

MULTI-THREADING

Informatik Workshop
Master Studiengang
Wintersemester 2009/2010

18.02.2010

Pascal Hahn, Marcus Körner, Alexander Mezler, Johannes Wachter

Agenda

- Einführung und Motivation
- Wieso ist Multi-Threading so kompliziert?
- Bekannte Methodiken
- Automatisierung von Multi-Threading
- Umsetzungen in Programmiersprachen
 - Concurrent Java Programming
 - Multi-Threading mit C#
 - Actor-Modell in Scala
- Ausblick und Fazit

EINFÜHRUNG

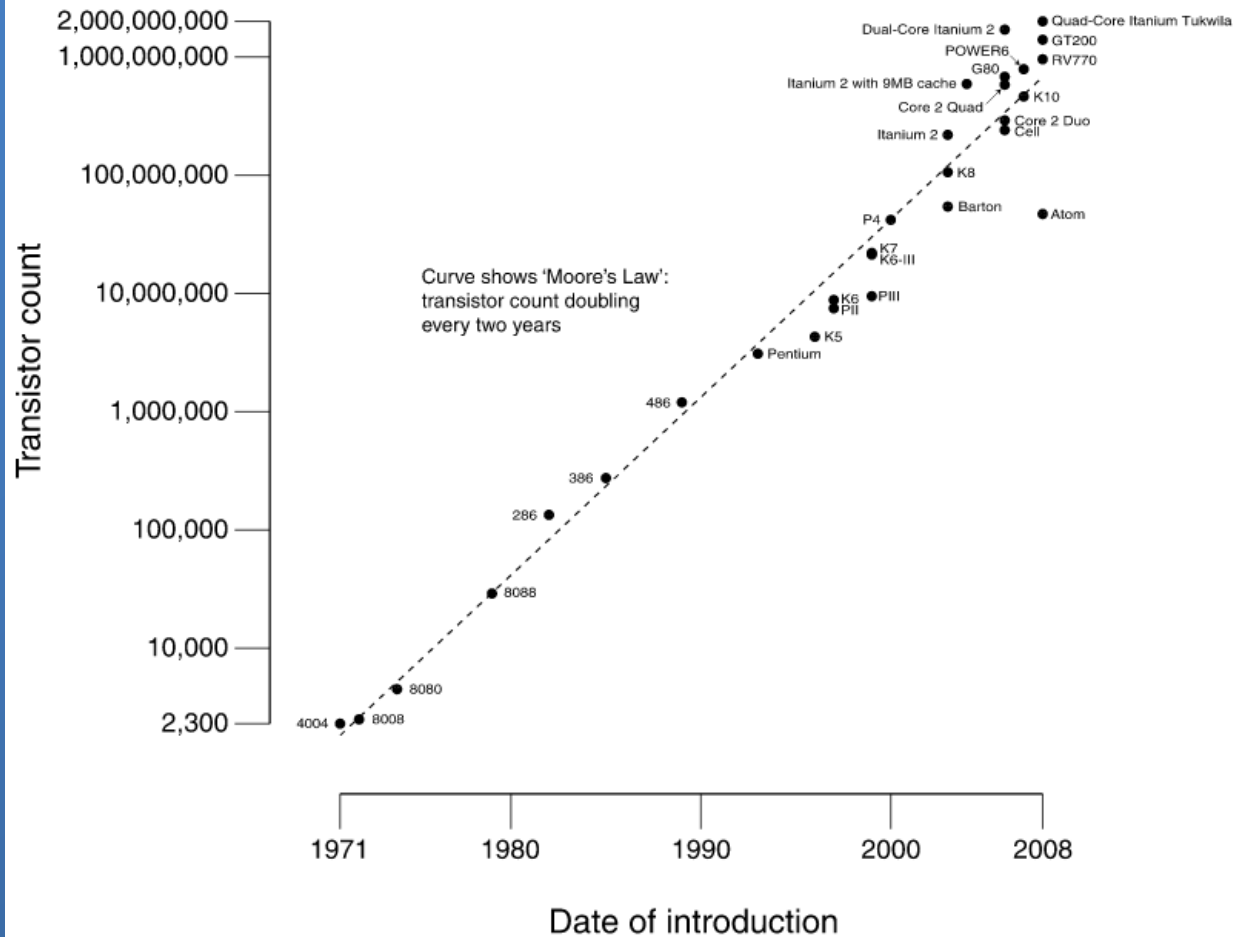
WAS IST MULTI-THREADING?

Multi-Threading ist die Eigenschaft der CPU mehrere Threads scheinbar zur gleichen Zeit zu verarbeiten.

Relevanz

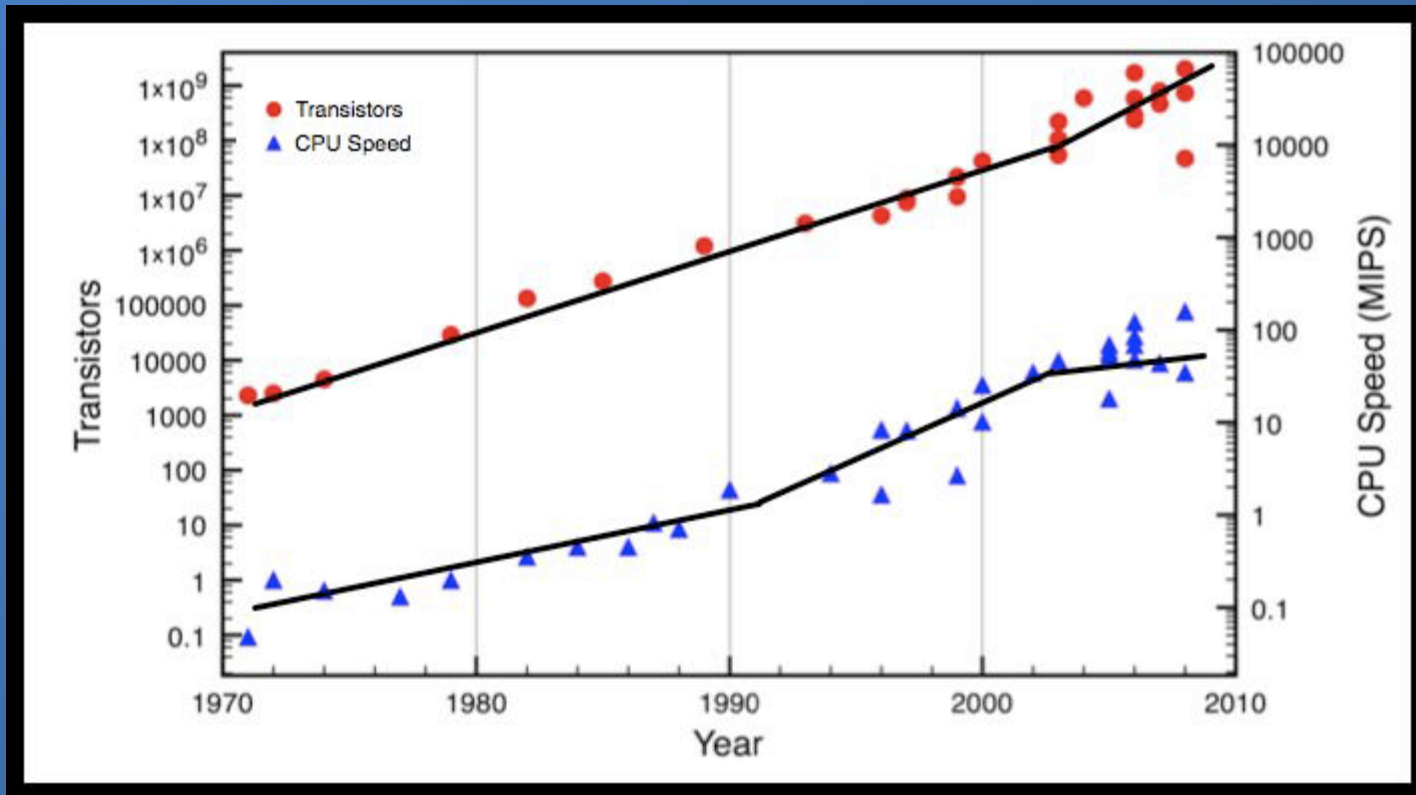
- Moores' Law hat sich geändert!
 - „Anzahl Transistoren verdoppeln sich alle 18 Monate“ (wurde sogar übertroffen)
 - Aber höhere Taktraten kaum möglich!
 - Etwa um Jahr 2000: physikalisches Limit erreicht
- Entwicklungen der letzten Jahre hin zu
 - Multi-Core-Systemen
 - System-on-a-Chip (z.B. Pineview, Sandy Bridge)
- Kommende Entwicklungen
 - Vielkern-Prozessoren
 - Singlechip Cloud Computer (SCC)

CPU Transistor Counts 1971-2008 & Moore's Law



Gesetz von Moore bezüglich Intel Prozessoren (Stand 2008)

http://upload.wikimedia.org/wikipedia/commons/0/00/Transistor_Count_and_Moore's_Law_-_2008.svg



Transistoren und MIPS (Millionen Instruktionen per Sekunde) Entwicklung

<http://www.javaworld.com/javaworld/jw-02-2009/images/actors1-graph.jpg>

Warum Multi-Threading?

- Hauptziele
 - Steigerung der Performance
- Grundlegend
 - Viele Probleme sind zerlegbar in kleine unabhängige Probleme
 - > sequentielle Verarbeitung nicht sinnvoll!
- Langfristig
 - (Endlose) Skalierbarkeit
 - Effizientere Nutzung von Ressourcen

Trend zu Multi-Core-Systemen

- Auf dem Desktop
 - P4 mit Hyper-Threading
 - Core 2 Duo
 - Core 2 Quad
 - i7, i5, i3
 - AMD Athlon X2
 - AMD Phenom
 - AMD Athlon II
- Aber auch im Serverbereich
 - Sun UltraSPARC T1 (Niagara) mit 4x8 Threads
 - Sun UltraSPARC T2 mit 8x8 Threads



Trend zu Multi-Core-Systemen (2)

- Playstation 3
 - Cell (1x 3.2 Ghz PPE + 8x SPEs)
- Xbox 360
 - Triple-Core Xenon von IBM
- Intel Larrabee
- Intel Terascale (Mehrkern mit hunderten Kernen)
 - Erreicht Leistung des ASCI-Red-Supercomputers (von 1996 – Bestand aus 10.000 Pentium-Pro 200Mhz)



Massive Parallelisierung

- Stream processing (FPGAs, GPUs)
 - Grundidee, Gegeben Satz an Daten und Kernel
 - Kernel wird auf jedes Datenelement angewendet
 - CUDA
 - OpenCL
 - Intel Ct (Throughput Computing)
- Folding@home, Cluster
- Zweck:
 - Nutzung sämtlicher verfügbarer Rechenressourcen
 - Nutzung der Grafikkarte, da das Potential sonst nur in Spielen genutzt wird

Wofür parallel programmieren?

- Desktop-Systeme
 - Leistung eigentlich ausreichend
 - High-Performance Anwendungen
- Wissenschaftliche und mathematische Berechnungen
- Spiele oder andere stark technisch anspruchsvolle Software
 - Bsp. Medizinische Bildverarbeitung

BEISPIELE

Für parallelisierbare Probleme

Parallelisierbare Probleme

- Terminplanung
 - Berechnung von freien Ressourcen → Ressourcen Matrix
 - Ressourcen sind z.b. Personen, Räume, Präsentationsmittel, Hardware, Zeitslots
 - In dem Bereich gibt es eine Firma die momentan diese Überschneidungen auf CUDA Basis berechnet.
 - Bsp: 40 Personen, 1 Beamer, 10 Räume, 3 Tage
 - Wann könnte der Termin stattfinden? (ohne Überschneidung)



Id tech 5

Next-Gen Game-Engine von Id Software

id tech 5 - Herausforderung

- Jede Menge Berechnungen
 - Animation Blending ~2 ms
 - Collision Detection ~4 ms
 - Obstacle Avoidance ~4 ms
 - Transparency Sorting ~2 ms
 - Virtual Texturing ~8 ms
 - Misc Processing ~4 ms
 - Rendering ~10 ms
 - Audio ~4 ms
- Sequentiell: 38 ms
- Anforderung 60Hz ~ 60 FPS -> 1000 ms / 60
 - 16 ms Zeit!

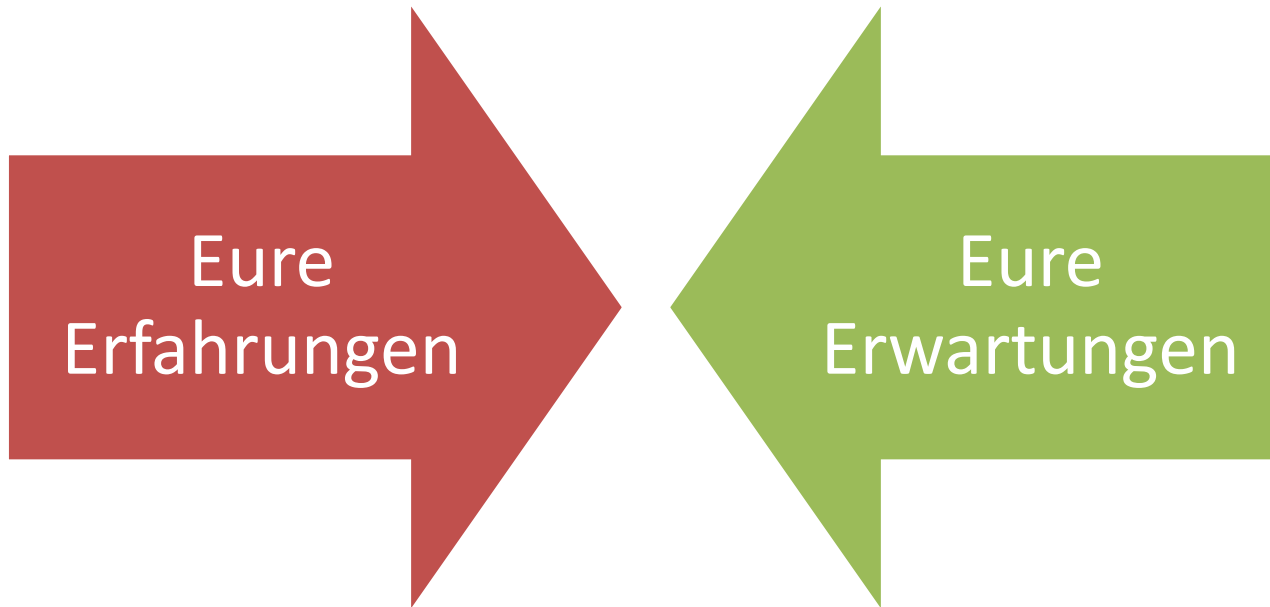
id tech 5 - Lösung

- Cell Prozessor bringt neue Anforderung:
Portierbare parallele Softwarearchitektur
 - Refactoring der Engine in Jobs
 - Verteilen der Jobs auf allen Cores
- Engine muss alle verfügbare Parallelität ausnutzen
 - Support für CUDA, OpenCL, Larrabee geplant

MULTITHREADING

Wieso ist Multithreading so kompliziert?

