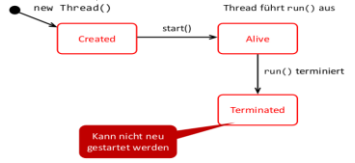
ParaProg

Multithreading

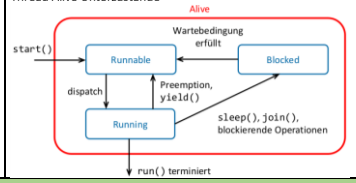
Thread(Leichtgewichtsprozess) : Parallele Ablaufsequenz innerhalb eines Prozesses. Adressraum wird geteilt
Prozess(Schwergewichtsprozess) : Parallel laufende Programminstanz im System mit eigenem Adressraum
Parallelität : Zerlegung eines Ablaufs in mehrere Teilprozesse, welche gleichzeitig auf mehreren Prozessoren laufen
Nebenläufigkeit : Gleichzeitig oder verzahnt ausführbare Abläufe, welche auf eine gemeinsame Ressource zugreifen
Sync Kontextwechsel : Thread wartet auf Bedingung & gibt dann Ressource wieder frei(*kooperativ)*.reiht sich als wartend ein.
Async Kontextwechsel : Thread gibt Ressource nicht von sich wieder frei. Scheduler unterbricht per Timer-Interrupt laufenden Thread(*preemtiv)*

Thread.sleep(ms) ms warten und dann wieder ready **Thread.yield()** gibt CPU frei, direkt wieder in ready → mehr Threadwechsel
DaemonThread : JVM terminiert wenn alle Threads beendet sind mit Ausnahme der Daemon Threads (t1.setDaemon(true))
Aktueller Thread: Thread.currentThread(); **Join:** t1.join() wartet bis Thread t1 terminiert. **Erstellen:** Thread / Runnable / Lamda

Java Thread Lifecycle



Thread Alive Unterzustände



Synchronisationsprimitiven

Schutz vor Shared Ressourcen bei Multi-Threading : Monitor, Semaphore, Lock&Condition, RW-Lock	Zeitlicher Synchronisationspunkt von mehreren Theads : Latch, CyclicBarrier, Phase, (Semaphore)

Monitor (Gegenseitiger Ausschluss + Wait&Signal Mechanismus) immer fairness Problem → Starvation Gefahr

Jedes Objekt hat Monitor-Lock. Beziehen mittels synchronized-Keyword. notify()/notifyAll()/wait() nur in synchronized Block, ansonsten IllegalMonitorStateException. notifyAll() bei mehreren Wartebedingungen nicht effizient, da viele Kontextwechsel & hohe Synchronisationskosten. notify() nicht fair, einige Threads könnten sogar nie dran kommen -> besser notifyAll() verwenden

wait()	1. In Inneren Warteraum gehen 2. Monitor freigeben 3. Inaktiv bis Wecksignal 4.Monitor neu beziehen (Äusseren Warteraum)
notify()	1. Weckt alle Threads in Warteraum 2. Behält das Monitor Lock (Keine FIFO Garantie, Spurious Wakeup möglich)

```
class BankAccount {
   private int balance = 0;
   public synchronized void deposit(int amount) {
      balance += amount;
      notifyAll();     }
   public synchronized void withdraw(int amount) throws InterruptedException {
      while (amount > balance) { wait(); }
      balance -= amount;   }
   public synchronized int getBalance() {
      return balance;   }   }
```

Monitor auf Objekt: synchronized(this){..}
Monitor auf statische Methode:
synchronized(this.Class)

```
public void method(){
   synchronized(this){
      //code
   }
}
```

Typische Fallen:
- wait() mit if : Condition könnte durch "schnelleren" Thread invalidiert werden bevor Monitor bezogen wird →Überholproblem
- Spurious Wakeup: wait() könnte auch ohne Signalisation returnen, es besagt nur, dass sich etwas geändert haben *könnte.*
- Single notify(): Falls unterschiedliche Wartebedingungen in Schlaufe mit wait() drin muss notifyAll() verwendet werden!

Semaphor (Vergabe einer beschränkten Anzahl freier Ressourcen, Objekt mit Zähler)

Vergleich Monitor :Semaphor ist schneller und konstanter. Fairness kann mit Flag konfiguriert werden.

acquire()	Bezieht freie Ressource => Dekrementiert Zähler. Wartet falls keine verfügbar (Count <= 0) throws InterruptedException → allenfalls mit try, finally Semaphore wieder frei geben!
release()	Gibt eine Resource frei => Inkrementiert Zähler. Benachrichtigt Wartende

```
Class BoundedBuffer<T> {
  private Queue<T> queue = new LinkedList<>() ;
  private Semaphore upperLimit = new Semaphore(capacity, true) ; // fair
  private Semaphore lowerLimit = new Semaphore(0, true) ;
  private Semaphore mutex = new Semaphore(1, true) ; // gewährleistet Atomarität vgl mit synchronize Funktionalität bei Monitor
  public void put(T item) throws InterruptedException { // multi acquire(int permits) und multi release(int permits) possible
    upperLimit.acquire() ; mutex.acquire() ; queue.add(item) ; mutex.release() ; lowerLimit.release() ;  }
  public T get() throws InterruptedException {
    lowerLimit.acquire() ; mutex.acquire() ; T item = queue.remove() ; mutex.release() ; upperLimit.release() ; return item ;  }}
```

Lock&Conditions (Monitor mit mehreren Wartelisten für verschiedene Bedingungen)

Äussere Warteliste	Lock Objekt : Sperre für Eintritt in den Monitor (lock() und unlock()
Innere Warteliste	Condition Objekt : Wait&Signal für bestimmte Bedingung (mehrere Conditions pro Lock möglich)

```
Class BoundedBuffer<T> {
  private Queue<T> queue = new LinkedList<>() ;
  private Lock monitor = new ReentrantLock(true) ; // fair
  private Condition nonFull  = monitor.newCondition() ;
  private Contition 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 { monitory.unlock() ; }  } }
```

