(futures.toArray(new
CompletableFuture[0])).thenRun(() -> print("Done")); }
```

### 5.4.1 Continuation

**Ziel:** Folgeaufgabe an asynchrone Aufgabe anhängen. Ausführung der Continuation durch beliebigen Thread.

```
// Continuation:
future.thenAccept(result -> syso(result));
// Tasks verketten:
future.thenApplyAsync(second).thenAcceptAsync(third);
// Multi-Continuation:
CompletableFuture.allOf(future1, future2)
.thenAcceptAsync(continuation); //Warten aufeinander
CompletableFuture.any(future1, future2)
.thenAcceptAsync(continuation)//Sobald einer fertig ist
// runAsync() ist Fire and Forget. Workers sind Deamons
      und ignoriert Exceptions
```

# 6 Task Parallel Library

## 6.1 Threading in .NET

Keine Vererbung: Delegate bei Konstruktor. Exception in Thread führt zu Abbruch des Programms.
**.NET Monitor:** FIFO Warteschlange, wait() in schlaufe, PulseAll() für aufwecken, Synchronisation mit Hilfsobjekt ist Best Practice, Kein Fairness-Flag, Kein Lock & Condition.
**Zusätzlich:** ReadWriteLockSlim für Upgradeable Read/Write, Semaphoren auch auf OS-Stufe nutzbar, Mutex (Binärer Semaphore auf OS-Stufe)

```
var myThread = new Thread(() => {
  for (int i = 0; i < 100; i ++) {
    Console.WriteLine("MyThread step {0}", i); } });
myThread.Start(); myThread.Join();
// Monitor in .NET
class BankAccount {
  private decimal balance;
  private object syncObject = new() //Monitor Hilfsobj.
  public void Withdraw(decimal amount) {
    lock (syncObject) { // Analog zu synchronized
      while (amount > balance) {
        Monitor.Wait(syncObject); }
```

```
    balance -= amount; } }
  public void Deposit(decimal amount) {
    lock (syncObject) {
      balance += amount;
      Monitor.PulseAll(syncObject)}}} // wie notifyAll
```

### 6.2 Task Parallelität

**Vorteil:** Effizienz, durch wenig Contention (eigene Warteschlange) **Nachteil:** Fairnessproblem bei unausgeglichener Verteilung (und LIFO) bei Subtasks.
**Work Stealing Thread Pool** mit Abstraktionsstufen:
Task Parallelität: Explizite Tasks starten und warten
Data Parallelität: Parallele Statements und Queries
Asynchrone Programmierung: Mit Continuation Style

```
task task = Task.Run(() => { implementation });
task.Wait() // Warten auf task
Task<int> task = Task.Run(() => {return 2;});
Console.Write(task.Result) // Wartet auf Task Ende
```

### 6.3 Datenparallelität

```
// Parallele Statements wartet bis alle fertig sind
Parallel.Invoke(
  () => MergeSort(l,m), () => MergeSort(m,r));
// Parallele Loop
Parallel.ForEach(list, file => Convert(file));
// Falls Iteration unabhängig: Anzahl Threads: Freie
     Worker Threads (logische Prozessoren)
Parallel.For(0, array.Length, i => doSmth(array[i]));
```

### 6.4 Asynchrone Programmierung

```
var task = Task.Run(LongOperation); // ist asynchron
int result = task.Result;
// Task Continuations (Wie CompletableFutures
task1.ContinueWith(task2).ContinueWith(task3);
// Multi-Continuation
Task.WhenAll(task1, task2).ContinueWith(continuation);
Task.WhenAny(task1, task2).ContinueWith(continuation);
```

## 7 GUI und Threading

### 7.1 GUI & Threading

GUI Frameworks erlauben nur **Single-Threading**: Nur spezifischer UI-Thread darf auf UI-Komponente zugreifen.
**GUI Implikationen:** Keine langen Operat. in UI Events, Kein Zugriff durch fremde Threads (blockieren & Race Condition)
**UI Interaktion:** UI Operationen müssen als Events in die UI Event Queue eingereiht werden.
**Dispatching:** Benutzung der Klasse SwingUtilities