Nachgefragt



Einleitung

- Wieso ist Multithreading so kompliziert?
 - Deutlich höhere Komplexität
 - Menschliche Faktoren
 - Kein standardisiertes Vorgehen
 - Fehlendes Wissen
 - Wenig Tool-Support und Probleme beim Testen
 - Komplizierte Laufzeit-Analyse und Wartung

Komplexität

- Programmablauf schlecht nachvollziehbar
 - Reproduzierbarkeit
 - Bugs
 - Programmcode schwer verständlich
- Multithreading erfordert neue Architekturen und Designs
- Finden eines optimalen Algorithmus schwieriger
 - Welche Standardalgorithmen gibt es?

Menschliche Faktoren

- Menschen denken sequenziell
 - Insbesondere bei Konzentration auf 1 Thema
 - Aber: Gehirn arbeitet parallel
- Geeignete Abstraktionsmechanismen fehlen
 - Parallelismus wird unsichtbar
 - Neuartige Konzepte und neue Denkweisen wie wir Programme entwickeln

Keine standardisiertes Vorgehen

- Multithreading-Boom durch Consumer-PCs
 - Software hinkt der Hardware-Industrie hinterher
- Jede Programmiersprache hat eigene Konzepte
 - Mainstream-Programmiersprachen haben oftmals schlechte Unterstützung
 - Modernere Sprachen bieten viele neue Konzepte an
- Was wird sich durchsetzen?
 - Langzeiterfahrungen fehlen

Multithreading im Studium

- Sequenzielles Programmieren steht im Vordergrund
- Neuartige Konzepte in Java nicht enthalten
- Nutzen von Multithreading verdeutlichen
- Was kann man tun?
 - Multithreading als Grundwissen betrachten
 - Parallele und sequenzielle Algorithmen auf eine Stufe stellen
 - Architektur von Programmen die Multithreading nutzen vermitteln

Tool-Support und Testen

- Tools für sequenziellen Ablauf entworfen
- Visual Studio Debugger
 - Thread wechseln, Thread einfrieren, Threads als unwichtig markieren
- Intel Vtune Performance Analyzer
 - Shared Memory Konflikte erkennen
 - HotSpots in einzelnen Threads erkennen
- Normale Unit-Tests reichen nicht aus
 - Oft keine Context-Switches in einem Unit-Test
 - ConTest von IBM für Java

Laufzeitverhalten und Wartung

- Laufzeitverhalten ermitteln
 - Wie skaliert das Programm?
 - Wie hoch ist der sequenzielle Anteil?
- Lohnt es sich überhaupt?
 - Abschätzung Performance / Aufwand
 - Wartung wird komplizierter
 - Ist Multithreading-Erfahrung im Team vorhanden?

Schlussfolgerung

- Multithreading wird nicht verschwinden
 - Grundwissen eines jeden Informatikers
- Lösung nicht unbedingt in naher Zukunft
 - Möglicherweise radikal neue Denkweisen und Programmiersprachen nötig
 - Compiler übernehmen die Parallelisierung
- Bessere Tool-Unterstützung notwendig
- Standardisierte parallele Algorithmen

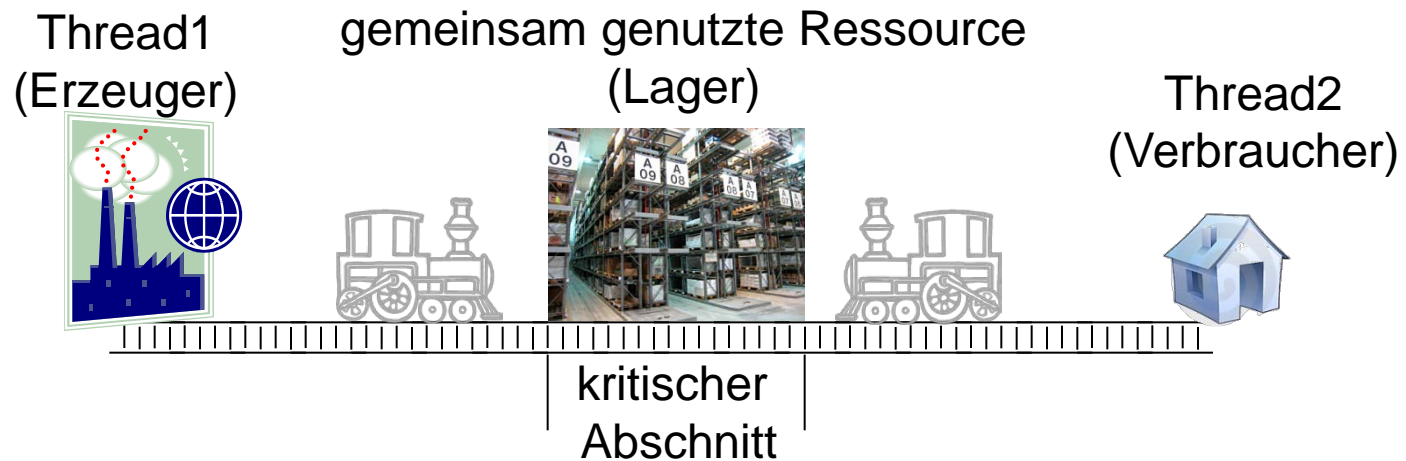
BEKANNTE METHODIKEN

Inhalt

- Locks / Semaphoren
- Monitorkonzept
- Message Passing
- Fork/Join

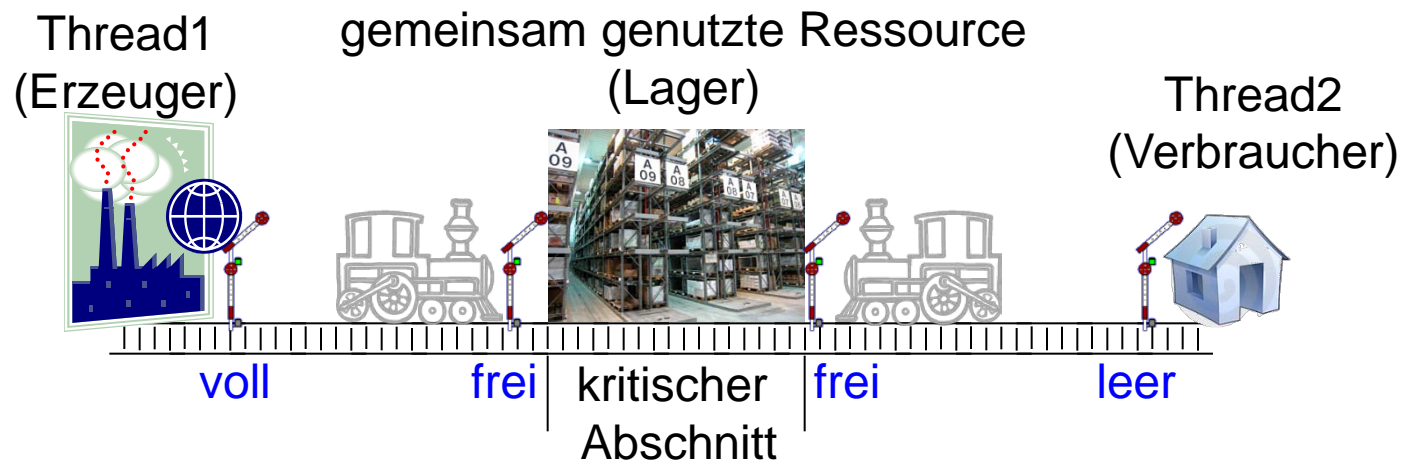
Locks / Semaphoren

- verschiedene Threads greifen auf gemeinsam genutzte Ressource zu
- Erzeuger-Verbraucher Problem

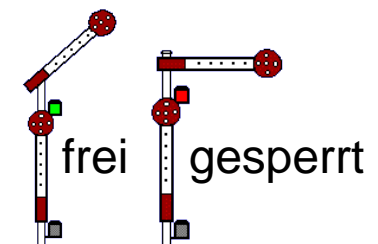


Locks / Semaphoren

- verschiedene Threads greifen auf gemeinsam genutzte Ressource zu
- Erzeuger-Verbraucher Problem



- Lösung von Dijkstra: zählende Semaphoren
- Atomare Aktion
- Binäre Semaphoren = Mutexe



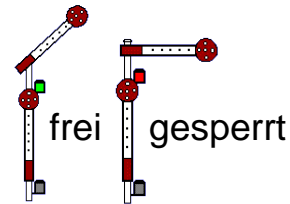
Monitorkonzept

- Monitor = abstrakter Datentyp, Klasse
- Kennzeichnet Zugriff auf kritischen Bereich
 - `synchronized`
- Höherstufiges Synchronisationsprimitiv
 - Compiler setzt Semaphoren
- Zustandsvariablen
 - Operation `wait()` und `notify()`

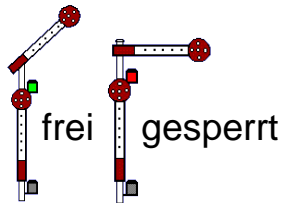
Monitorkonzept

```
static class monitor {
    private int buffer[] = new int[N];
    private int count=0, lo=0, hi=0;

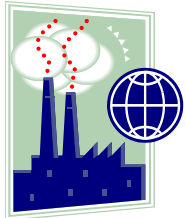
    public synchronized void insert(int val)
    {
        if(count==N) wait();
        buffer [hi] = val;
        hi=(hi+1)%N;
        count=count+1;
        if(count==1) notify();
    }
}
```



```
public synchronized int remove()
{
    int val;
    if(count==0) wait();
    val=buffer[lo];
    lo=(lo+1)%N;
    count=count-1;
    if(count==N-1) notify();
    return val;
}
}
```



```
static class producer extends Thread {
    public void run(){
        int item;
        while(true) {
            item = produce_item();
            mon.insert(item);
        }
    }
    private int produce_item()
    {...}
}
```

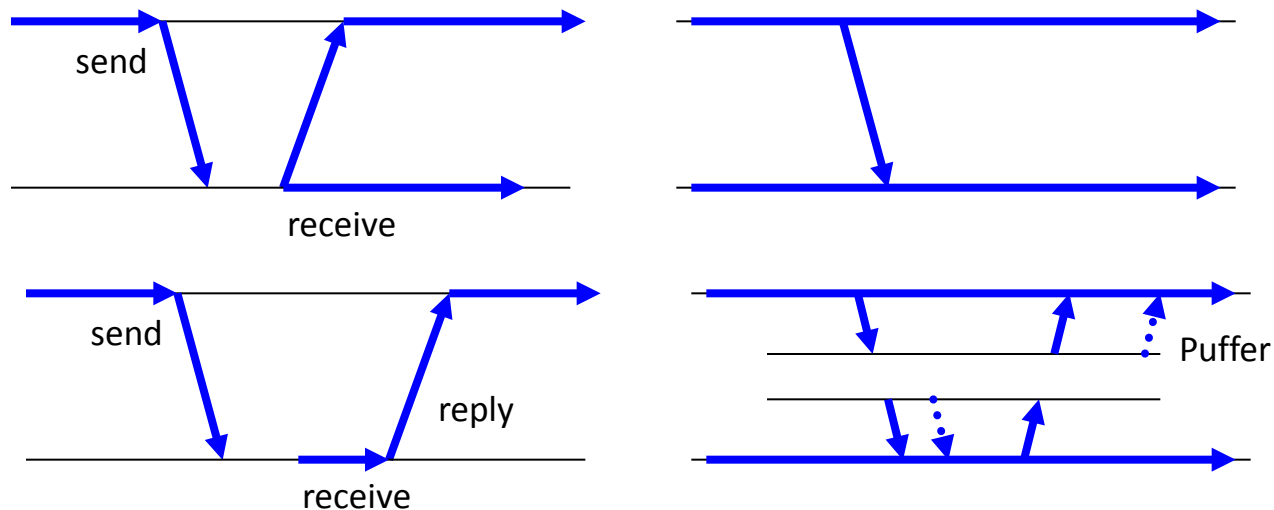


```
static class consumer extends Thread {
    public void run() {
        int item;
        while(true) {
            item = mon.remove();
            consume_item(item);
        }
    }
    private void consume_item(int item)
    {...}
}
```



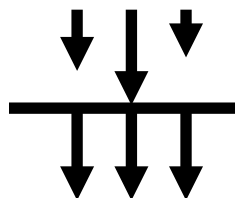
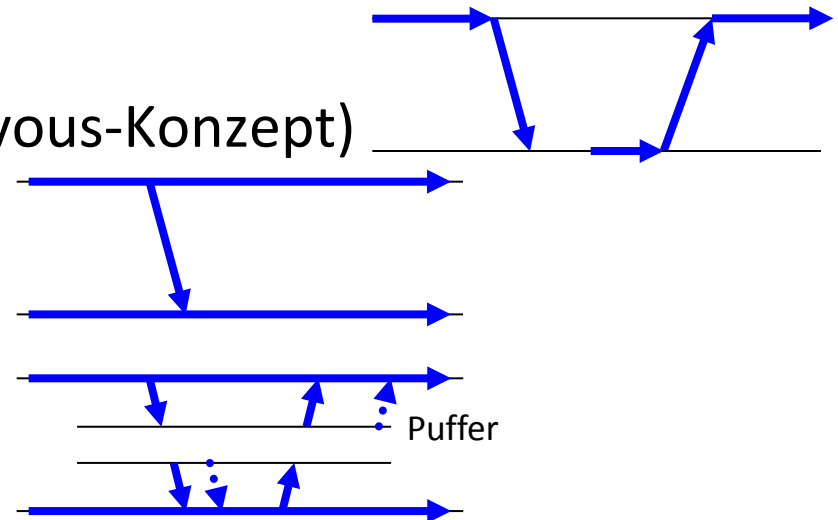
Message Passing