Read-Write Locks (erlaubt parallele Read-Only-Zugriffe, Mutual Exclusion bei Write-Zugriffen)

Write Lock kann nicht bezogen werden, wenn bereits ein Read Lock bezogen ist

```
class NameDatabase {
  private Collection<String> names = new HashSet<>() ;
  private ReadWriteLock rwLock = new ReentrantReadWriteLock(true) ; // fair
  public Collection<String> find(String pattern) { rw.readLock().lock() ; try {
    for(String name : names) { if (name.matches(pattern) {return true ;} return false ; } // read only access
  } finally { rwLock.readLock().unlock() ; }
  public void put(String name) { rwLock.writeLock().lock() ; try  { names.add(name) ; // write access
  } finally {rwLock.writeLock().unlock() ; } } }
```

Count Down Latch (Eine bestimmte Anzahl Threads warten bis Counter <= 0)

countDown()	Dekrementiert den Zähler. (Blockiert nie)
await()	Warten bis Zähler <= 0 → Blockiert solange Zähler > 0

Latches sind nur einmalig verwendbar. (Da es den Latch nicht bekannt ist, wie viele Threads await() aufrufen. Er kann also nicht entschieden werden, ob alle notwendigen Threads den Latch passiert haben, bevor er wieder geschlossen wird)
CountDownLatch carsReady = new CountDownLatch(N) ; CountDownLatch startSignal = new CountDownLatch(1) ;
carsReady.countDown() ; ----------> carsReady.await() ; startSignal.await() ; <---------- startSignal.countDown() ;

Cyclic Barrier (Warten auf eine fixe Anzahl Threads) → Können wiederverwendet werden. Anzahl Teilnehmer beim Konstruktor

await()	Blockiert bis alle Parties (getParties()) await() auf der Barrier aufgerufen haben. Retourniert fehlende Anzahl Threads

CyclicBarrier raceStart = new CyclicBarrier(N) ; raceStart.await(); (N-Times) raceStart.getParties() ; //Autos fahren direkt los, sobald N da sind !

Phaser (Verallgemeinerte Cyclic Barrier)→ Nachträgliches an- und abmelden von Teilnehmer

Phaser phase = new Phase(0) ; phaser.register() ; phase.arriveAndAwaitAdvance() ; phaser.arriveAndDeregister() ;

Exchanger (Genau 2 Teilnemer (Rendez-Vous) Blockiert bis anderer Thread exchange(x) aufruft. Liefert x des anderen Thread

```
Exchanger<Integer> exchanger = new Exchanger<>() ;
for (int k = 0 ; k< 2 ; k++) { new Thread(() -> {for (int in = 0 ; in < 5 ; in++) {try {int out = exchanger.exchange(in) ;
Sysout(Thread.currentThread().getName() + 'got' + out) ; } catch(InterruptedException ) {} }}}).start() ;}
```

Gefahren der Nebenläufigkeit

Es braucht keine Synchronisation wenn die Objekte unveränderbar (final), daher nur lesend zugegriffen werden oder wenn ein Objekt immer nur einem Thread zur gleichen Zeit gehört. (Confinement)

Thread Confinement	Objekt nur über Referenzen von einem Thread erreichbar
Object Confinement	Objekt in anderem bereits synchronisierten Objekt eingekapselt
Thread Safety	Klassen/Methoden die intern synchronisiert sind und keine Race Conditons aufweisen und die Critical Section nur innerhalb einer Methode ist. (nicht über mehrere Methoden) → moderne Collections sind nicht Thread Safe, deshalb Concurrent Collections verwenden. (Default : Overhead sparen) (Concurrent Collections : BlockingQueue, ConcurrentNavigable Map (TreeMap))

Race Conditions (ungenügend synchronisierte Zugriffe auf Shared Ressources)

Race Condition (semantischer Fehler)	Ein semantischer Fehler, der wegen Timing oder der Reihenfolge von Events auftritt. Ursache ist oft ein Data Race, aber nicht immer. (z.B account.setBalance(account.getBalance() + 100) ; z.B. fehlende Sichtbarkeit von Änderungen (ohne volatile)
Data Race (formaler Fehler)	Wenn zwei Threads auf die gleiche Speicheradresse zugreifen und mindestens ein Thread schreibt. Dabei ist der Zugriff ungenügend synchronisiert. (Data Race auf volatile Variablen ist nicht möglich, Race Conditions hingegen schon)

Deadlocks (Gegenseitiges Aussperren)

Erkennung mit Betriebsmittelgraph (Zyklus im Graphen). Lösung : Lineare Sperrordnung der Ressourcen einführen oder Grobgranulare Locks. Spezialfall Livelocks blockieren sich permanent, verbrauchen aber noch CPU während der Wartenistruktion
4 Bedingungen für Deadlock: Geschachtelte Locks, Zyklische Warteabhängigkeiten, Gegenseitiger Ausschluss, Sperren ohne Timeout/Abbruch

Starvation (Kontinuirliche Fortschrittsbehinderung wegen Fairness Problemen → z.B beim Java Monitor, da nicht fair)

Andere Threads überholen ständig, und schnappen einem Thread die Ressource weg. Folglich verhungert er. (abhängig von Scheduling). ACHTUNG : Bei Thread Prioritäten kann es zur Verdrängung kommen. Lösung : Faire Synchronisation	do { success = account.withdraw(100); } while (!success);
Priority Inversion	Hoch prioritärer Thread wartet auf Bedingung von tief prioritärem Thread → verhungern

