Parallele und funktionale Programmierung (6)

Prof. Dr. Michael Philippsen

Gliederung

- *Teil II – Anwendungstypen und Effizienzfragen*

6. Task-paralleles Vorgehen (1)

   Einfache Task-Abhängigkeiten

   Klient & Dienstleister
   - Web-Server

   Chef & Arbeiter
   - Seitenanzeige im Web-Browser
   - Monte-Carlo-Simulation

7. Task-paralleles Vorgehen (2)
8. Datenparalleles Vorgehen (1)
9. Datenparalleles Vorgehen (2)

   Deklarativer Ansatz; MapReduce

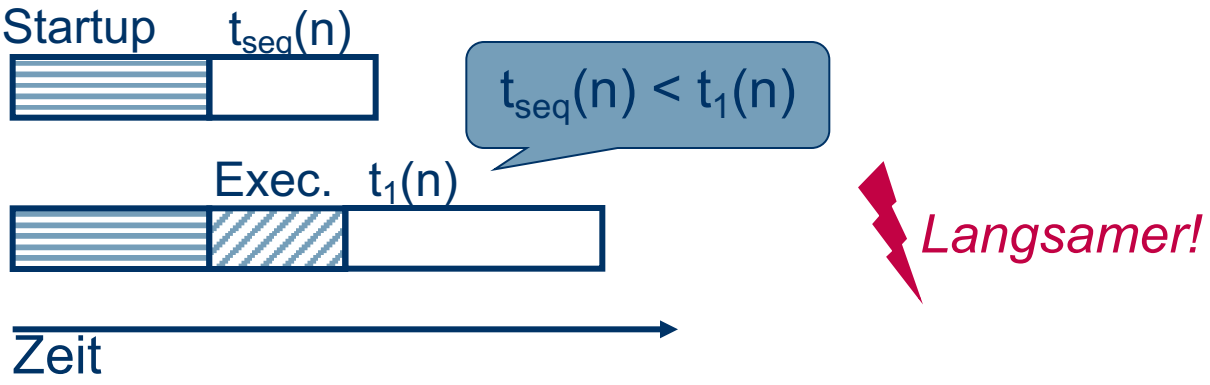
Granularität;
Mindestproblemgröße;
Lastbalance;
Arbeitspaketgröße;
Speedup; Amdahl

Wie viel Nebenläufigkeit tut gut?

- Bei der Organisation einer Anwendung in parallel ausführbare Arbeitseinheiten wird man oft zur Übertreibung verleitet.
- Es ist nicht sinnvoll, die Problemlösung bis zu massenhaften Arbeitsschritten minimaler Größe zu zersplittern.
- Wichtige Aspekte:
  - *Problemgröße:* parallele Ausführung verursacht Fixkosten. Lohnt sich die parallele Ausführung überhaupt?
  - *Granularität:* Je mehr feingranulare Aktivitätsfäden man hat, desto häufiger blockieren diese sich an kritischen Abschnitten. Diese Blockierung verlangsamt die Ausführungszeit.
  - *Kernanzahl:* Mehrere Aktivitätsfäden auf einem Prozessorkern laufen nur pseudoparallel. Das kann nur schneller sein, wenn (E/A- und sonstige) Wartezeiten ausgenutzt werden können.