- Synchronisation von verteilten Systemen
- Kommunikation beruht auf dem Versenden und Empfangen von Nachrichten (send / receive)
- Asynchrone und synchrone Kommunikation
- Gepufferter / ungepufferter Modus
 - Mailbox (Nachrichtenpuffer)



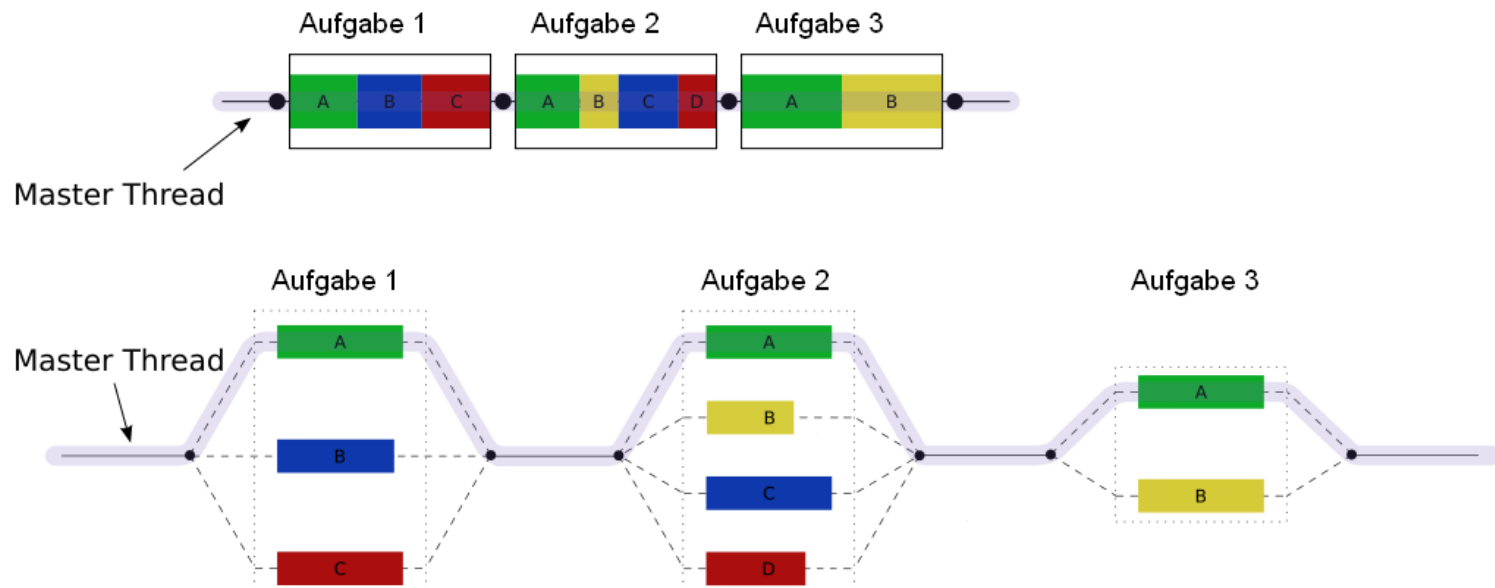
Message Passing

- Message Passing Interface (MPI)
 - Von IEEE standardisierte Kommunikationsbibliothek
 - Legt Syntax, Semantik und Grundoperationen fest
- Punkt zu Punkt Kommunikation
 - Blockierender Modus (Rendezvous-Konzept)
- Asynchrone Kommunikation
 - Nicht blockierender Modus
 - (Puffer) blockierender Modus
- Gruppenkommunikation
 - Broadcast
- Barriersynchronisation



Fork/Join

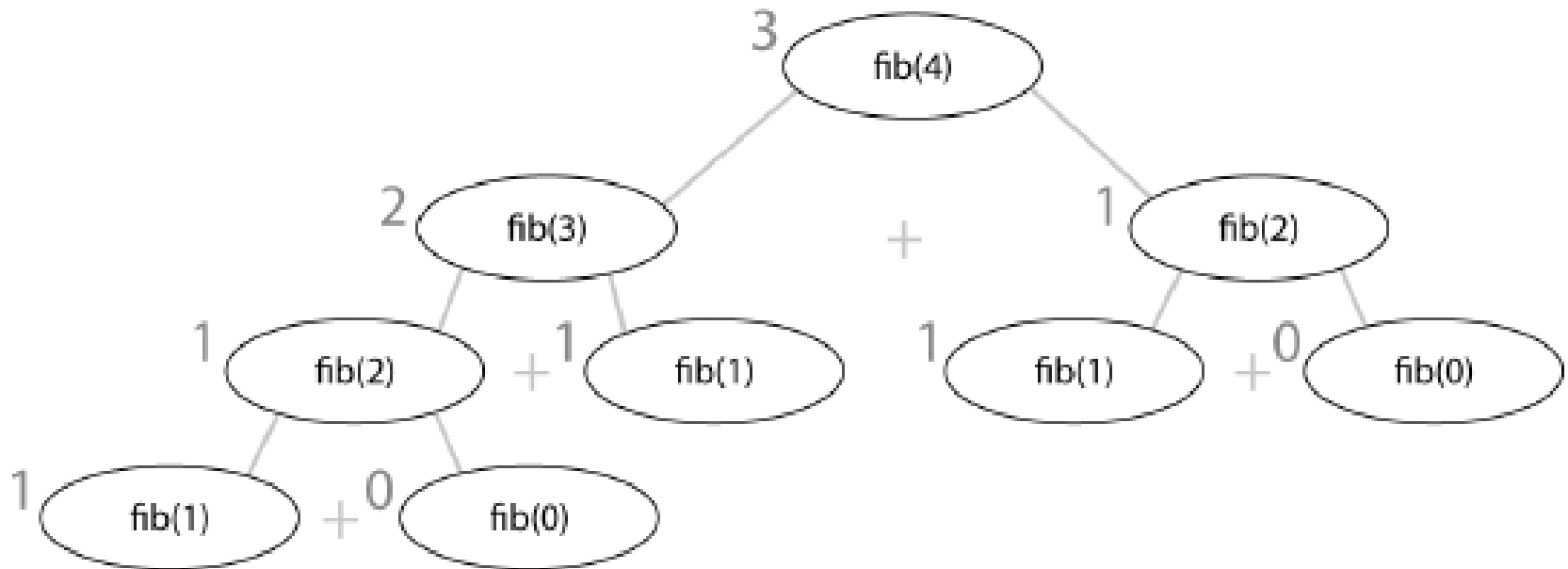
- `fork`: perfekte Kopie/Snapshot des Vaterprozesses
- Divide-and-Conquer
- Master-Thread
 - Thread pro parallelisierbare Teilaufgabe
 - Zusammenführen/Synchronisieren



Fork/Join

$$\text{fib}(n) = \begin{cases} n & \text{wenn } n \leq 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{sonst} \end{cases}$$

Baum bei rekursiver Berechnung



Automatisierung von Multithreading

Inhalt

- Automatische Parallelisierung
- Automatische Vektorisierung
- Parallele Sprachen
- OpenMP / JaMP

Automatische Parallelisierung

- Automatische Parallelisierung durch Compiler (von Sun und Intel für C/C++, Fortran)
 - Sequentielles Programm → paralleles Programm
- Parallelisierung von Schleifen
 - Analyse der Datenabhängigkeit
 - Paralleler Code für einfache Schleifen
 - Einfache Code-Transformationen

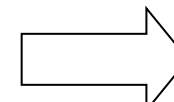
```
for (i=0; i<N; i++)  
    for (j=0; j<M; j++)  
        a[i][j]=b[i][j] * c[i][j];
```


Automatische Parallelisierung

- Probleme bei der Parallelisierung zum Teil durch Code-Transformationen lösbar

– skalare Werte  Feldvariablen statt skalare Variablen

```
for (i=0; i<N; i++) {  
    t = a[i];  
    a[i] = b[i];  
    b[i] = t;  
}
```

– Funktionsaufrufe  „Inlining“ von Funktionen

```
for (i=0; i<N; i=i++)  
    a[i] = a[i] + compute(a, b, i);
```

Automatische Parallelisierung

- Beispiele nicht lösbarer Probleme bei der Parallelisierung

- Index-Abhängigkeiten

```
for (i=1; i<N; i++)  
    a[i] = a[i-1] + b;
```

- indexabhängige Bedingungen

```
for (i=0; i<N; i++)  
    for (j=0; j<M; j++) {  
        if (j>i)  
            a[i][j] = a[i][j] + b[i][j] * c; }  
    }
```

- mehrere datenabhängige Ausgängen

```
for (i=0; i<N; i++) {  
    if (b[i] == 0) break;  
    a[i] = a[i] / b[i]; }
```

Automatische Vektorisierung

- Für Vektorprozessoren bzw. Vektoreinheiten
- Voraussetzung: Schleifeniterationen unabhängig
- Vorgehensweise
 - N Iterationen einer Schleife in Streifen unterteilen
 - k Feldelemente pro Streifen
 - 1 Streifen passt in ein Vektorregister
- Beispiel: Prozessor mit 128 Bit Vektorregister

- vor Vektorisierung

```
for (i=0; i<N; i++)  
    c[i] = a[i] * b[i];
```

- nach Vektorisierung

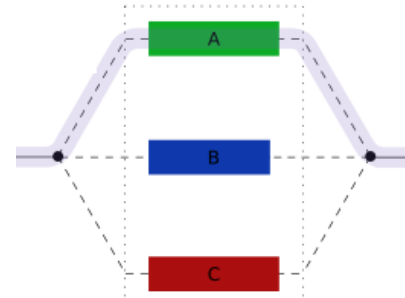
```
for (i=0; i<N; i++)  
    c[i:i+3] = a[i:i+3] * b[i:i+3];
```

c[0]	c[1]	c[2]	c[3]
4x8Bit	4x8Bit	4x8Bit	4x8Bit

Parallele Sprachen

- Fortress
 - Open Source Forschungsprojekt von Sun
 - Programmiersprache für High Performance Computing
 - Impliziter Parallelismus
 - `for`-Schleifen, `also-do`-Ausdrücke, Tupel-Ausdrücke, Summen automatisch parallelisiert

```
for i ← 1:m, j ← 1:n do  
    a[i,j] := b[i] c[j]  
end
```

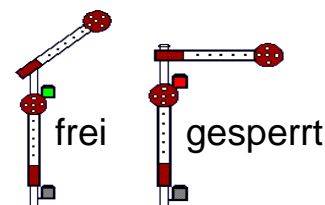


- Explizite Threads mit `spawn`

```
t1 = spawn do e1 end
```

```
t2 = spawn do e2 end
```

- Wechselseitiger Ausschluss `atomic`



Parallele Sprachen

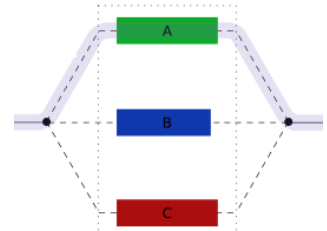
- Unified Parallel C (UPC)

- Statische Anzahl Threads

```
upcc -o hello -THREADS 4 helloworld.upc
```

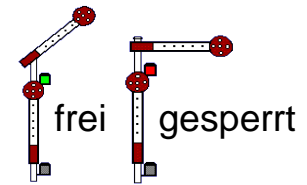
- `upc_forall` zur Lastverteilung auf mehrere Threads

```
upc_forall (i=0; i<100; i++; &a[i])  
    a[i]=b[i]*c[i];
```



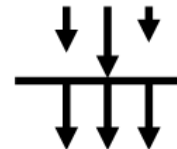
- Sperrfunktionen (Zugriff auf shared Daten/Variablen)

```
void upc_lock(upc_lock_t *l)  
void upc_unlock(upc_lock_t *l)
```



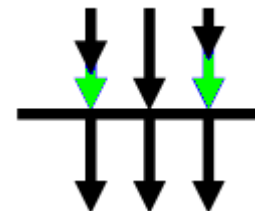
- Barrieren

```
upc_barrier expression;
```



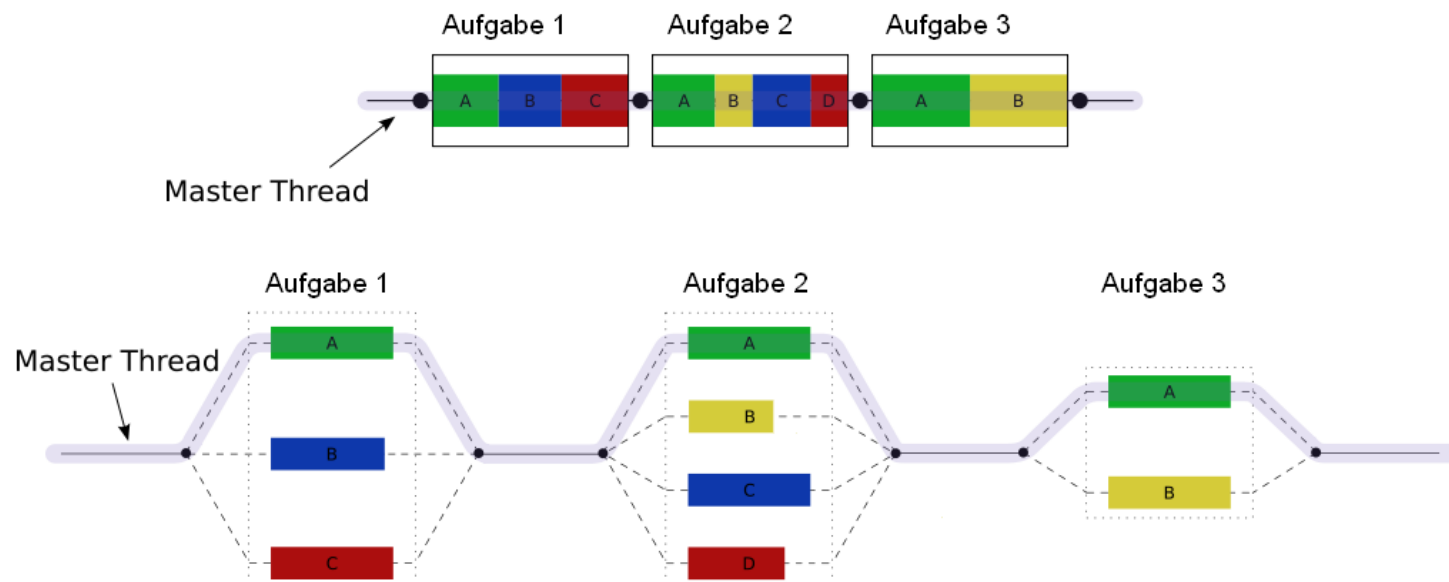
- Split Phase Barrieren

```
upc_notify expression;  
upc_wait expression;
```