Thread Pools (Beschränkte Anzahl von Worker-Threads)

Thread Pool Konzept	- Tasks modellieren potentielle Grad an Parallelität, der Thread Pool realisiert dann den effektiven Grad an Parallelität je nach vorhandenen Systemressourcen - Auszuführende Tasks werden in Warteschlange eingereiht. - Holen Tasks aus der Warteschlange und führen sie aus
Vorteile	- Recycling von Threads (Spare Thread-Erzeugung und Freigabe) - Beschränkte Anzahl von Threads (Viele Threads verlangsamen System oder überschreiten Speicher) - Höhere Abstraktion (Trenne Task-Beschreibung von Task-Ausführung) - Anzahl Threads pro System konfigurierbar (#Worker Threads = #Prozessoren + #I/O-Aufrufe)
Nachteile	Tasks dürfen nicht aufeinander warten → Deadlock Gefahr! Worker Threads laufen als Daemon Threads und laufen evtl. nicht zu Ende
Work-Stealing	Jeder Thread bemüht sich aktiv um Tasks die er ausführen kann. Befinden sich bei Thread1 3 Tasks in der Warteschlange, während Thread2 keine Tasks hat, wird Thread2 Thread1 Tasks «stehlen», damit die globale Auslastung der Prozesse besser wird. Globale FIFO Queue, Mehrere lokale LIFO Queues. **Vorteil:** höhere Effizienz durch weniger Contention (→ Weniger Threads streiten um gleiche Locks) **Nachteil:** Fairnessprobleme bei unausgeglichener Verteilung (LIFO bei SubQueues → wegen Fork/Join)

ForkJoinPool
 	
submit() = async
invoke() = submit + get → sync
	ForkJoinPool threadPool = **new ForkJoinPool();**
	Future<Integer> future = threadPool.submit(() -> { // Future repräsentiert zukünftiges Resultat
	  int value = ...;
	  // long calculation
	  return value; });
	int result = future.get(); //get result from long calculation

Fire and Forget (als Deamon!)	threadPool.submit(() -> { //Task Implementation }); // **Unbehandelte Exception in Task werden ignoriert**

Asynchrone Programmierung RecursiveAction entspricht RecursiveTask<> ohne Rückgabewert

RecursiveTask nur verwenden wenn Ausführungsreihenfolge egal ist!

Expliziter Fork Join Thread Pool

```
ForkJoinPool threadPool = new ForkJoinPool();
Double result = threadPool.invoke(new SumTask());
```

Default Thread Pool (Java 8)

```
ExecutorService threadPool =
ForkJoinPool.commonPool();
Double result = new SumTask().invoke()
```

Asynchrones Zählen

```
Future<Integer> left =
  threadPool.submit( () -> count(leftPart));
Future<Integer> right =
  threadPool.submit( () -> count(rightPart));
result = left.get() + right.get();
```

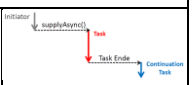
Moderne Async Programming mit Futures:

```
ForkJoinPool.commonPool();
CompletableFuture<Long> future =
CompletableFuture.supplyAsync(() ->
longOperation());
future.get();
```

Rekursives Zählen

```
public static double parallel_sum(double[] array, int from, int to) {
 ForkJoinTask<Double> task = new SumTask(array, from, to);
 return new ForkJoinPool().invoke(task);
}
class SumTask extends RecursiveTask<Double> {
 private double[] array;
 private int from, to;
 public SumTask(double[] a, int f, int t) {
  array = a;from = f; to = t;
 }
 @Overwrite
 protected Double compute() {
  if (to - from >= 1000) {
   int middle = (to + from) / 2;
   SumTask subTask1 = new SumTask (array, from, middle).fork();
   SumTask subTask2 = new SumTask (array, middle, to).fork();
   return subTask2.join() + subTask1.join(); // Reihenfolge wichtig
  } else {
   return linearSum(array, from, to);
  }  }}
```

Fork Join Pool	- fork(): Starte als Sub-Task in einem anderen Task - T join(): Warte auf Task-Ende und frage Resultat ab - T invoke(): Einen Sub-Task starten und synchron abwarten (→ submit(t1).get()) - invokeAll(t1,t2): Mehrere Sub-Tasks starten und abwarten.
Continuation (Folgeaufgabe an asynchrone Aufgabe anhängen)	CompletableFuture<Long> future = CompletableFuture.supplyAsync(() -> longOperation()); future.thenAccept(result -> System.out.println(result)) .thenAccept(): Für Handler ohne Rückgabe .thenApply(..): Für Funktion mit Rückgabe




```
CompletableFuture<Void> analyzeAsync(String website) {
    List<CompletableFuture<Void>> futures = new ArrayList<>();
    for (String link : extractLinks(website)) {
        futures.add(CompletableFuture.runAsync(() -> {
            if (!isReachable(link)) {
                System.out.println(link + "is dead");
            }        }));     }
    return CompletableFuture.allOf(futures.toArray(new CompletableFuture[0]));     }
//Aufruf (Ausführung der Futures): analyzeAsync(...).get();
```

```
CompletableFuture<T> =
CompletableFuture
supplyAsync();

CompletableFuture<Void> =
CompletableFuture
runAsync();
```

Multi Continuation	CompletableFuture.any(future1, future2) .thenAccept(continuation); CompletableFuture.allOf(future1, future2).thenAccept(continuation);

.NET Task Parallel Library

.NET Parallel Programming (fair: FIFO Warteschlange, Best Practice: extra *object* als Lock-Objekt verwenden, statt *this*)

Unterschiede: Thread myThread = new Thread(() => { .. } myThread.Start(); myThread.Join();
Lambda kann umgebende auch schreibend Variablen zugreifen (Prädestiniert für DataRaces)
Wenn eine Exception in einem Thread auftritt, bricht das gesammte Programm ab
lock(obj){...} anstatt synchronized(obj) {...} UND pulseAll() anstatt notifyAll()
Monitor ist fair, es gibt kein Spurious Wakeup, Best Practice ist ein extra object als Lock Objekt verwenden.