```
button.addActionListener(event -> {
  new Thread(() -> {
    var text = readHugeFile();
    SwingUtilities.invokeLater(() -> { // asynchron
      textArea.setText(text); });//synchron:invokeAndWait
    }).start(); });
// Sauberer Setup: frame.pack() und setVisible(true) (
     Auf JFrame bezogen) in SwingUtilities ausführen
// Background Worker:
class BackgrCalc extends SwingWorker<Integer, Void> {
  @Override // Int: Resultat, Void: Zwischenresultat
  public Integer doInBackground () {
    return longComputation (); }
  @Override // UI Thread
  protected void done() {
    try { int result = get();
      label.setText("Result: " + result);
    } catch (InterruptedException|ExecutionException e) {
    } } } // get() gibt Resultat von doInBackground
```

**.NET UI Thread Modell:** Gleiches Prinzip wie in Java.

```
// UI Thread ist der Aufrufen von Application.Run()
// UI Event Dispatching:
control.Dispatcher.InvokeAsync(action); // WPF
control.BeginInvoke(delegate) // WinForm
// Sind beide asynchron. Synchron mit Invoke
```

### 7.2 C# async/await

async für Methode (Aufrufer wird blockiert). await für Tasks (Warten auf Ende eines TPL Task).
**Rückgabetypen:** void: Fire & Forget, Task: Keine Rückgabe, erlaubt Warten, Task<T> Rückgabetyp T.
**Ausführungsmodell:** Aufrufer führt async Methode synchron aus, bis ein blockierendes await anliegt. Erst danach asynchron.

```
async Task<string> ConcatWebSitesAsync(string url1,
      string url2) {
  HttpClient client = new HttpClient();
  Task<string> download1 = client.GetStringAsync(url1);
  Task<string> download2 = client.GetStringAsync(url2);
  string site1 = await download1;
  string site2 = await download2;
  return site1 + site2; }
```

## 8 Memory Models

### 8.1 Java Memory Model

**Problem bei Weak Consistency:** Speicherzugriffe werden in verschiedenen Reihenfolgen von verschiedenen Threads gemacht. (Ausnahme: Synchronisationen/Speicherbarrieren)
**Problem bei Omptimierungen:** Betrifft Compiler, Laufzeitsystem und CPUs. Sie ordnen Instruktionen um
→ Keine sequentielle Konsistenz bei Nebenläufigkeit
**Atomicity:** Einzelnes lesen und schreiben ist atomar bei primitiven Datentypen bis 32 Bit → Sonst volatile Keyword.
**Visibility:** Sieht Änderungen eines anderen Thread eventuell nicht oder viel später (z.B. wegen Compiler-/JIT-Optimierung)
**Visibility Garantien:**

- Locks Release & Acquire: Änderungen vor Release werden bei Acquire sichtbar
- Volatile Variable: Änderungen bis zum write werden beim read sichtbar
- Thread/Task-Start und Join: Bei Start Eingabe und bei Join Ausgabe
- Final Variablen: Nach Ende des Konstruktors sichtbar

**Ordering Garantien:**

- Program Order: Sequentielles Verhalten jedes einzelnen Thread bleibt erhalten
- Synchronization Order: Syncbefehle werden zueinander nie umgeordnet (Locking, volatile, Thread start/join)
- Happens-Before Relation: Alles andere kann umgeordnet werden (ausser garantierte Sichtbarkeit unter Threads)

**Atomic Klassen:** Gibt es für Boolean, Integer, Long und Referenzen (auch für Array Elemente). Divere atomare Operationen wie addAndGet(), getAndAdd() etc.

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

### 8.2 .NET Memory Model

**Unterschied zu Java Memory Model:**
Atomacity: long/double nicht mit volatile atomar. Visibility: Nicht definiert, implizit durch Ordering. Ordering: Nur Half und Full Fences → Atomare Instruktionen mit Interlocked Klasse
**.NET Volatile Half Fences:**
**Volatile Write:** Vorangehende Zugriffe bleiben davor **Volatile Read:** Nachfolgende Zugriffe bleiben danach
**.NET Volatile Full Fences:**
Thread.MemoryBarrier(); → Verbietet Umordnung in beide Richtungen.

## 9 Actor Model

**Jeder Actor hat seinen eigenen lokalen/isolierten Speicher** **Akka:** Java ist Shared Memory basiert (Heap). Daher: Aufpassen, dass Actoren keine Referenzen auf gemeinsame Heap-Objekte verwenden. (ActorRef stattdessen)
Aktive Objekte: Objekte haben nebenläufiges Innenleben
Kommunikation: Objekte senden und empfangen Nachrichten
Kein Shared Memory: Nur Austausch von Nachrichten über Kanäle/Mailboxen
**Akka Empfangsverhalten:**
**Reaktion auf ankommende Nachricht:** Spezielle Behandlungs Methode wird ausgeführt. **Effekte per Behandlung:** Ändere privaten Zustand, Sende nachrichten, Erzeuge neue Actors. **Intern sequentiell:** Nur eine Nachricht auf einmal bedienbar (Buffer)