OpenMP / JaMP

- Bibliothek für C/C++, Fortran bzw. Java zum halbautomatischen Parallelisieren
- Kennzeichnung von parallelen Blöcken durch Direktiven
`#pragma omp direktive-name [clause[clause]...]`
- Master-Thread für sequentiellen Anteil
- Lastverteilung bei parallelen Blöcken auf mehrere Threads



OpenMP / JaMP

- `for` Pragma zum Parallelisieren von Schleifen
 - Teilt Schleifeniterationen unter Threads auf

- Beispiel: $y = a * x + y$

```
const float a;
```

```
const vector<float>& x;
```

```
const vector<float>& y;
```

```
...
```

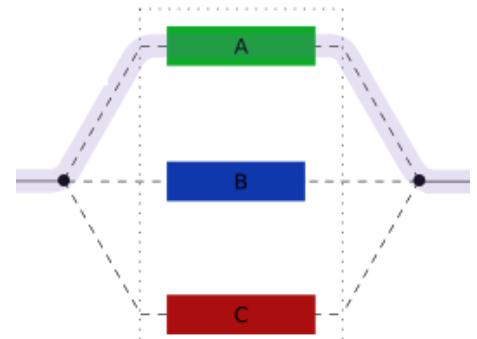
```
#pragma omp parallel for
```

```
for(int i=0; i<x.size(); i++)
```

```
{
```

```
    y[i] += a * x[i];
```

```
}
```



OpenMP / JaMP

- `master` Pragma
Codeblock darf nur vom Master-Thread ausgeführt werden
- `single` Pragma
Codeblock darf nur von einem Thread ausgeführt werden

```
#pragma omp parallel
{
    DoManyThings( );
    #pragma omp single
    {
        printf("Hello from single"\n");
    } /* Die anderen Threads warten hier */
    DoRestofThings( );
}
```


OpenMP / JaMP

- Threads können standardmäßig auf alle Variablen im parallelen Block zugreifen
- Datenzugriffsklausel zur Festlegung der Sichtbarkeit und des Gültigkeitsbereichs von Variablen

`private(var)`

`shared(var)`

`reduction(operator: var)`

- Gemeinsam genutzte Variablen
- Nur bestimmte Operationen

OpenMP / JaMP

- Wechselseitiger Ausschluss durch Kennzeichnung der kritischen Abschnitte

```
#pragma omp critical  
#pragma omp atomic
```

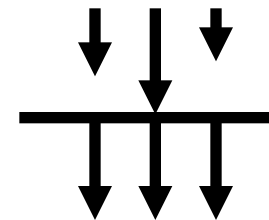
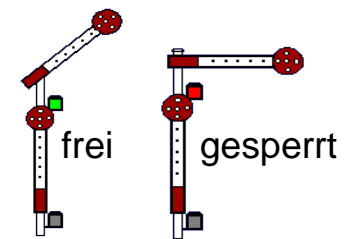
- Wechselseitiger Ausschluss durch Locks

```
omp_set_lock  
omp_unset_lock
```

- Barrieren
 - Implizit z.B. bei `for`-Direktiven
 - Explizit

```
#pragma omp barrier
```

- Synchronisation aller Threads
- ggf. Ausbremsen paralleler Threads

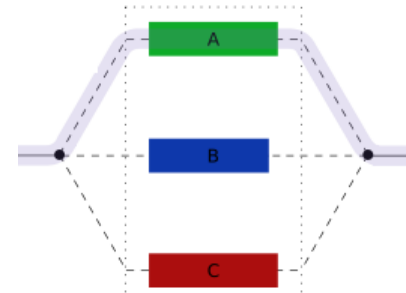
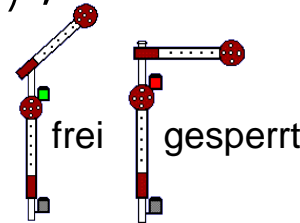


OpenMP / JaMP

$$\Pi = \int_0^1 \frac{4}{1+x^2} dx$$

- Berechnung von π

```
const double delta_x = 1.0 / num_iter;
double sum = 0.0;
double x, f_x;
int i;
#pragma omp parallel for private(x,f_x) shared(sum)
for (i = 1; i <= num_iter; i++) {
    x = delta_x * (i-0.5);
    f_x = 4.0 / (1.0 + x*x);
    #pragma omp critical
        sum += f_x;
}
return delta_x * sum;
```

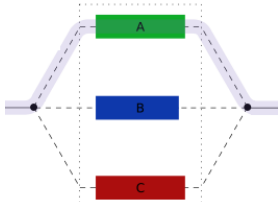


OpenMP / JaMP

- Berechnung von π mit Reduktion
 - Variablen implizit privat
 - Mögliche Operatoren: $+$, $*$, $-$, $\&$, $|$, $^$, $\&\&$, $||$

...

```
#pragma omp parallel for private(x,f_x) reduction(+:sum)
for (i = 1; i <= num_iter; i++) {
    x = delta_x * (i-0.5);
    f_x = 4.0 / (1.0 + x*x);
    sum += f_x; //berechne lokale sums pro Thread
}
return delta_x * sum; //sum enthält die Summe aller
                      //lokalen Instanzen von sum
```

A diagram showing a hexagonal mesh structure. Three nodes are highlighted: a green node labeled 'A' at the top, a blue node labeled 'B' in the middle, and a red node labeled 'C' at the bottom. Dashed lines connect these nodes to form a central triangle and extend to other parts of the mesh.

„Writing correct programs is hard; writing correct concurrent programs is harder“

– Brian Goetz

CONCURRENT JAVA PROGRAMMING

CONCURRENT JAVA PROGRAMMING

Probleme

Java Puzzle [1/3]

```
public class StopThread {  
    private static boolean stopRequested;  
    public static void main(String[] args)  
        throws InterruptedException {  
        Thread backgroundThread = new Thread(new Runnable() {  
            public void run() {  
                int i = 0;  
                while (!stopRequested)  
                    i++;  
            }  
        });  
        backgroundThread.start();  
        TimeUnit.SECONDS.sleep(1);  
        stopRequested = true;  
    }  
}
```

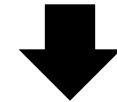
Wie lange läuft das Programm?

- a. Ungefähr eine Sekunde
- b. Unterschiedlich
- c. Gar nicht
- d. Endlos

Java Puzzle [2/3]

```
public class StopThread {  
    private static boolean stopRequested;  
    public static void main(String[] args)  
        throws InterruptedException {  
        Thread backgroundThread = new Thread()  
        {  
            public void run() {  
                int i = 0;  
                while (!stopRequested)  
                    i++;  
            }  
        };  
        backgroundThread.start();  
        TimeUnit.SECONDS.sleep(1);  
        stopRequested = true;  
    }  
}
```

```
while(!done) {  
    i++;  
}
```



```
if(!done)  
    while(true)  
        i++;
```

Wie lange läuft das Programm?

- a. Ungefähr eine Sekunde
- b. Unterschiedlich**
- c. Gar nicht
- d. Endlos

Java Puzzle [3/3]

```
public class StopThread {  
    private static volatile boolean stopRequested;  
    public static void main(String[] args)  
        throws InterruptedException {  
        Thread backgroundThread = new Thread(new Runnable() {  
            public void run() {  
                int i = 0;  
                while (!stopRequested)  
                    i++;  
            }  
        });  
        backgroundThread.start();  
        TimeUnit.SECONDS.sleep(1);  
        stopRequested = true;  
    }  
}
```

[→ Effective Java, Item 66]

Prinzipien [1/2]



Synchronisierung(Synchronisation)



Atomarität (Atomicity)

Synchronisierung + Atomarität [1/3]

```
final Hashtable<String, String> table =  
    new Hashtable<String, String>();  
  
// important code  
  
if (!table.contains("foo")) {  
    table.put("foo", "bar");  
}
```

Synchronisierung + Atomarität [2/3]

```
final Hashtable<String, String> table =  
    new Hashtable<String, String>();  
  
// important code  
  
if (!table.contains("foo")) {  
    table.put("foo", "bar");  
}
```

- *Check-Then-Act*
- *Hashtable* ist threadsicher
- Verwendung von *contains* und *put* nicht atomar

Synchronisierung + Atomarität [3/3]

```
final Hashtable<String, String> table =  
    new Hashtable<String, String>();  
// important code  
synchronized (table) {  
    if (!table.contains("foo")) {  
        table.put("foo", "bar");  
    }  
}
```

Atomarität der Operation

!=

Threadsicherheit der Klasse

Prinzipien [2/2]



Unveränderbarkeit (Immutability)

- Zustand des Objekts kann nach der Erzeugung nicht mehr verändert werden → threadsicher
- Ändernde Operationen liefern neue Instanz

Klassische Umsetzung

- `synchronized`
 - Methoden **(Achtung!)**
 - Implizite Synchronisierung auf die Instanz/Klasse (statisch)
 - Blöcke
 - Synchronisierung auf beliebige Objekte
- `volatile`
 - Felder
 - garantiert atomare Operationen
 - Für primitive Datentypen außer long und double
 - Für Objektreferenzen, z.B. publizieren unveränderbarer Objekte

Klassische Umsetzung

- `synchronized`
 - Methoden
 - Implizite Synchronisierung auf die Instanz/Klasse (statisch)
 - Blöcke (**Achtung!**)
 - Synchronisierung auf beliebige Objekte
- `volatile`
 - Felder
 - garantiert atomare Operationen
 - Für primitive Datentypen außer long und double
 - Für Objektreferenzen, z.B. publizieren unveränderbarer Objekte

Klassische Umsetzung

- `synchronized`
 - Methoden
 - Implizite Synchronisierung auf die Instanz/Klasse (statisch)
 - Blöcke
 - Synchronisierung auf beliebige Objekte
- `volatile`
 - Felder **(Achtung!)**
 - garantiert atomare Operationen
 - Für primitive Datentypen außer long und double
 - Für Objektreferenzen, z.B. publizieren unveränderbarer Objekte

CONCURRENT JAVA PROGRAMMING

Explicit Locking

ReentrantLock

Funktionsweise vergleichbar zu **synchronized**

```
Lock lock = new ReentrantLock();  
  
lock.lock();  
try {  
    // compound operation  
} finally {  
    lock.unlock();  
}
```

- Verbesserungen im Vergleich zu **synchronized**
 - Warten unterbrechbar
 - Verhindern von Deadlocksituationen
 - **tryLock** & **tryLock(TIMEOUT, ...)**

ReentrantLock - Erweiterung

- Wenn Lesezugriffe häufiger als Schreibzugriffe
– ReadWriteLock

```
public void put(final int i) {  
    final WriteLock writeLock = rrw.writeLock();  
    try {  
        value = i;  
    } finally {  
        writeLock.unlock();  
    }  
}  
public int get() {  
    final ReadLock readLock = rrw.readLock();  
    try {  
        return value;  
    } finally {  
        readLock.unlock();  
    }  
}
```

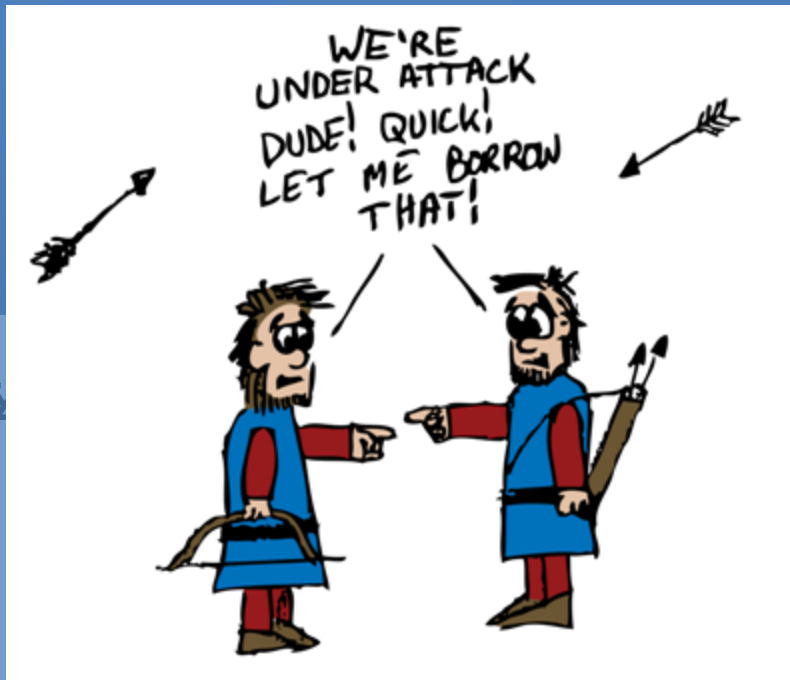
ReentrantLock vs. synchronized [1/3]

`transferMoney(fromAccount, toAccount, amount)`

```
synchronized(fromAccount) {  
    synchronized(toAccount) {  
        // do actions  
    }  
}
```

Reentrant

transferMoney



nized [1/3]

ant, amount)

s

}

ReentrantLock vs. synchronized [2/3]

```
private static final Object tieLock = new Object();

int fromHash = System.identityHashCode(fromAcct);
int toHash = System.identityHashCode(toAcct);

if (fromHash < toHash) {
    synchronized (fromAcct) {
        synchronized (toAcct) {
            // do actions
        }
    }
} else if (fromHash > toHash) {
    synchronized (toAcct) {
        synchronized (fromAcct) {
            // do actions
        }
    }
} else {
    synchronized (tieLock) {
        synchronized (fromAcct) {
            synchronized (toAcct) {
                // do actions
            }
        }
    }
}
```