.NET Task Parallel Library (TPL) → Einer der modernsten Work Stealing Thread Pool (Hill Climbing Algorithmus)

```
Task<int> task = Task.Run(() => {
 int total = ...
 // some calculation
 return total;   });
task.Result // blockiert bis Task-Ende
```

```
Task task = Task.Run(() => {
  // task implementation
});
// perform other activity
task.Wait(); //blockiert bis Task-Ende
```

```
task1.ContinueWith(task2).ContinueWith(task3); // vgl. Java Completable Future
Task.WhenAll(task1, task2).ContinueWith(continuation); Task.WhenAny(task1, task2).ContinueWith(continuation);
```

Paralleles Statement

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

Paralleler For Each

```
Parallel.ForEach(list,
 file => Convert(file)
);
```

Paralleler For Loop mit Index

```
Parallel.For(0, array.Length,
 i => DoComputation(array[i])
);
```

Parallel LINQ (PLINQ) (LINQ: lazy eval/ PULL mechanism → Auswertung erst durch Itrieren der Abfrage)

```
from book in bookCollection.AsParallel()
 where book.Title.Contains("Concurrency") select book.ISBN
```

```
from number in inputList.AsParallel().AsOrdered()
 select IsPrime(number)
```

GUI und Threading (Single-Threading: nur GUI Thread darf auf GUI Komponenten zugreifen → je nach Framework keine Exception!)

.NET (Verwendung v.a. im UI Layer sinnvoll, Naming Convention: async-Methoden mit "Async" Suffix)

Schlüsselwort async für Methoden → läuft synchron bis zu await, sollte await enthalten sonst Compiler-Warning
Schlüsselwort await für Tasks → ab await = asynchrone Ausführung nach Task-Ende, muss in async Methode stehen

Keine ref oder out Parameter, da async!	Für jedes async muss es ein await geben!

```
async Task<int> GetSiteLengthAsync(string url) {
  HttpClient client = new HttpClient();
  Task<string> task = client.GetStringAsync(url);
  string site1 = await task;
  return site1.Length; }
```

```
public async Task<bool> IsPrimeAsync(long number) {
 return await Task.Run(() => {
  for (long i = 2; i <= Math.Sqrt(number); i++) {
   if (number % i == 0) { return false; }  }
  return true;   });  }
```

Java (Java UI Thread = Event Dispatching Thread) → Swing: pack() & setVisible() im UI Thread ausführen!

UI Komponenten Zugriffe an UI Thread delegieren: Klasse SwingUtilities → **invokeLater()** asynchon, **invokeAndWait()** synchron.
Single-Thread Modell im UI wegen Deadlock Risiko zwischen Parent und Child und zu teurer Synchronisation.

```
blau = workerthread, grün = UI Thread;

button.addActionListener(event -> {
 ForkJoinPool.commonPool().submit(() -> {
 // OR CompletableFuture.runAsync(() -> {..});
   String text = readHugeFile();
   SwingUtilities.invokeLater(() -> {
     textArea.setText(text);
     //allfällige weitere Thread-Starts hier
 }); }) });
```

```
class BackgroundCalculator extends SwingWorker<Integer, Void> {
@Override
public Integer doInBackground() {
  return longComputation();
}

@Override
protected void done() {
  try {
    Integer result = get();
    label.setText("Result: " + result);
  } catch (InterruptedException | ExecutionException e) {
    …
  }
}
```

Resultat von Background
Keine Zwischen-Resultate
Separater Thread
Resultat von doInBackground()
UI Thread

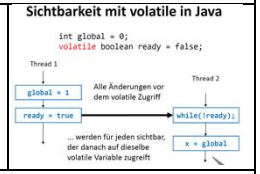
Memory Models (Lock freie Datenstrukturen: Ermöglicht effiziente Synchronisation. Problem Compiler optimiert, ordnet um)

Volatile Keyword	Volatile verhindert Data Race auf Variable. Änderungen werden anderen Zugreifenden propagiert. Keine Umordnung durch Compiler. Atomares Lesen & Schreiben auch für long und double
Atomicity (Unteilbarkeit)	Zugriff auf Variable (Lesen/Schreiben) ist atomar für - primitive Datentypen bis 32 Bit - Objekt Referenzen - long und double nur mit volatile Keyword atomar
Visibility (Sichtbarkeit)	- Sehen Änderungen eines anderen Threads eventuell gar nicht oder erst später - Optimierung, z.B. hält VM Variablenwert in Register

Sichtbarkeit ist garantiert bei:
- Locks Release & Acquire (Änderungen vor Release werden bei Acquire sichtbar)
- Volatile Variable (Zugriff macht Änderungen anderen Zugreifern sichtbar)
- Initialisierung von final Variablen (Nach Ende des Konstruktors
- Thread-Start und Join (ebenso Task Start und Ende)

Volatile Sichtbarkeit:
Alle Änderungen vor dem Zugriff auf die volatile Variable werden für alle Threads sichtbar, die danach auf diese volatile Variable zugreifen.Lese und Schreibzugriff invalidiert den Hauptspeicher (Memory Flush)

Sichtbarkeit mit volatile in Java



Ordering	Innerhalb des Thread = seriell Zwischen zwei Threads = Umordnungen durch Compiler (ausser volatile)

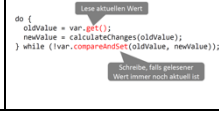
Atomare Objekte (Kein Blockieren oder Warten auf Locks. Garantieren auch Ordering und Visiblity)