```
// Einfacher Akka Actor
public class NumberPrinter extends UntypedActor {
  public void onReceive(final Object message) {
    if (message instanceof Integer) {
      System.out.println(message); } }
// Erzeugen und Senden
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();
```

## 10 GPU Parallelisierung

Cores innerhalb Streaming Multiprocessor sind nur mit Vektor Parallelisierung effizient nutzbar. → Cores führen dieselbe Instruktion auf unterschiedlichen Daten/Speicherstellen aus
**NUMA Modell:** Non-Uniform Memory Access
**Kein gemeinsamer Hauptspeicher zwischen GPU und CPU** → Explizites Übertragen. **Unterschiedlicher Instruktionssatz/Architektur** → Code für GPU kompilieren/designen

### 10.1 CUDA (Computer Unified Device Architecture)

**CUDA Threads:** Gleicher Kernel wird von mehreren Threads ausgeführt. **SIMT:** Single Instruction Multiple Threads. **CUDA Blocks:** Threads sind in Blöcke gruppiert. → Ein Block: Gleicher SM, Threads können innerhalb von Block interagieren.
**CUDA Ausführungsmodell:** **Thread:** Virtueller Skalarprozessor. **Block:** Virtueller Multiprozessor. **Blöcke müssen unabhängig sein**: Run To Completion, Beliebige Ausführungsreihenfolge (sequentiell, parallel), Blocks in Threadpool
**Datenaufteilung:** Jede Kernel-Ausführung bestimmt seinen Datenteil. threadIdx.x: Nummer des Threads innerhalb Block, blockIdx.x: Nummer des Blocks, blockDim.x: Blockgrösse

```
// CUDA Kernel
__global__ // Läuft auf GPU (Device)
void VectorAddKernel (float *A,float *B,float *C,int N){
  int i = blockIdx.x * blockDim.x + threadIdx.x; //
       eindeutiger Index basierend auf Block & Thread ID
  if (i < N) { C[i] = A[i] + B[i]; } }
int main() { // Läuft auf CPU (Host)
  //kernel invocation
  int blockSize = 1024;
  int gridSize = (N + blockSize -1) / blockSize;
  VectorAddKernel<<<gridSize, blockSize>>>(A, B, C, N);}
// Mehrdimensional:
dim3 blockSize(32, 32);
dim3 gridSize((N+31)/32, (M+31)/32);
```

**CUDA Ausführung:**

- Auf GPU allozieren: cudaMalloc
- Daten zu GPU transferieren: cudaMemcpy
- Kernel ausführen: <<<gridD, blockD>>>
- Daten von GPU zu PU transferieren: cudaMemcpy
- Auf GPU deallozieren: cudaFree

```
void CudaVectorAdd(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);
  int blockSize = 1024;
  int gridSize = (N + blockSize - 1) / blockSize;
  VectorAddKerne<<<gridSize,blockSize>>>(d_A,d_B,d_C,N);
  cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
  cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
//handleCudaError(...) um Malloc, Memcpy und Free
```

## 11 GPU Parallelisierung 2

**Speichermodell:**
Global Memory ist relativ teuer (ca. 600 Zyklen). Die Threads lesen bei Matrix Multiplikation wiederholt die selben Elemente von A & B. → Effizienzsteigerung mit Cache Speicher
**Synchronisation:**
__syncthreads() synchronisiert alle Threads innerhalb eines Blocks. (Nicht zwischen Blöcken)
**Warps:** Ein Block wird intern in Wraps zerlegt (zu je 32 Threads). Alle Threads führen die gleiche Instruktion aus. Verzweigungen (if, switch, while, do, for) werden abwechselnd ausgeführt (Divergenz) → Einzele Threads müssen warten
**Effizienzsteigerung durch Coalescing:**
Jeweils ein Tile mit Coalescing von Device Memory ins Shared Memory einlesen. Syncthreads. Von Shared Memory mit Coalescing zurück ins Device Memory schreiben.

## 12 Cluster Parallelisierung

**Ziel:** Ein Programm auf mehreren Nodes ausführen. → Kein Shared Memory (NUMA) zwischen Nodes. Shared Memory (SMP) für Cores innerhalb von Node
**MPI:** Verteiltes Programmeirmodell, Basiert auf Actor/CSP-Prinzip. Jeder Prozess hat seine Identifikation (Rank) und arbeitet unabhängig in seinem Adresraum. Starten und terminieren synchron → Können untereinander Kommunizieren (Nachrichten Senden/Empfangen), Synchronisation mit Barrieren