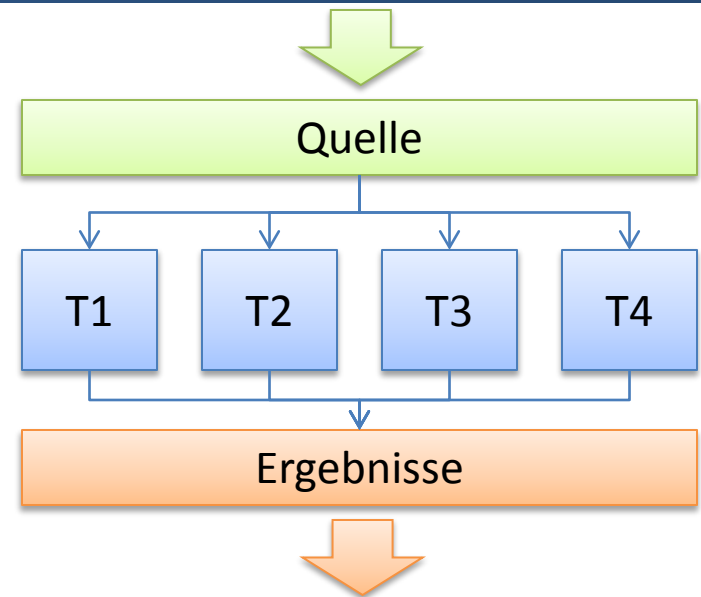
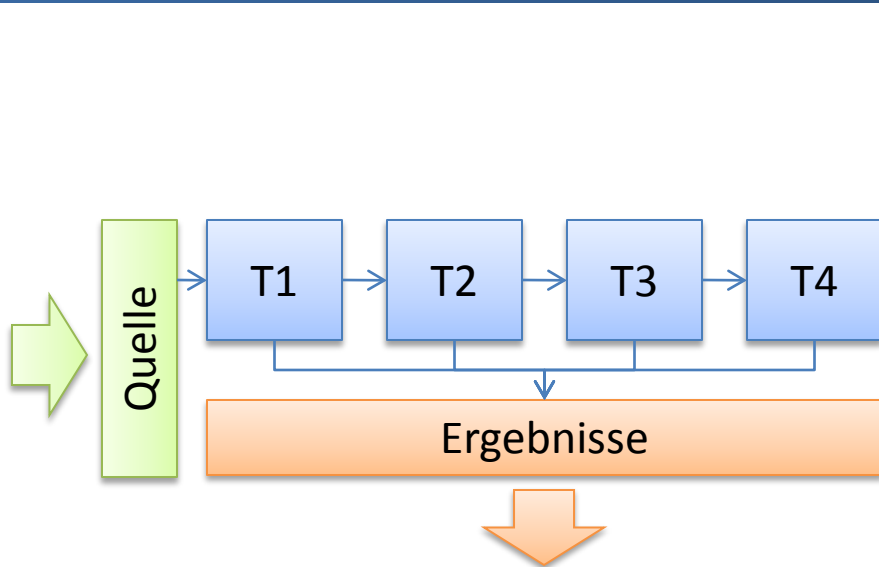
ReentrantLock vs. synchronized [3/3]

```
while (true) {
    if (fromAcct.lock.tryLock()) {
        try {
            if (toAcct.lock.tryLock()) {
                try {
                    // do actions
                } finally {
                    toAcct.lock.unlock();
                }
            }
        } finally {
            fromAcct.lock.unlock();
        }
    }
    if (System.nanoTime() < stopTime)
        return false;
    NANOSECONDS.sleep(fixedDelay + rnd.nextLong() % randMod);
}
```


CONCURRENT JAVA PROGRAMMING

Task Execution

Konzept



- Unabhängige Aufgaben
 - Parallelisierbar
 - Mehrere Aufgaben als Paket zusammen durchführen (Unit Of Work [PoEAA184])

Executor

- Abstraktion über das Ausführen der Tasks
 - Thread Pool Verwaltung
 - Verteilen der Tasks

```
public interface Executor {  
    void execute(Runnable command);  
}
```

- Vorteile
 - Kontrolle über die Anzahl der Threads
 - Wiederverwendung „leerer“ Threads
 - Automatische Neuerzeugung wenn Thread stirbt

Executor Service [1/3]

- Erweitert Executor
- Tasks können Ergebnisse zurückliefern

- Direkte Ergebnisse

- `<T> Future<T> submit(Callable<T> task)`

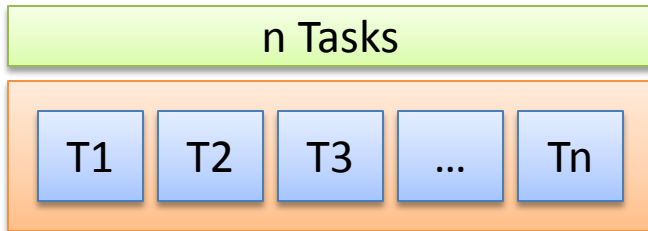
- `<T> Future<T> submit(Runnable task, T result)`

- Durchgelaufen ja/nein

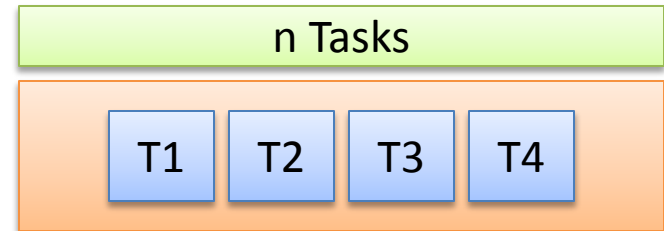
- `Future<?> submit(Runnable task)`

Executor Service [2/3]

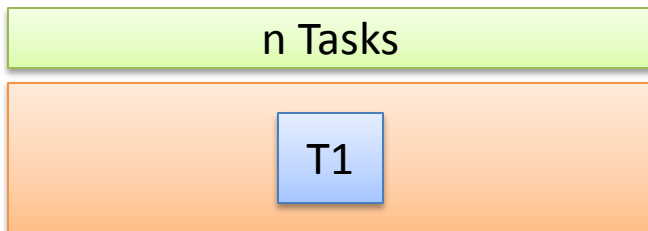
CachedThreadPool



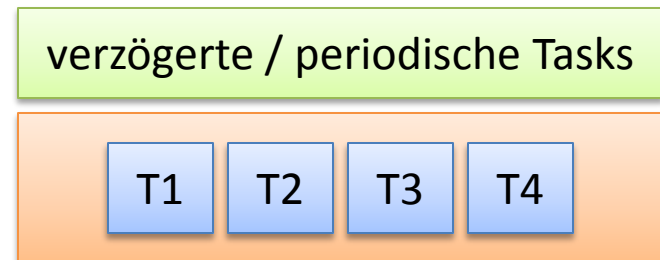
FixedThreadPool



SingleThread

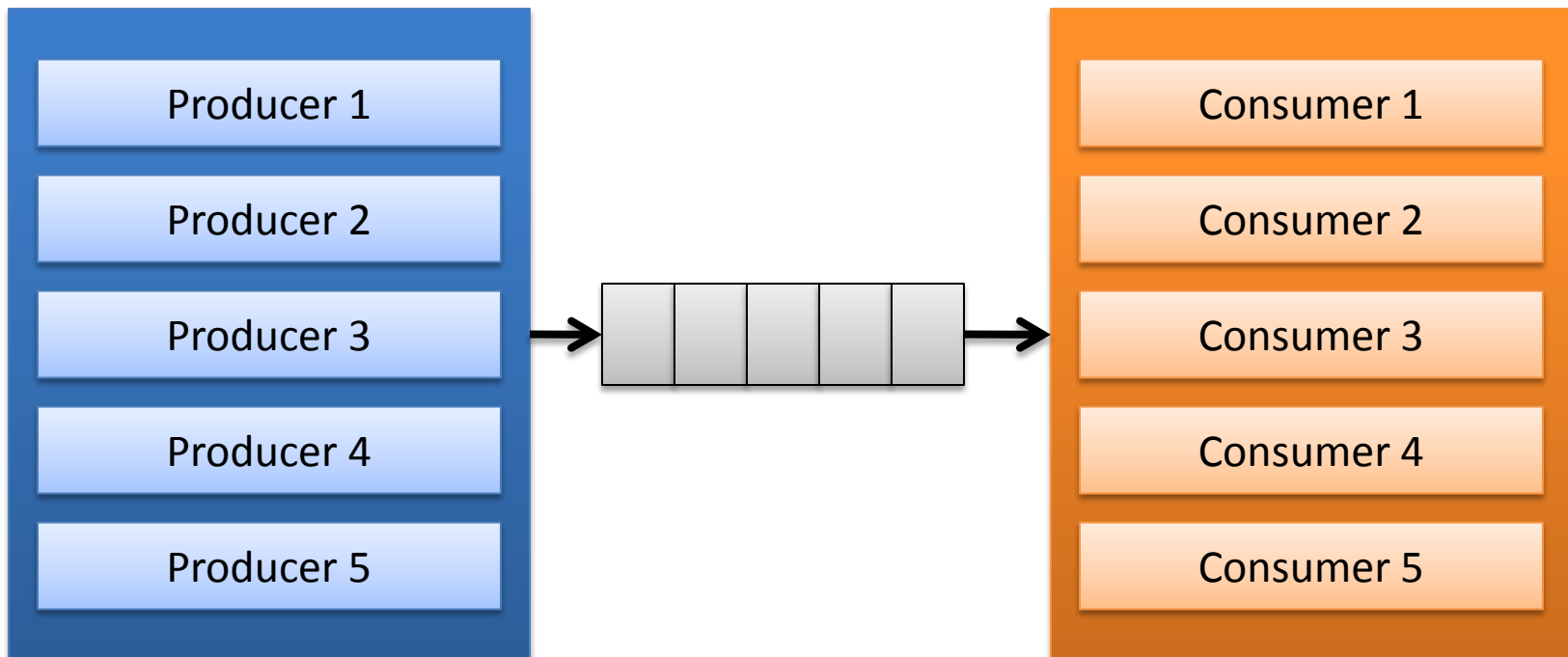


ScheduledThreadPool



Executor Service [3/3]

- Erzeugung durch Hilfsklasse
 - `Executors.new???(<Konfiguration>)`
- Optimal für Consumer-Producer



Executor Service – Lifecycle [1/2]

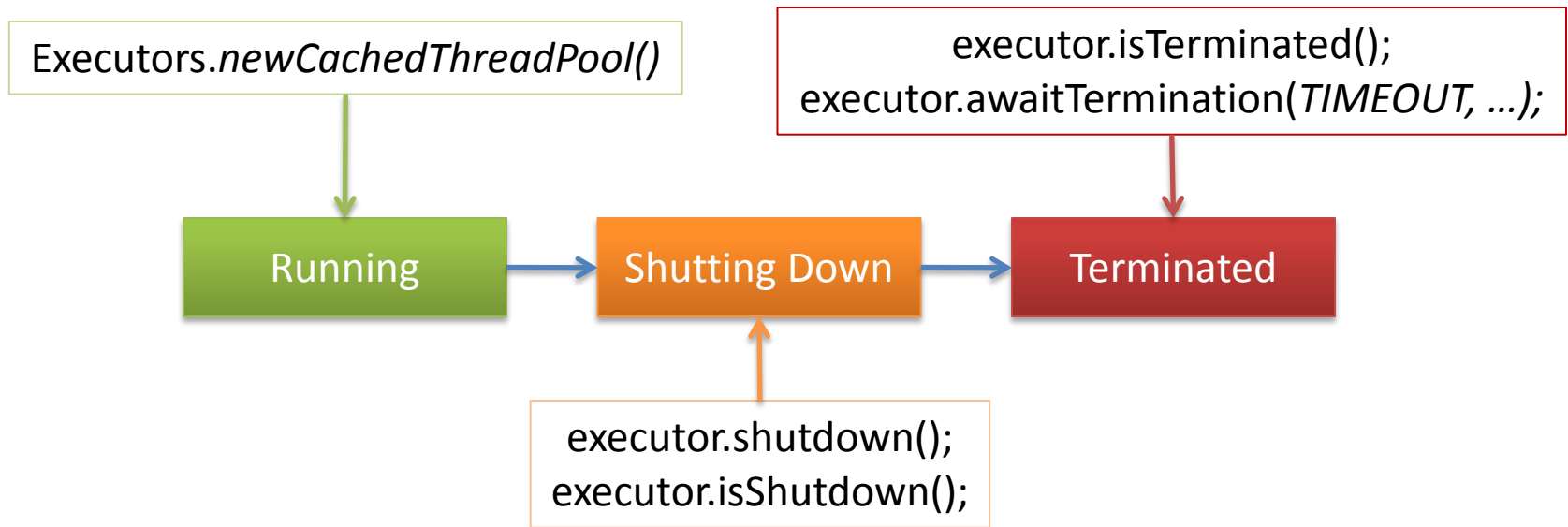


Running: Initialer Zustand

Shutting Down: - Keine neuen Tasks annehmen
- Vorhandene laufen zu Ende

Terminated: Alle verbleibenden Tasks beendet

Executor Service – Lifecycle [2/2]



Running: Initialer Zustand

Shutting Down: - Keine neuen Tasks annehmen
- Vorhandene laufen zu Ende

Terminated: Alle verbleibenden Tasks beendet

Anwendung: WebServer

```
public class TaskExecutionWebServer {
    private static final int NTHREADS = 100;
    private static final Executor exec
        = Executors.newFixedThreadPool(NTHREADS);
    public static void main(String[] args) throws IOException{
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            final Socket connection = socket.accept();
            Runnable task = new Runnable() {
                public void run() {
                    handleRequest(connection);
                }
            };
            exec.execute(task);
        }
    }
    private static void handleRequest(Socket connection) {
        // request-handling logic here
    }
}
```

Anwendung: WebServer

```
public class TaskExecutionWebServer {
    private static final int NTHREADS = 100;
    private static final Executor exec
        = Executors.newFixedThreadPool(NTHREADS);
    public static void main(String[] args) throws IOException{
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            final Socket connection = socket.accept();
            Runnable task = new Runnable() {
                public void run() {
                    handleRequest(connection);
                }
            };
            exec.execute(task);
        }
    }
    private static void handleRequest(Socket connection) {
        // request-handling logic here
    }
}
```

CONCURRENT JAVA PROGRAMMING

Weitere Erweiterungen seit Java 5

Collections

Problem vor Java 5: Iteratoren

ConcurrentHashMap

- Performant & Skalierbar durch Lock-Striping
- Atomare Operationen
 - **putIfAbsent**

BlockingQueue

- Ideal bei *Producer-Consumer-Implementierungen*

Atomare Datentypen

- AtomicInteger, AtomicLong, AtomicBoolean, ...
 - Threadsicher
 - Atomare Operationen
 - *incrementAndGet, get, ...*
 - „*Die besseren volatiles*“

CONCURRENT JAVA PROGRAMMING

Zusammenfassung

Best Practices [1/2]

- Sichtbarkeit von Feldern einschränken [EJ, I13]
 - Was nicht öffentlich sein **MUSS** → `private`
- Veränderbarkeit minimieren [EJ, I15]
 - Was nicht änderbar sein **MUSS** → `final`
- Synchronisierte Bereiche klein halten [EJ, I67]
- Thread-(nicht-)Sicherheit dokumentieren
- Synchronisierung wenn möglich **IN** den Objekten
 - Objekte Thread-Safe designen und die Synchronisierung kapseln

Best Practices [2/2]

- Executor Framework der direkten Verwendung von Threads vorziehen
 - ScheduledThreadPool der Verwendung von TimerTask vorziehen!