AtomicReference<Object>, AtomicInteger, AtomicBoolean

Spin-Lock mit atomarer Operation

Compare and Set

Optimistische Synchronisation

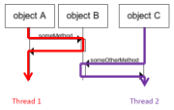



```
boolean success = false;
do {  int value = count.get();
if (value > 0) { success = count.compareAndSet(value, value - 1); }
} while (!success);
```

.Net Memory Model (Unterschiede zu Java)	- Atomarität: long/double nicht mit volatile atomar - Visibility: Nicht definiert. Implizit durch Ordering - Ordering: volatile ist nur partielle fence - Atomare Instruktionen mittels Interlocked Klasse

Actor Model (Kein Shared Memory, Keine Race Conditions, async über Messages)

Motivation: Threads operieren normalerweise auf Modell von passiven Objekten (Objekte und Threads haben nichts miteinander zu tun.

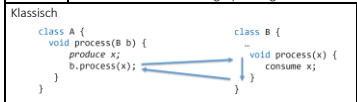


Im Actor Model sind die Objekte (Actors) aktiv

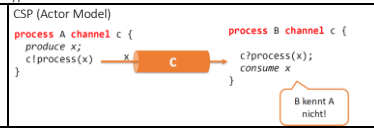
- Kommunikation über empfangen/senden von serialisierten immutable Messages
- Kein Shared Memory, nur lokalen isolierten Speicher

Vorteile	- Inhärente Nebenläufigkeit (Alle Objekte laufen nebenläufig) - Keine Data Races (Kein Shared Memory) - Gute Verteilbarkeit (Kein Shared Memory, so fürs Netz prädestiniert)
Akka	Achtung: Actoren dürfen keine Referenzen auf gemeinsame Heap Objekte verwenden (nur mit ActorRef und Kommunikation via Messages) → Es gibt auch keine Typensicherheit

Klassisch



CSP (Actor Model)



Cluster Parallelisierung (Viele verteilte CPU, HPC : High Performance Computing, MPI : Message Passing Interface)

Zwischen den Nodes gibt es kein Shared Memory → MPI als verteiltes Programmiermodell (basiert auf Actor/CSP Prinzip)

HPC Cluster : mpiexec -n <anz. instances> Program.exe

MPI Programm wird in mehrere Prozesse gestartet. Jeder Prozess hat eine Idenfikation (Rank) innerhalb einer Gruppe.
Die Prozesse arbeiten unabhängig in ihrem Adressraum. Alle Prozesse starten und terminieren synchron. Prozesse können mit Messages kommunizieren. (innerhalb **Communicator.world**).

send(x,y,z)	x: data to transmitt, y: rank of destination, z: tag to distinguish msg type (only one type: e.g. use 0)
T Allreduce(T value, Op<T>)	Jeder erhält das Gesamtresultat als Rückgabewert, Implizite Barriere/Broadcast über alle Prozesse world.Allreduce(value, (a, b) => a + b)
T Reduce(T v, Op<T>, int rank)	Nur ein Prozess (rank) sieht das Gesamtresultat, Effizienter als Allreduce, weil kein Broadcast

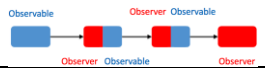
```
using MPI ; using System ;
class Programm { public static void Main(string[] args) {
  using (new MPI.Environment(ref args)) {
    var world = Communicator.world ; int rank = world.Rank ; int size = world.Size ;
    if (rank == 0) {world.Send(value, to_N, tag) ; } else { int value ; world.Receive(from_0, tag, out value) ;}
  } }
```

Reactive Programming

LINQ = Pull-Mechanismus: Pipeline-Schritte werden rückwärts ausgelöst. Problem Aktive Quellen → Input mit Pausen oder unbekannt lange Streams) → **DESHALB**: Reactive Programming mit PUSH Mechanismus. Input lösen pro Wert einen Ereignis aus (OnNext). Nachfolgeschritt abonniert Events des Vorgängers (**async**)

Asynchrone Datenflüsse, rein deskriptiv. - Automatische Parallelisierung - Verteilbarkeit - Keine Data Races/Deadlocks/Starvation, kein Shared Memory

Rx.NET

+ Aktive Datenflüsse statt nur passive LINQ + Skaliebare Parallelität durch Wahl der Scheduler + Durchgängig asynchron	- Zerstückelung komplexer Logiken in Handler - Allfälliger Kontext muss durchgeschleust werden - Komplizierte Aggregation (Observable statt Skalar)
Rx Observer Pattern IObservable<T> → Subscribe(Observer<T>) IObserver<T> → OnNext(T), OnError(Exception), OnCompleted()	Rx Pipelining 

Defaultmässig alles sequentiell, aber asynchron → concurency erst mit ObserveOn()	var sales1 = salesEurope.ToObservable(). ObserveOn(TaskPoolScheduler.Default);
Ausführung im GUI Thread	DispatcherScheduler.Current combinedSales.ObserveOnDispatcher().Subscribe(e => textBlock.Text += e.Nr + " " + e.Size);

```
var subject = new Subject<string>();
subject.Subscribe(
   val => Console.WriteLine(val),
   err => Console.WriteLine(err.Message),
   () => Console.WriteLine("Complete")
);
subject.OnNext("Test");
subject.OnCompleted();      Console.ReadKey();
```

```
var sales = Sales.salesEurope.ToObservable().
ObserveOn(TaskPoolScheduler.Default).
Merge(Sales.salesAsia.ToObservable().
ObserveOn(TaskPoolScheduler.Default));

sales.ObserveOn(DispatcherScheduler.Current).
Subscribe(item => textBlock.Text += item.Article + " " + item.Volume
+ Environment.NewLine);
```