```
#include<stdio.h> #include"mpi.h"
int main(int argc, char* argv[]) {
  MPI_Init(&argc, &argv); // MPI Initialisierung
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);//Prozess Identif
  printf("MPI process %i", rank);
  MPI_Finalize(); // MPI Finalisierung
  return 0; }
MPI_Comm_size(MPI_COMM_WORLD, &size); // Anzahl Prozesse
MPI_Barrier(MPI_COMM_WORLD); // Barriere für alle Proz.
MPI_AllReduce(&value, &total, 1, MPI_INT, MPI_SUM,
      MPI_COMM_WORLD); // Aggregation von Teilresultaten
// Senden/Empfangen eines Int-Wertes (tag: Frei wählbare
       Nummer für Nachrichtenart >=0)
MPI_Send(&value, 1, MPI_INT, receiverRank, tag,
      MPI_COMM_WORLD);
MPI_Recv(&value, 1, MPI_INT, senderRank, tag,
      MPI_COMM_WORLD, MPI_STATUS_IGNORE);
// Monte Carlo:
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
srand(rank * 4711) // Willkürlicher Seed
long hits = count_hits(TRIALS / size) // Nur Bruchteil
long total;
MPI_Reduce(&hits, &total, 1, MPI_LONG, MPI_SUM, 0,
      MPI_COMM_WORLD); // Prozess 0 erhält Gesamtwert
if (rank==0) {double pi = 4 * ((double)total / TRIALS);}
// Count_hits:
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; }
```

## 13 Concurrency in Python & JavaScript

### 13.1 Python Concurrency

**Global Interpreter Lock (GIL):** Nur ein Thread kann Python Byte-Code ausführen. Kein Speedup für CPU-Bound Operationen möglich. Data Races sind dennoch möglich (Reordering möglich, Visibility nicht garantiert) → Kein Memory Model
**Shared Memory bei Prozessen:** Muss explizit definiert werden, Erfordert Angabe eines Type-Codes oder ctypes, Als Argument an die Startfunktion des Prozesses, Können Teil einer Klasse sein.
**asyncio (async/await):** Keine parallele Ausführung. → async Methoden werden erst beim zugehörigem await ausgeführt

## 14 Others

```
// Monitor mit Semaphore
public class Monitor {
  private final Semaphore monitorLock =new Semaphore(1);
  private final Semaphore innerWaiting=new Semaphore(0);
  private final Semaphore innerPulsed =new Semaphore(0);
  private int waitingThreads = 0;
  public void lock() throws InterruptedException {
    monitorLock.acquire(); }
  public void unlock() { monitorLock.release(); }
  public void await() throws InterruptedException {
    waitingThreads++; monitorLock.release();
    innerWaiting.acquire(); innerPulsed.release();
    monitorLock.acquire(); }
  public void signalAll() throws InterruptedException {
    innerWaiting.release(waitingThreads);
    innerPulsed.acquire(waitingThreads);
    waitingThreads = 0; } }
// Akka
class ParProgActor extends UntypedActor {
  private final int number; private ActorRef next;
  private boolean pendingSend;
  private int value = 0;
  public ParProgActor(int number) {
    this.number = number;
    this.pendingSend = number == 0; }
  public void onReceive(Object message) {
    if (message instanceof ActorRef) {
      next = (ActorRef)message;
    } else if (message instanceof Integer) {
      this.value = (int)message;
      if (number == 0) {
        System.out.println("result: " + value);
      } else { pendingSend = true; } }
    if (pendingSend) { pendingSend = false;
      next.tell(value + number, getSelf()); } } }
```