Entwicklungsumgebung

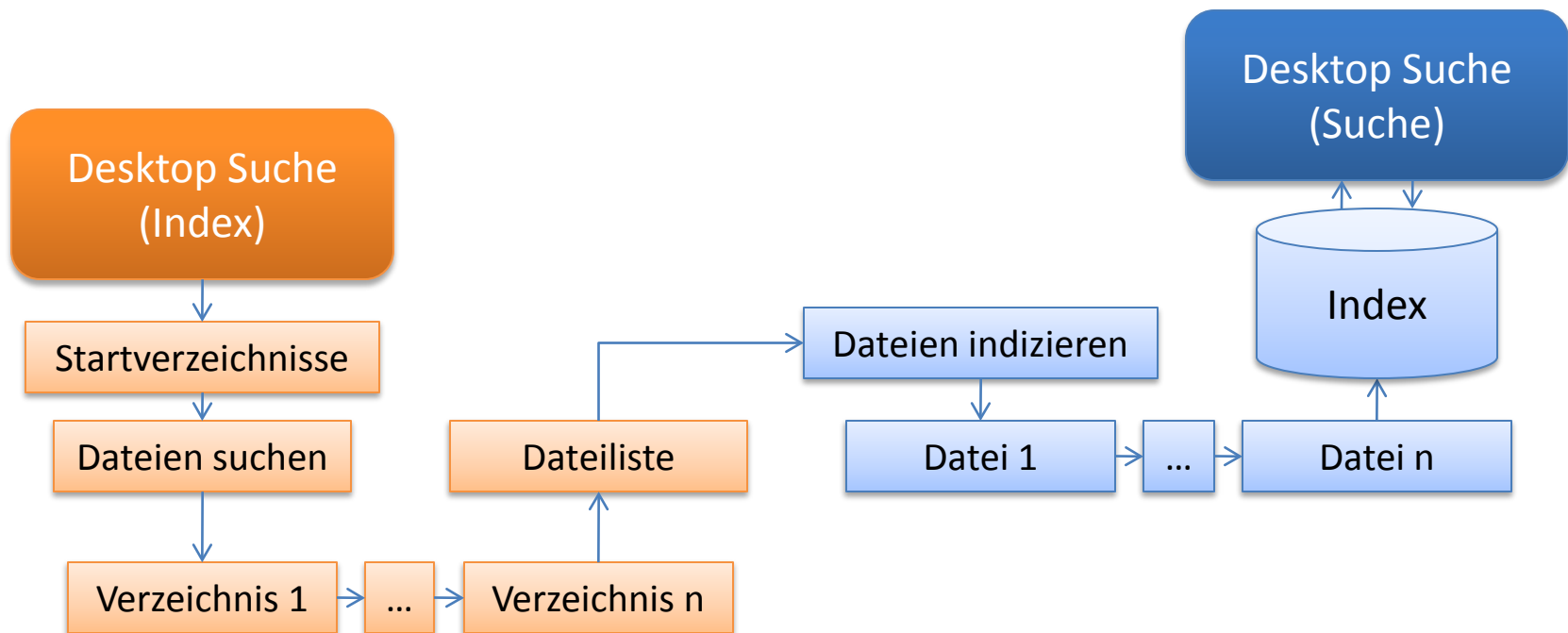
- **JVM** mit **-server** starten
- Statische Code Analyse
 - FindBugs [<http://findbugs.sourceforge.net>]

ÜBUNG

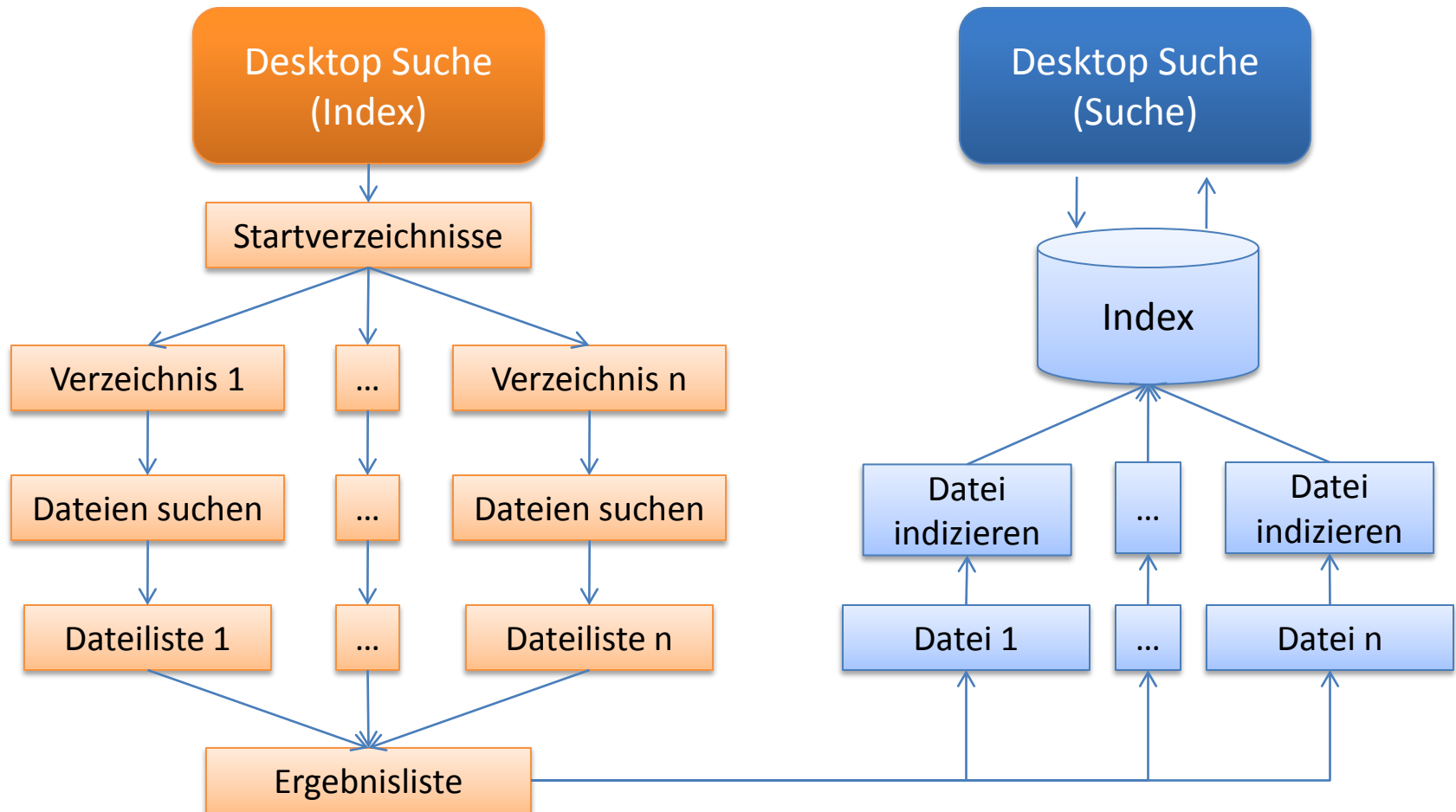
ÜBUNG

DESKTOP SUCHE

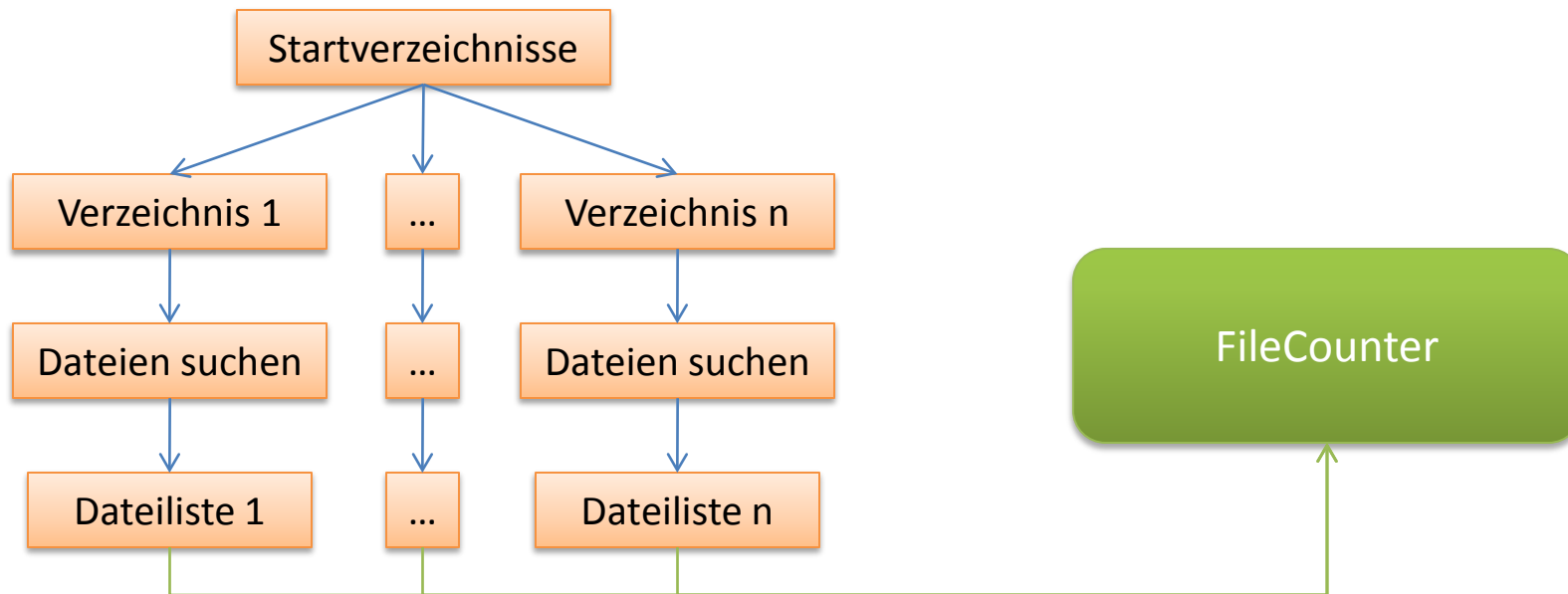
Aufgabenstellung [1/2]



Aufgabenstellung [2/2]