Observables

Hot = Aktiv Notifizieren spontan, auch ohne registrierte Observers Observable.Interval(…) Observable.Timer(…) / .FromEventPattern(…)	Cold = Passiv Notifizieren on Request, erst bei Anmeldung von Observers collection.ToObservable(…) Observe.Range(…) / .Generate(…)

Software Transactional Memory (STM)

Prinzip : deskriptive Spracherweiterung, sodass keine Data Races, Deadlocks. Definieren von atomic Blöcken, korrekte Ausführung dem System überlassen. Nach ACI aber ohne D(urability)/Persistenz. Eignet sich nicht für die Verteilung
Optimistic Concurrency Control(OCC) : Methode für Transaktionen, Annahme dass keine Störungen zwischen 2 Transaktionen passieren, vor Übermittlung auf Konflikt prüfen. Bei Konflikt wird vom System ein Rollback durchgeführt -> keine Verwendung von Locks. Phasen : **Begin->Modify->Validate->Commit/Rollback**. Bei Interrupt folgt retry. Achtung : STM muss HW unterstützen.

```
Scala STM mit Transkationsretry:
final Ref.View<Integer> balance = STM.newRef(0) ; final Ref.View<LocalDate> lastUpdate = STM.newRef(LocalDate.now()) ;
void withdraw(int amount) {
STM.atomic(() -> {
    if (balance.get() < amount) {
        STM.retry();       }
    balance.set(balance.get() - amount);
    lastUpdate.set(LocalDate.now());       });}
```

Schwächen :

Starvation	Falls Transaktion wiederholt wegen Konflikt abbricht könnte eine Transaktion im Zyklus verhungern
Seiteneffekte	In einer atomic Operation dürfen keine Seiteneffekte auftauchen. Generell kein IO(Konsole, Files, Netz, Thread Start/Wait/Join). Auch keine normalen Variablen, nur final oder lokale Vars lesen, ansonsten Ref.View<T> Zugriffe mit T immutable.
Write Skew (In Scala STM nicht möglich, andere Systeme schon)	Wenn zwei Transaktionen zwar konfliktfreie Updates machen aber damit eine mit den beiden Werten verbundene Condition invalidiert wird. Beispiel : 2 Bankkonten, 1 darf negativ aber beide zusammen nie unter 0. Snapshot wird gemacht-> beide Konti $100. Gleichzeitige Transaktion von $200 abheben funktioniert, weil zum Zeitpunkt von Snapshot gülitg(geschrieben wird nur veränderter Wert).
Ref-Wrapper	Nur Zugriffe auf Ref.View<T> Wrapper mit T immutable, vordefinierte transaktionelle Collections(STM.newMap()/newSet()/newArrayAsList()) normale List fehlt

GPU : Physische parallelisierung von Operationen auf Vektoren und Matrizen (8*16=128 / 16*16=256 / 16*32=512 / 32*32=1024)

Verleich GPU	Vergleich CPU
Extrem hohe Datenparallelität	Niedrige Datenparallelität
Wenig Verzweigunen	Viel Verzweigungen
Kein beliebiges Warten bei Parallelität	Beliebige Thread-Synchronisation
Einfache, aber viele Cores	Wenige, aber mächte Cores
Kleine Caches pro Core	Grössere Caches per Chip
Aufbau GPU (Physisch)	**Variablen (je Bedarf mit *x,y,z* Dimensionen, hier nur 1D, *x*):**
Streaming Multiprocessor (hat Global Memory) Grid (Beinhaltet meist 512, 1024, 2048,… Blocks)	
Block (je 4 Warps / 128 Threads) (hat Shared Memory)	*blockDim.x* : Logische Blockgrösse *blockIdx.x* : Block ID (0 bis Anzahl mögliche Blocks)
Warp (32 Threads)	Alle Threads in Warp führen gleiche Instruktion aus (SIMD)
Thread (führt einen Kernel aus) (hat Register-Speicher)	*threadIdx.x* : Thread ID innerhalb Block (0 bis (blockDim -1))
Shared Memory	**Global Memory**
Per streaming Multiprozessor Zugriff : *__shared__ float x ;* Schnell ca. 4 Zyklen per Zugriff Nur zwischen Threads innerhalb Block sichtbar Paar KB	« Main Memory » des GPU Device Zugriff : *cudaMalloc()* Langsam, ca. 400-600 Zyklen per Zugriff In allen Threads sichtbar Mehrere GB

SIMD (Vektor-Parallelität): Alle Threads führen denselben Kernel-Code aus. Die Ausführung unterscheidet sich nur in den Werten von threadIdx und blockIdx. Zu einem Zeitpunkt führen alle Threads (konkret in einem Warp) dieselbe Instruktion auf (typischerweise) verschiedenen Daten aus. Bei Verzweigungen werden die Instruktionen der Pfade abwechselnd ausgeführt. Threads ignorieren dabei die Instruktionen eines anderen Pfades.

Divergenz: gibt es Threads innerhalb eines Warps, die unterschiedliche Dinge tun? (logische if,else verzweigung) → schlecht!

Memory Coalescing: Es sollte immer in 128 Byte Blöcken (oder vielfachen) in %128Byte Abständen auf den Speicher zugegriffen werden, sonst braucht es zwei Memory Zyklen. → gut!

CUDA-Barriere *__syncThreads() ;* // synchronisiert alle Threads innerhalb eines Blocks, damit alle Threads gleichzeitig durch diesen Punkt gehen. Achtung : Falls in if/else Block, sind if oder else case zwei unterschiedliche Barrieren!