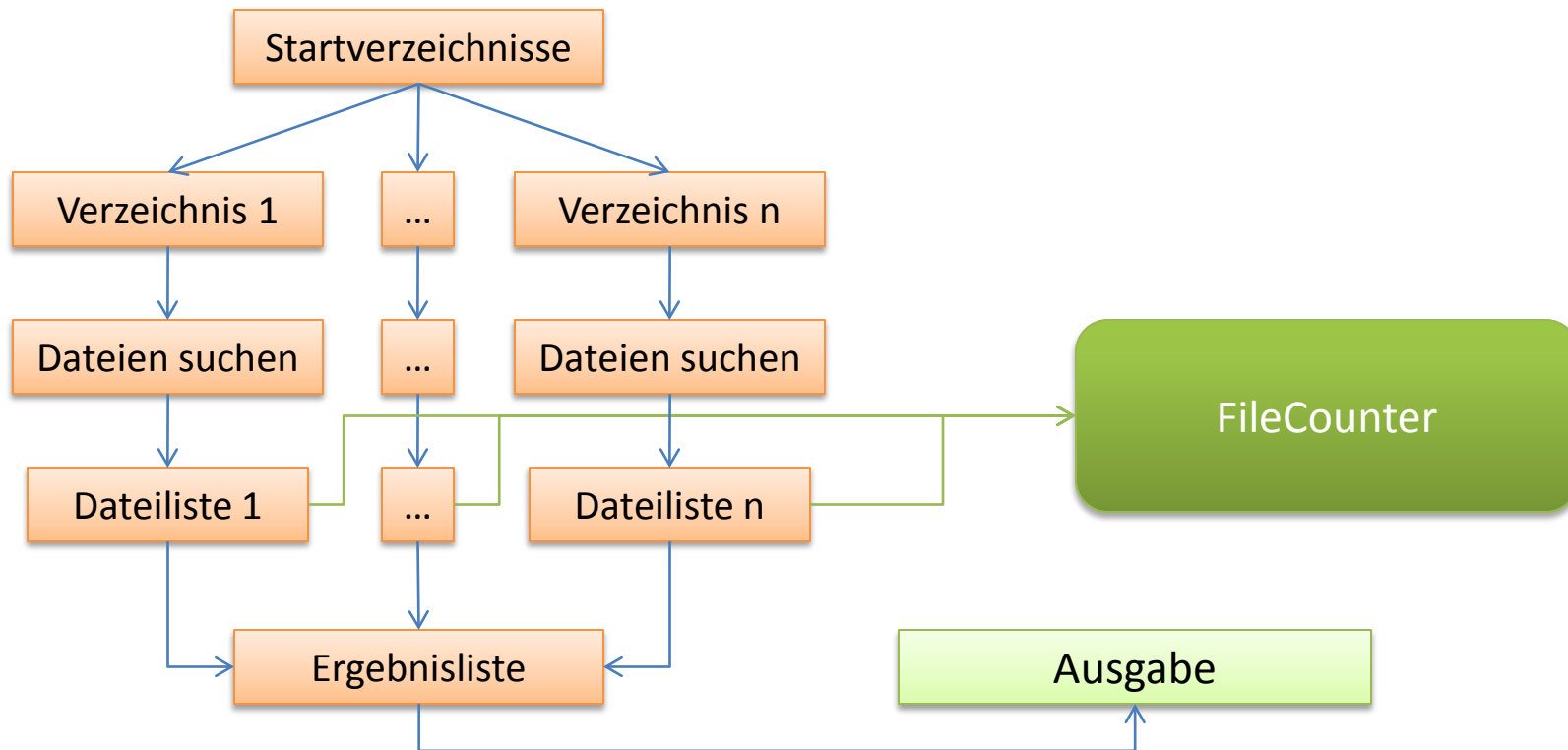
Aufgabe – Teil 1



Crawler parallelisieren – Teil 1

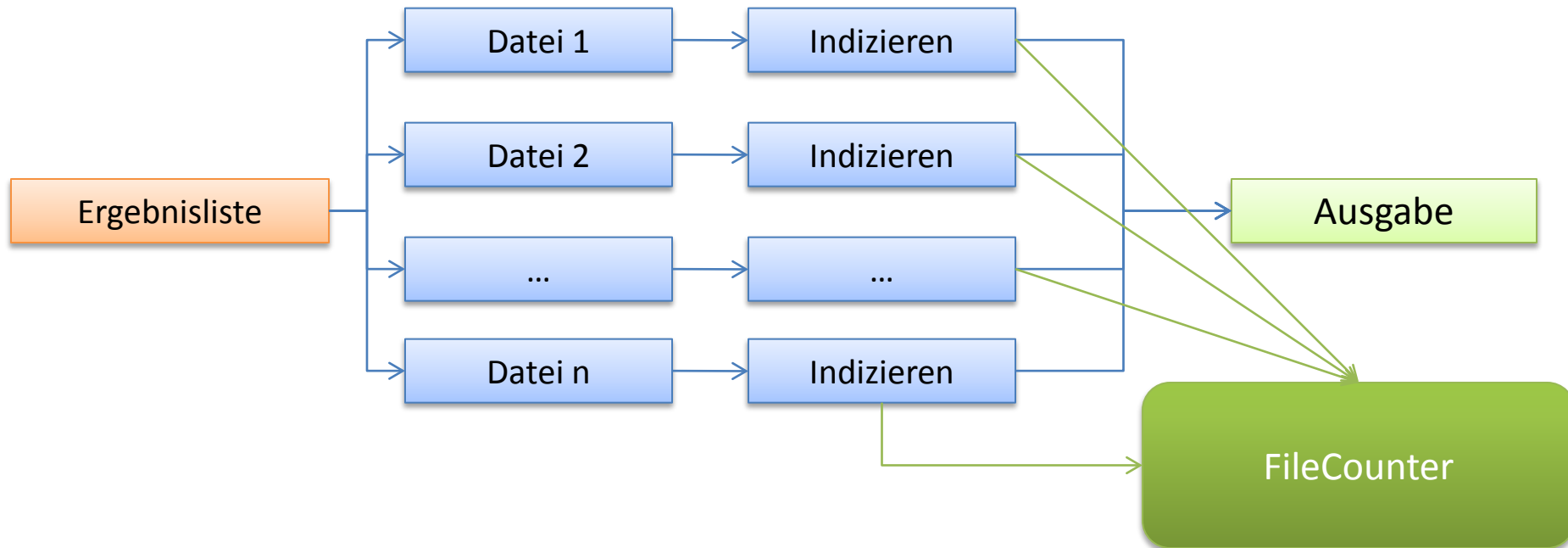
- Aufteilung anhand der Startverzeichnisse
- Implementierung mittels des Executor Frameworks

Aufgabe – Teil 2



Crawler parallelisieren – Teil 2
– Rückgabe der gefundenen Dateien

Aufgabe – Teil 3



Parallellisieren der Indizierung

- Weiterverarbeitung der Ergebnisse des Crawlers

Entwicklungsumgebung

C:/Temp/eclipse-iws-multithreading
/eclipse/eclipse.exe

Musterlösung

<http://github.com/jwachter/iws-fileindex-example>

MULTITHREADING

Was bietet C#?

Multithreading mit C#

- Klassisches Threadmodell
- Parallel Extensions
 - Task Parallel Library (TPL)
 - Tasks
 - Parallel For / Do
 - Coordination Data Structures (CDS)
 - Parallel LINQ (PLINQ)

Multithreading mit C# - Praxis

- Problem:
 - Summe aller Primzahlen unter 1.000.000 berechnen

```
private static long SumPrimesBelowSeq(int number) {  
    long sum = 0;  
    for (int i = 0; i < number; i++) {  
        if (isPrime(i))  
            sum += i;  
    }  
    return sum;  
}
```

- Wie parallelisiert man diesen Code?
- Wie unterscheiden sich die Varianten?

KLASSISCHES VORGEHEN

Threads

Klassisches Threadmodell

Aufgabe	C# Code
Neuen Thread erzeugen	<pre>Thread thread = new Thread(delegate() { // Funktionsrumpf })</pre>
Thread starten	<pre>thread.Start()</pre>
Warten bis Thread terminiert	<pre>thread.Join()</pre>
Synchronisation	<pre>lock(Variable) { // Block } Interlocked.Add, Interlocked.Increment</pre>

Lösung mit Threads

```
int threadCount = 4; // Festlegen der Threadanzahl
int numPerThr = number / threadCount;
Thread[] threads = new Thread[threadCount];

for (int i = 0; i < threadCount; i++) {
    threads[i] = new Thread(delegate() {
        for (int num = i*numPerThr; num < (i+1)*numPerThr; num++) {
            if (isPrime(num))
                Interlocked.Add(ref sum, 1); // Threadsicheres Addieren
        }
    });
    threads[i].Start();
}

for (int i = 0; i < threadCount; i++) {
    threads[i].Join(); // Warten auf Ende aller Threads
}
```

Probleme mit Threads

- Manuelles Festlegen der Threadanzahl
- Addieren muss thread-sicher erfolgen
- Warten bis alle Threads terminieren
- Keine dynamische Arbeitsteilung
- Erzeugen von Threads verursacht Overhead!

PARALLEL EXTENSIONS

Task Parallel Library

TPL - Tasks

- „Leichtgewichtige Threads“
- Tasks werden von Taskmanager verwaltet
 - Verteilt Tasks auf Threads in einem Thread-Pool

Aufgabe	C# Code
Neuen Task erzeugen	<pre>Task<int> task = new Task<int>(delegate(){ // Funktionsrumpf })</pre>
Task starten	<pre>task.Start()</pre>
Wert zurückgeben (Blockierender Aufruf)	<pre>task.Result</pre>
Warten auf Beenden von mehreren Tasks	<pre>Task.WaitAll(Task[] tasks)</pre>

Tasks - Lösung

```
int threadCount = 100; // Anzahl der Tasks festlegen
int numPerThr = number / threadCount;
Task<long>[] tasks = new Task<long>[threadCount];

for (int i = 0; i < threadCount; i++) {
    tasks[i] = new Task<long>(delegate() { // Tasks erzeugen
        long sum = 0;
        for (int num = i*numPerThr; num < (i+1)*numPerThr; num++) {
            if (isPrime(num))
                sum += num;
        }
        return sum;
    });
    tasks[i].Start();
}

Task.WaitAll(tasks); // warten bis alle Tasks terminiert sind
for (int i = 0; i < threadCount; i++) {
    totalSum += tasks[i].Result; // Summe berechnen
}
```

TPL – Parallel For / Parallel Do

- Bauen auf Tasks auf
- Parallel for

```
Parallel.For(int from, int to, delegate(int i) {  
    // Funktionsrumpf  
});
```

- Parallel do
 - Paralleles Ausführen von mehreren Anweisungen
 - Wartet auf Beenden aller Anweisungen

```
Parallel.Do(  
    () => foo();  
    () => bar();  
);
```

Parallel For Lösung

```
private static long SumPrimesBelow(int number) {  
    long sum = 0;  
    Parallel.For(0, number, delegate(int i) {  
        if (isPrime(i)) {  
            Interlocked.Add(ref sum, i);  
        }  
    });  
    return sum;  
}
```

- Anzahl erzeugter Worker-Threads: $(\text{\#Kerne} * 2) - 1$
- Optimierungspotential der bisherigen Parallel For Lösung
 - Zwischensummen mit ThreadLocalState

PARALLEL EXTENSIONS

Coordination Data Structures

Coordination Data Structures

- CDS besteht aus 2 Komponenten:
 - Thread-sichere Datenstrukturen
 - Synchronisations-Primitiven
- Einsatz:
 - vorhandene Funktionalität reicht nicht aus
 - Optimierung von einzelnen Abschnitten
- Thread-sichere Datenstrukturen
 - BlockingCollection
 - Producer / Consumer Pattern mit Add() und Take()
 - Stack, Queue, Dictionary, ...

Coordination Data Structures

- Synchronisations-Primitiven
 - SpinWait / SpinLock
 - Aktives Warten
 - WriteOnce
 - Einmaliges Setzen einer Variable
 - LazyInit
 - Initialisierung des Objekts beim erstmaligen Gebrauch
 - CountdownEvent
 - Nützlich für Fork / Join Szenarien

PARALLEL EXTENSIONS

Parallel LINQ

LINQ

- Language Integrated Query
 - SQL-ähnliche Abfragen auf Datenstrukturen

```
long SumPrimesBelowLINQ(int to) {  
    // Primzahlen auflisten  
    var primes = from number in Enumerable.Range(0, to)  
                  where isPrime(number)  
                  select (long) number;  
    // Summe bilden  
    return primes.Sum();  
}
```

Die Abfrage wird erst beim Aufruf von Sum() ausgeführt!

Parallel LINQ

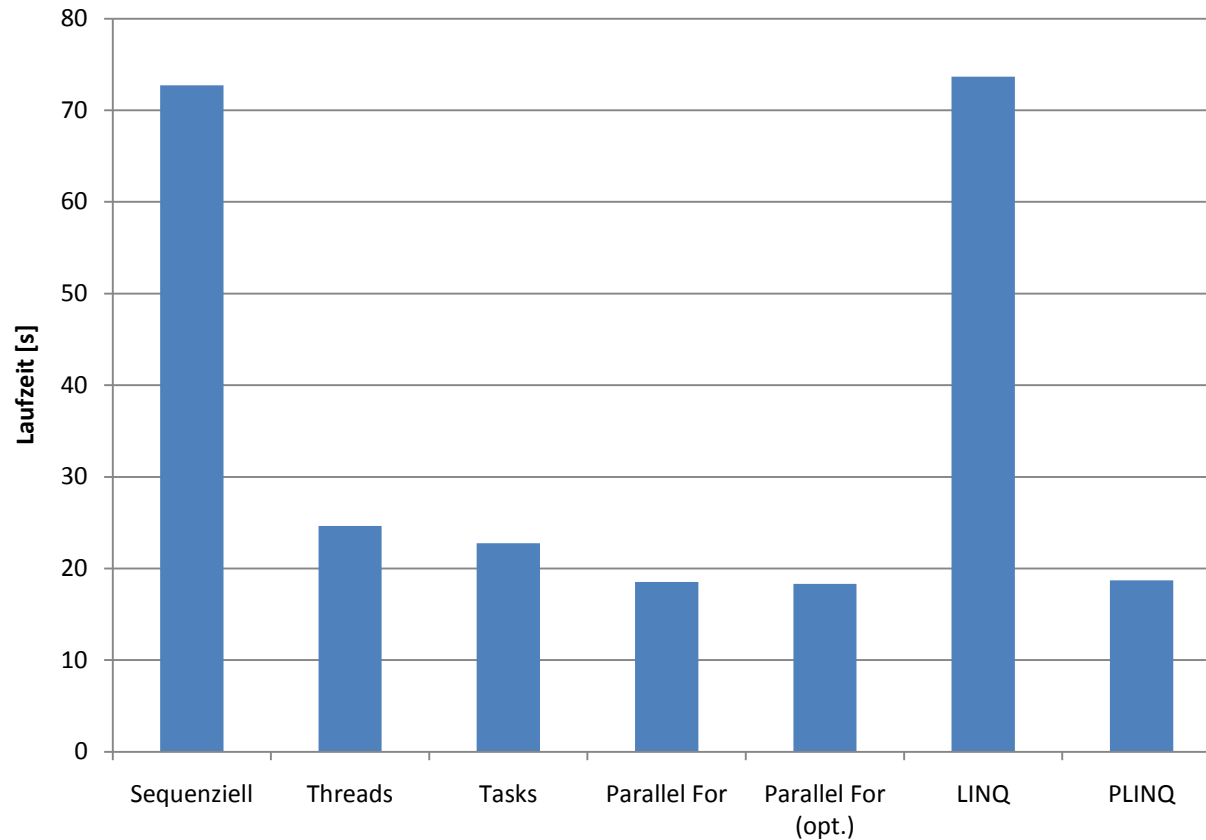
- Automatisierte Parallelisierung der Abfragen
 - Durch Anfügen von AsParallel()
 - Framework kümmert sich um Parallelisierung
 - Reihenfolge der zurückgelieferten Elemente nicht deterministisch!

```
long SumPrimesBelowPLINQ(int to) {  
    // Primzahlen auflisten  
    var primes = from number in Enumerable.Range(0, to).AsParallel()  
                 where isPrime(number)  
                 select (long) number;  
    // Summe bilden  
    return primes.Sum();  
}
```

MULTITHREADING MIT C#

Vergleich und Best Practices

Performance-Vergleich



Parallel For mit Zwischensummen am schnellsten!

Best Practices

- Klassische Threads sollten vermieden werden
- Parallel For / Do
 - Aufgaben im gleichen lexikalischen Scope
- Tasks
 - Aufgaben die nicht geeignet sind für Parallel For / Do
- Coordination Data Structures
 - Implementierung von Multithreading Patterns
 - Low-level Multithreading
- PLINQ
 - Automatisierte Parallelisierung von LINQ Abfragen

ACTOR-MODEL IN SCALA

Multi-Threading mal anders

„An actor is a thread-like entity that has a mailbox for receiving messages.“

aus Programming in Scala

Was sind Actors

- Grundlegendes mathematisches Modell beschreibt
 - Primitive of concurrent digital computation
 - Kommunikation mittels Message-Passing
- Zustandsänderungen sollen nur über Nachrichten erfolgen
- Reihenfolge der eintreffenden Nachrichten darf keine Auswirkung haben!

Definition

- Ein Actor ist eine Verarbeitungseinheit, die beim Erhalt einer Nachricht, parallel...
 - eine endliche Anzahl von Nachrichten an andere Actors schicken kann
 - eine endliche Anzahl neuer Actors erzeugen kann
 - das Verhalten beim Erhalt der nächsten Nachricht bestimmen kann

Alles ist ein Actor

- Ein Actor kapselt Zustand und Verhalten...
 - Und ähnelt damit mehr dem ursprünglichen OOP Konzept als Klassen
- Kein Shared Memory
 - Daher gibt es auch nichts zu synchronisieren
 - Fördert die Vermeidung von Race Conditions
- Erlang hat die am meisten bekannteste Actor Implementierung (rein Event basiert)

ACTORS AUF DER JVM!

Scala

- Funktional
 - Pattern-Matching
 - Sehr guter Support für Listen
 - Tail-Recursive
- Objektorientiert
- Statisch typisiert
- Nativer XML Support
- Scala Actors bilden Erlangs Parallelitätskonzept nach

WIE SIEHT SCALA CODE AUS?

```
val seq = List(1,2,3,4,5,6,7,8,9) map ( number =>  
number*number )
```

```
seq: List[Int] = List(1, 4, 9, 16, 25, 36, 49, 64, 81)
```

Actors in Scala

- Entweder nativer Hardware-Thread, oder
- Leichtgewichtiger Event-“Thread”
- Können das volle Potential von Scala nutzen!
 - Pattern-Matching...
 - Case-Classes...
 - Tuples...
 - !, !!, !? Operatoren...
 - erleichtern die Arbeit mit Actors ungemein

Hello World Actor

```
val helloWorldActor = actor {  
  receive {  
    case 'hello => println ("Hello World")  
  }  
}
```

```
helloWorldActor ! 'hello
```

```
> Hello World
```

Case Classes & Pattern Matching

- Grundbausteine der Kommunikation für Scala Actors

```
case class Message (attr1:String, attr2:String)
case class Exit
react {
  case Message (a1, a2) => println (a1+" "+a2)
  case Exit => System.exit (0)
  case _ => println ("Unkown message")
}
```

Futures

- Konzept zur „Lazy Evaluation“ parallel berechneter Ergebniswerte
- Beispiel:
val awaitResult = actor !! ProcessData (data)
...
val result = awaitResult ()
- Kann geprüft werden mittels awaitResult.isSet()
- Gut geeignet zum Joinen verteilter Berechnungen

```
case class SearchTrip (val trip :(specification.Place, specification.Place, Date))
case class SearchTripResponse (val flights:Seq[specification.Itinerary])
```

```
object ParallelMapper extends Mapper {
```

```
  def map (trips: Seq[(specification.Place, specification.Place, Date)]) = trips.map (
    singleTrip => {
      actor {
        react {
          case SearchTrip(trip) =>
            reply (SearchTripResponse(searchOneway
              (trip._1, trip._2,trip._3)));
        }
      } !! SearchTrip (singleTrip)
    }).map (future => {
      val ft = future ().asInstanceOf[SearchTripResponse]
      ft.flights
    })
  )
}
```

SCALA ACTOR BEISPIEL

Paralleler Merge-Sort

```
case class RegisterWorker (worker:Worker)
case class UnregisterWorker (worker:Worker)
case class SortList (data:List[Int])
case class SortListResponse (data:List[Int])
case class Stop
```

```
class Worker extends Actor{
  def act = loop {
    react {
      case SortList (data) =>
        reply (SortListResponse (data.sort {(a,b) => a < b}))
      case _ => 0
    }
  }
  override def start = {
    Master ! RegisterWorker (this)
    super.start ()
  }
}
```

```
object Master extends Actor {  
  private var workers = List [Worker]()  
  def act = loop {  
    react {  
      case RegisterWorker (newWorker) => workers = newWorker :: workers  
      case SortList (data:List[Int]) => {  
        val futures = MasterHelper.distributeWork (workers, data, data.size/workers.size)  
        while (futures.count(_._isSet == false) > 0){}  
        val responses = futures.map ( x => {  
          val res = x ()  
          (res.asInstanceOf[SortListResponse]).data  
        })  
        reply (SortListResponse (MergeSortHelper.mergeAll (responses.head, responses.tail)))  
      }  
      case Stop => System.exit (0)  
    }  
  }  
}
```



```
val rnd = new Random
```

```
val lst = new ListBuffer[Int] ()
```

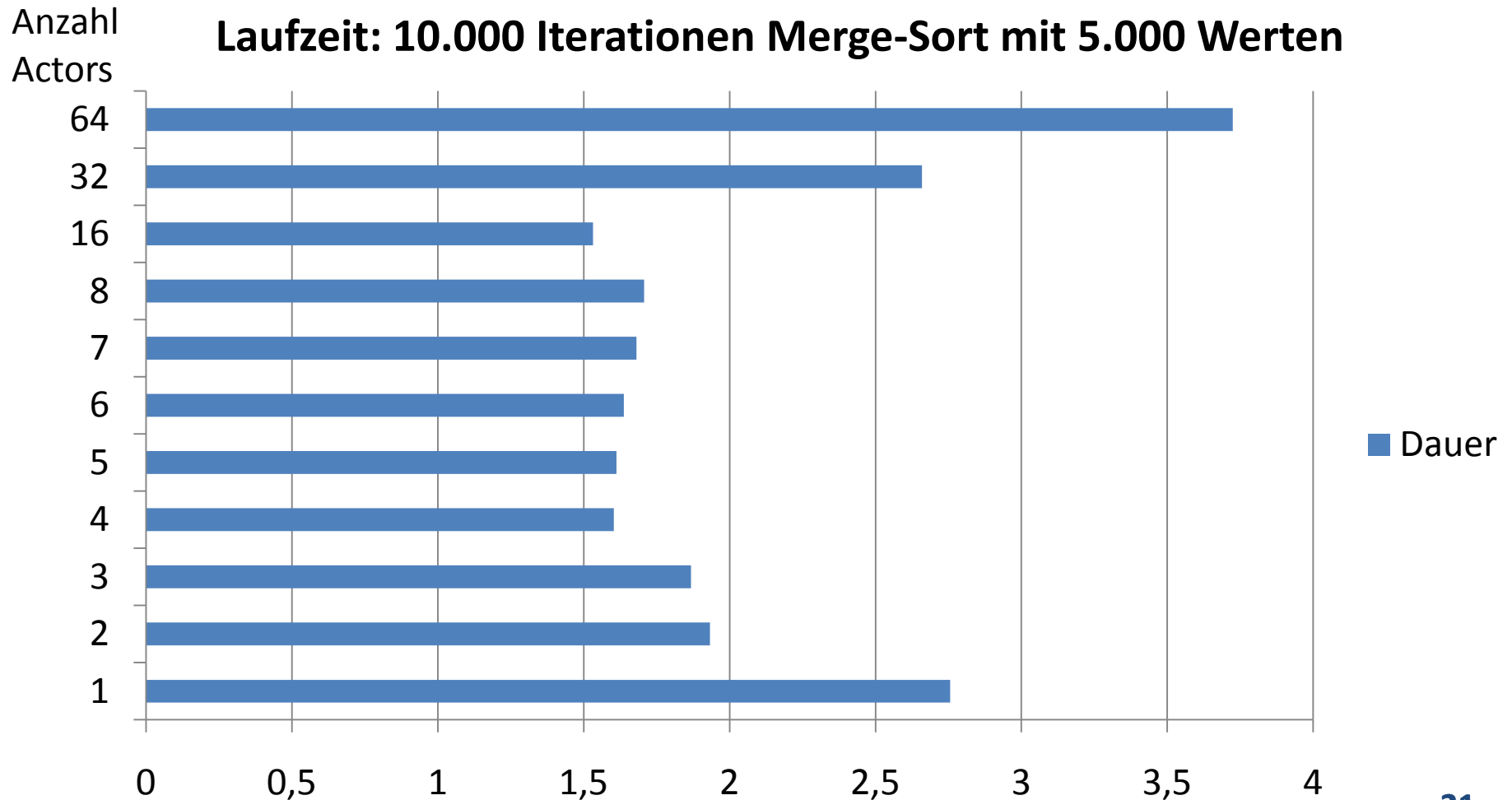
```
for (i <- 0 to 5000) lst.append (rnd.nextInt(5000))
```

```
Master !? SortList (lst.toList) match {
```

```
  case SortListResponse(list) => println ("Sortierte Liste: " + list)
```

```
}
```

Ergebnisse



Unterschied zwischen react und receive

- receive
 - Spezifiziert direkt einen Hardware-Thread
- react
 - nutzt einen Shared-Pool an Threads
 - Leichtgewichtige Verarbeitung
 - Angelehnt an ein Nachrichten-System
 - Dank dynamischer Auslastung bessere Nutzung der Ressourcen

Kommende Neuerungen in Scala 2.8

- Neuer Trait: Reactor
 - Leichtgewichtiger als Actor
 - Kein impliziter sender mehr (-> ReplyActor!)
 - Nurnoch Event basiert verfügbar (react)
 - Weniger Zustände werden gespeichert
 - Keine !! Oder !? Methoden mehr verfügbar
 - Vorteil: Ermöglicht massiv höhere Anzahl an instantiierten Reactors als Actors

Weitere Actors-Implementierungen

- Java
 - Akka Actors, Kilim, Actors Foundry, Actors Guild
- Groovy
 - GParallelizer
- Scala
 - Scala Actors, Akka Actors
- C++
 - Theron, ACT++

Fazit – Actors

- + Sehr gute Skalierbarkeit
- + kein Shared Memory → keine Locks
- + Event- und Messaging-System “Beigabe”
- + Effizientere Nutzung von Systemressourcen
- - Stellt bisheriges Programmiermodell auf den Kopf
- - Kein guter Support in etablierten Programmiersprachen
- Actors und ähnliche Nachrichtenbasierte Ansätze für parallele Systeme werden an Popularität gewinnen

Zukünftige Entwicklung?

Vielen Dank!

Literatur

QUELLEN

Einführung & Actors

<http://www.golem.de/0912/71893.html>

http://spiele.t-online.de/us-armee-forscht-mit-playstation-3/id_20849028/index

<http://ruben.savanne.be/articles/concurrency-in-erlang-scala>

<http://artisans-serverintellect-com.si-eioswww6.com/default.asp?W1>

<http://www.javaworld.com/javaworld/jw-03-2009/jw-03-actor-concurrency2.html>

<http://www.javaworld.com/javaworld/jw-02-2009/jw-02-actor-concurrency1.html>

<http://www.clickcaster.com/channel/tag/oop?channel=diveintoerlang>

<http://youshottheinvisibleswordsman.co.uk/2009/04/01/remotefactor-in-scala/>

Einführung & Actors

<http://www.heise.de/newsticker/meldung/Ausblick-auf-Intels-Hexa-Core-Prozessoren-921824.html>

http://www.ibm.com/developerworks/java/library/j-scala02049.html?ca=dgr-twtrScala-concurrencydth-JV&S_TACT=105AGY83&S_CMP=TWDW

<http://www.flickr.com/photos/elitepete/246903518/sizes/l/>

<http://www.flickr.com/photos/pphotography/4016777686/sizes/o/>

<http://jonasboner.com/2007/12/19/hotswap-code-using-scala-and-actors.html>

<http://www.jonasboner.com/2008/01/25/clustering-scala-actors-with-terracotta.html>

<http://www.scala-lang.org/node/2041>

<http://permalink.gmane.org/gmane.comp.lang.scala.user/23521>

Methodiken & Automatisierung

<http://www.mi.fh-wiesbaden.de/~barth/fsbsc/ss09/ParallelDecomposition.pdf>

<http://www.informatik.uni-ulm.de/ni/Lehre/SS04/HPC/HPCauto2.pdf>

<http://www.1001ausmalbilder.de/Fahrzeuge/Zug/Ausmalbild-Zug-Lokomotive-41.html>

<http://de.dreamstime.com/stockfotos-internet-homepage-symbol-ausf-uumlhrliche-ikone-des-blauen-hauses-image1747873>

<http://www.tour-blog.de/2006/11/Reca/Lager.jpg>

http://www.lokifahrer.ch/Signale/signale_der_vergangenheit.htm

http://upload.wikimedia.org/wikipedia/en/f/f1/Fork_join.svg

http://www4.informatik.uni-erlangen.de/Lehre/SS04/V_SOS1/Skript/SOS1-08-A5.pdf

<http://www.mathematik.uni-marburg.de/~loogen/Lehre/ws08/ParProg/Folien/ParProg6a.ppt>

<http://capp.itec.kit.edu/teaching/rs/ss08/zu/3-uebung.pdf>

Bengel G. u. a., Masterkurs Parallele und Verteilte Systeme, Vieweg+Teubner 2008

Tanenbaum A., Moderne Betriebssysteme, Pearson Studium 2002

Java

Deadlock Comic von <http://www.flickr.com/photos/mjswart/3763969995/>

GOETZ, Brian – **Java Concurrency in Practice**

ISBN 978-0-321-34960-6

LEA, Doug – **Concurrent Programming in Java**

ISBN 978-0-201-31009-2

MAGEE, Jeff; KRAMER, Jeff – **Concurrency: State Models and Java Programs**

ISBN 978-0-470-09355-9

BLOCH, Josh – **Effective Java: A Programming Language Guide**

ISBN 978-0-321-35668-0

Probleme Multithreading & C#

- Parallel Computing in .NET 4.0 – Overview
 - <http://codingndesign.com/blog/?p=43>
- Optimize Managed Code For Multi-Core Machines
 - <http://msdn.microsoft.com/en-us/magazine/cc163340.aspx#S6>
- Parallel Programming with .NET
 - <http://blogs.msdn.com/pfxteam/>
- The multi-threaded brain
 - <http://sandeepnade.blogspot.com/2005/10/multi-threaded-brain.html>
- .NET Multithreading
 - Alan Dennis, Manning 2002
- Advantages and Disadvantages of a Multithreaded/Multicontexted Application
 - http://download.oracle.com/docs/cd/E13203_01/tuxedo/tux71/html/pgthr5.htm