Mehrdimensionale Funktionen mit C: z.B gridSize(x,y,z)

```
float *matrix = (float *)malloc(NofRows * NofCols * sizeof(float)); // Zugriff mit: matrix[row * NofCols + col] = …;
dim3 gridSize(3,2); dim3 blockSize(4,3);
Call<<<gridSize, blockSize>>>(…);
```

N	Breite * Höhe
gridSize = BlocksPerGrid	G = (N + blockSize – 1) / blocksize → 1D [Aufrunden von N/blocksize]
blockSize = ThreadsPerBlock	Vielfaches von 32. Abhängig von GPU: z.B 1D: 1024 oder 2D: (32,32) → Möglichst gross damit die Threads den selben Speicher haben
dim3 blocksPerGrid((C_COLS + TILE_SIZE - 1) / TILE_SIZE, (C_ROWS + TILE_SIZE - 1) / TILE_SIZE);	dim3 threadsPerBlock(TILE_SIZE, TILE_SIZE);
insgesammt Laufende Threadzahl	gridsize (bei dim3 beide zahlen *) * blocksize
insgesammt benötigte Threadzahl	dim der Zielmatrix: C_Cols*C_Rows
Max. Concurrent Thread	Anz. Stream Multiprocessor (SM) * Maximum Resident Threads per SM
Max. Anz.Threads im Bezug auf Max. Resident Blocks	Anz. Stream Multiprocessor (SM) * Max. Resident Blocks per SM * blockSize

Einfacher Kernel für Vektoradition *A + B = C* ***gridSize*blockSize=Anz. CUDA Threads***

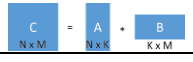
```
__global__ // kernel
void VectorAddKernel(float *A, float *B, float *C, int N) {
  int i = blockIdx.x * blockDim.x + threadIdx.x ;  // unique id
  if (i < N) {// Überflüssige Threads machen nichts
    C[i] = A[i] + B[i] ;
} }
```

```
void cuError(cudaError error) {
  if (error != cudaSuccess) {
    fprint(stderr, "CUDA : %s !\n",
      cudaGetErrorString(error) ) ;
    exit(EXIT_FAILURE) ;
} }
```

```
// Alternatives main(){} : unified Memory
cudaMallocManaged(&A, size) ;
cudaMallocManaged(&B, size) ;
cudaMallocManaged(&C, size) ;
VectorAddKernel<<<…>>>(A, B, C, N) ;
cudaDeviceSynchronize() ;
cudaFree(A) ; cudaFree(B) ; cudaFree(C) ;
```

```
// main(){} implementation
cuError(cudaMalloc(&d_A, size)) ; // 1. alloc on GPU
cuError(cudaMalloc(&d_B, size)) ;
cuError(cudaMalloc(&d_C, size)) ;
// 2. Transfer from CPU to GPU
cuError(cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice));
cuError(cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice));

int blockSize = 512 ; // Vielfaches 32 für Warps
 // sollten so gross wie möglich aber zu weniger verlust führen
int gridSize = (N + blockSize -1) / blockSize
// 3. execute kernel
VectorAddKernel<<<gridSize, blockSize>>>(d_A, d_B, d_C, N) ;
cuError(cudaGetLastError()) ;
// 4. Transfer from GPU to CPU
cuError(cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost));
// 5. Deallocate GPU
cuError(cudaFree(d_A)) ;
cuError(cudaFree(d_B)) ;
cuError(cudaFree(d_C)) ;
```



Beispiele

Invoke Later mit For Loop und Done Ausgabe

```
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);   }); }); }}
```

Array Rotieren mit Thread Pool (List of Futures)

```
int[] input = array;    int[] output = new int[input.length];
ForkJoinPool pool = new ForkJoinPool();
List<Future<?>> futures = new ArrayList<>();
for (int index = 0; index < input.length; index++) {
    int position = index;
    futures.add(pool.submit(() ->
    output[(position + 1) % input.length] = input[position]   ));    }
for (Future<?> future : futures) {
    future.get();     }
array = output;
```

Semaphore as Monitor

```
public class Semaphore {
  private int counter = 0; private int waiting = 0;
  public synchronized void acquire(int amount) throws InterEx {
    waiting++ try {
      while (counter < amount) { wait(); }
      counter -= amount;
    } finally { waiting--; }}
  public synchronized void release(int amount) {
    counter += amount; notifyAll(); }
  public synchronized int nofWaitingThreads() { return waiting; }
}
```

Cyclic Barrier with Semaphore

```
public class CyclicBarrier {
  private final int parties; private int count = 0; private Semaphore open;
  private Semaphore closed; private Semaphore mutex;
  public CyclicBarrier(int parties) { this.parties = parties; open = new
Semaphore(parties); closed = new Semaphore(0);mutex = new Semaphore(1);
  }
  public void await() throws InterruptedException {
    open.acquire(); mutex.acquire(); count++;
    if (count == parties) { closed.release(parties); }
    mutex.release(); closed.acquire(); mutex.acquire(); count--;
    if (count == 0) { open.release(parties); } mutex.release(); }
```

AtomicReference / CompareAndSet

```
public class Mailbox {
private AtomicReference<Object> item = new AtomicReference<>();
  public void put(Object value) {
    while (!item.compareAndSet(null, value)) { Thread.yield(); }
  }
  public Object get() { while (true) { Object value = item.get();
    if (value != null && item.compareAndSet(value, null)) {
      return value; } Thread.yield();  } }
}
```

Cyclic Barrier with Monitor

```
public class CyclicBarrier {
  private final int parties; private int entered = 0; private int exited = 0;
  public CyclicBarrier(int parties) { this.parties = parties; }
  public synchronized void await() throws InterruptedException {
    entered++; if (entered == parties) { exited = 0; notifyAll(); }
    while (entered < parties) { wait(); } exited--;
    if (exited == parties) { entered = 0; notifyAll(); }
    while (exited < parties) { wait(); }
  }}
```

Upgradable RW/Lock

```
public class UpgradeableReadWriteLock {
  private int readCounter = 0;  private Thread upgradeThread = null;
  private boolean writeLocked = false;
  public synchronized void readLock() throws InterruptedException {
    while (writeLocked) { wait(); } readCounter++; }
  public synchronized void readUnlock() { readCounter--;
    if (readCounter == 0) { notifyAll(); } }
  public synchronized void upgradeableReadLock() throws IntEx {
    while (writeLocked || upgradeThread != null) { wait(); }
      upgradeThread = Thread.currentThread(); }
  public synchronized void upgradeableReadUnlock() {
      upgradeThread = null; notifyAll(); }
  public synchronized void writeLock() throws IntEx {
    while (writeLocked || upgradeThread != null &&
Thread.currentThread() != upgradeThread || readCounter > 0) {
        wait(); } writeLocked = true; }
  public synchronized void writeUnlock() {writeLocked = false;
      notifyAll(); } }
```

Broadcast / Exchange mit Semaphore

```
public class Broadcast<T> { private final int amount; private T exchange;
private final Semaphore sender = new Semaphore(0); private final Semaphore
receivers = new Semaphore(0);
public Broadcast(int amount) { this.amount = amount; }
public void send(T item) throws InterruptedException { exchange = item;
sender.release(amount); receivers.acquire(amount);
}
public T receive() throws InterruptedException {
  sender.acquire(); receivers.release(); return exchange; }
}
```

Acquire All mit Starvation

```
void acquireAll(Lock[] lockSet) { int i = 0; while (i < lockSet.length) {
  if (lockSet[i].tryLock()) { i++; } else {
    while (i > 0) { i--; lockSet[i].unlock(); } } } }
```

Linear Sperrordnung (Min / Max)

```
table.acquireFork(Math.min(table.rightForkNumber(id),
table.leftForkNumber(id)));
```

Lock Free Stack mit Atomic Reference

```
public class LockFreeStack<T> {
private AtomicReference<Node<T>> top = new AtomicReference<>();
  public void push(T value) {
    top.getAndUpdate(topNode -> new Node<>(value, topNode));}
  public T pop() { return top.getAndUpdate(topNode -> {
    return top.get() == null ? null : topNode.getNext(); }).getValue();
  } }
```

MPI: Multiple Sender, One Receiver

```
if (world.Rank > 0) { world.Send(count, 0, 0) } else {
  long total = count;
  for (int other = 1; other < world.Size; other++) {
    long value; world.Receive(other, 0, out value); total += value; }
  Console.WriteLine("Amount of primes " + total);
}
```

Thread Pool: Tree Traversal

```
class TreeTraversal extends RecursiveAction {
  private final Node node;
  public TreeTraversal(Node node) { this.node = node; }
  @Override
  protected void compute() {
    if (node != null) {
      TreeTraversal leftTask = new TreeTraversal(node.getLeft());
      TreeTraversal rightTask = new TreeTraversal(node.getRight());
      leftTask.fork(); rightTask.fork(); Logic.operation(node.getValue());
      rightTask.join(); leftTask.join(); } } }
```

R/W Lock mit Semaphore:

```
public class ReadersWriterLock {
  private int readers = 0;
  private final Semaphore mutex = new Semaphore(1);
  private final Semaphore writer = new Semaphore(1);
  public void acquireReadLock() throws InterruptedException {
    mutex.acquire(); try { if (readers == 0) { writer.acquire(); } readers++;
    } finally { mutex.release(); } }
  public void releaseReadLock() throws InterruptedException {
    mutex.acquire(); try { readers--; if (readers == 0) { writer.release(); }
    readers++; } finally { mutex.release(); } }
  public void acquireWriteLock() throws InterEx { writer.acquire();}
  public void releaseWriteLock() { writer.release(); } }
```

Thread Pool: Recursive Task mit invokeAll

```
@Override
protected void compute() {
  if (end - start > THRESHOLD) {
    int middle = (start + end) / 2;
    invokeAll(
      new ReversionTask(array, start, middle),
      new ReversionTask(array, middle, end)
    );
  } else {  for (int left = start; left < end; left++) {
      int right = array.length - 1 - left;  int temp = array[left];
      array[left] = array[right];array[right] = temp;  }}}
```

GPU: Synchtreads Barriere

```
__global__
void processing(int* array, int length) {
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  int value; if (index < length) {
      value = array[index];
  }
  __syncthreads();
  if (index < length) {
  array[(index + 1) % length] = value;
  }
}
```

GPU: Image Processing

```
__global__
void imageProcessing(float *pixels) {
  int row = threadIdx.x; int col = threadIdx.y; // bad
  if (row < IMAGE_HEIGHT && col < IMAGE_WIDTH) {
    int pos = row * IMAGE_WIDTH + col;
    if (pixels[pos] < 0.5) { pixels[pos] = 0; } else { pixels[pos] = 1; } }
}
```

GPU: Array Reversion

```
__global__
void revert(int* array, int length) {
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  if (index < length / 2) { int right = length – 1 - i;
    int temp = array[left]; array[left] = array[right]; array[right] = temp; }
}
```

Prozess 1: Array gemäss Anzahl Prozesse segmentieren.
Prozess 1 verteilt Segment an anderen Prozesse.
Jeder Prozess verarbeitet sein Segment in Isolation.
Prozess 2 bis N-1 schicken ihre Lösung zurück an Prozess 1.
Prozess 1: kombiniert die Segmente und arrangiert sie.

Monte Carlo Simulation :

```
int rank =  world.Rank ; int size =  world.Size ;
long hits =  CountHits ( Trials/size ) ;
long totalHits = world.Reduce (hits,    (a, b)  => a  +  b,  0 ) ;
if   ( rank ==    0)   {
      double pi = 4   *   ( ( double ) totalHits    /    Trials);
}
```