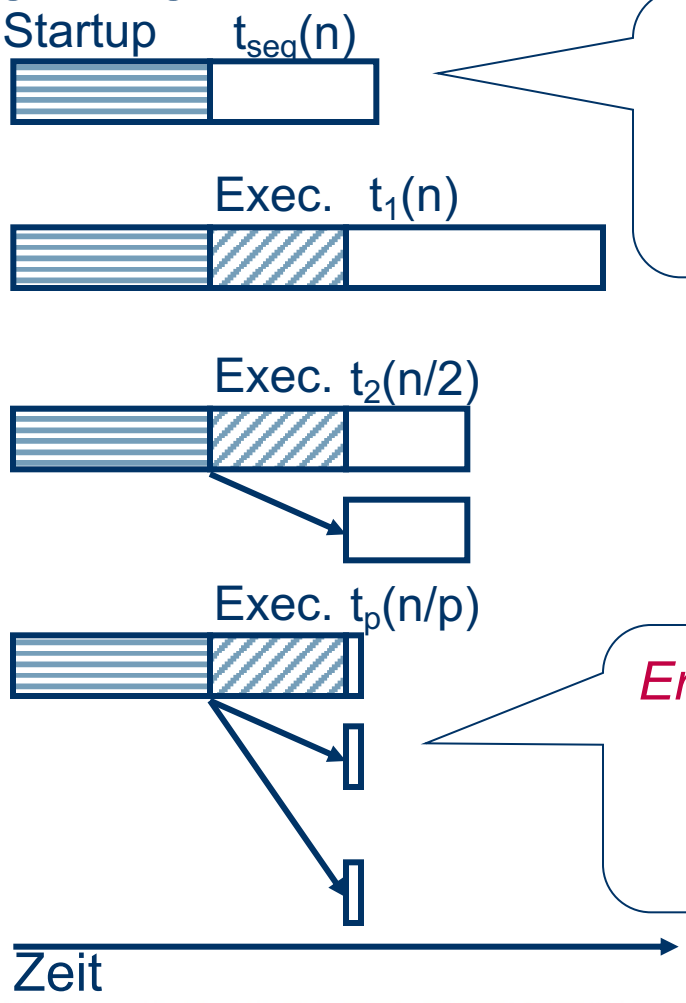
Grenze: Problemgröße paralleler Anwendungen (1)

- Jede parallel ausgeführte Arbeitseinheit muss verwaltet werden. Das bedingt:
  - die Erzeugung und spätere Vernichtung von Objekten,
  - Kosten für die Verwaltungsdatenstrukturen (z.B. **`Executor`**),
  - die Blockierungsverwaltung,
  - Verlangsamung durch Sichtbarkeitsregeln (keine Verwendung von Registern, Maßnahmen zur Cache-Konsistenz, …)
  - …
- Grafisch:
  - **`main`**:
  - 1 **`Thread`**:



Grenze: Problemgröße paralleler Anwendungen (2)

- Die Fixkosten der **`Thread`**-Verwendung amortisieren sich nur bei genügend großen Problemen:
  - **`main`**:
  - 1 **`Thread`**:
  - 2 **`Threads`** auf 2 Kernen:
  - p **`Threads`** auf p Kernen:

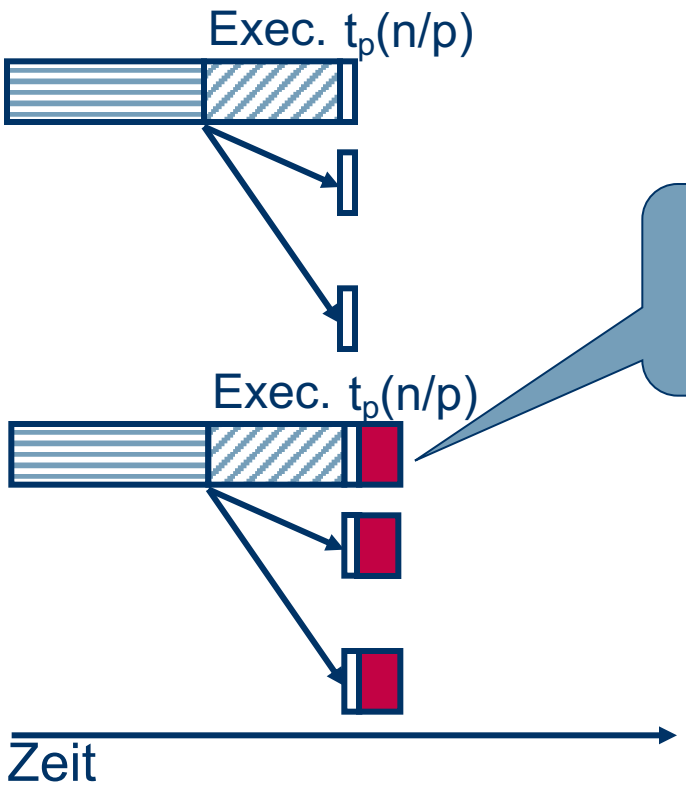


Eine noch kleinere Problemgröße n kann nicht lohnend parallel bearbeitet werden.

Erst bei genügend hohem Parallelitätsgrad lohnt sich die parallele Ausführung!

Grenze: Granularität paralleler Anwendungen

- Je höher der Parallelitätsgrad, desto mehr müssen die Aktivitätsfäden sich im Allg. synchronisieren und desto häufiger blockieren sie beim Zugriff auf gemeinsame Daten.
- Im Allg. gilt: $t_p(n/p)$ wächst bei wachsendem p.
  - Ideal:
  - Oft:



Synchronisationen führen zu einer Verlangsamung.

Zu hoher Parallelitätsgrad verlängert oft die Laufzeiten!

Grenze: Pseudoparallelitätsgewinne

- Aktivitätsfäden warten regelmäßig auf Ressourcen. Diese Wartezeit kann genutzt werden.
  - 1 **`Thread`**:

Wartezeit

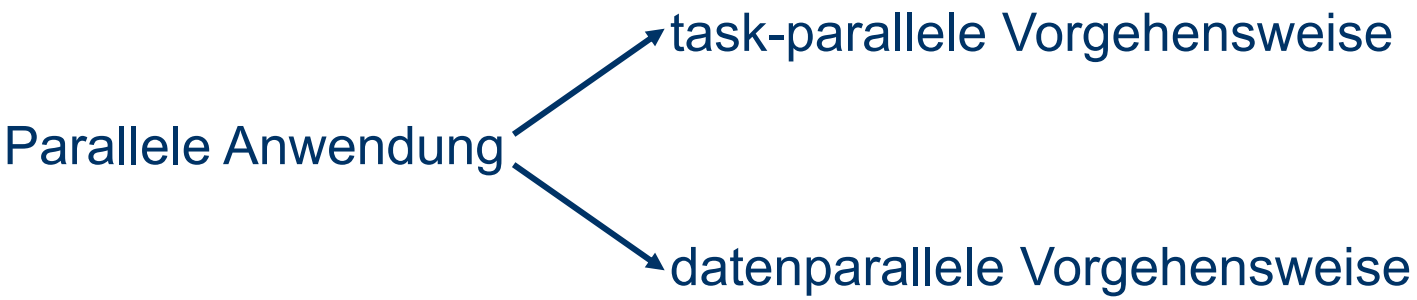
  - 2 **`Threads`** auf 1 Kern:

Synchronisation führt zu einer Verlangsamung.

Pseudoparallelitätsgewinne treten nur bei moderatem Parallelitätsgrad auf.

Muster für parallele Anwendungen (1)

- Zwei orthogonale Vorgehensweisen:
  - Gliederung der Problemlösung in Arbeitspakete, die erledigt werden müssen („task-parallel“).
    - Was muss alles gemacht werden, um das Problem zu lösen.
  - Gliederung der Problemlösung nach den Datenstrukturen, datenparallel („data-parallel“).
    - Es gibt viele Daten, auf denen ähnliche Operationen ausgeführt werden müssen, um das Problem zu lösen.



Muster für parallele Anwendungen (2)

- Es sind immer zwei Seiten derselben Medaille, die sich gegenseitig bedingen.
  - Bei einer task-parallelen Lösung fließen Daten zwischen Arbeitspaketen, bzw. Aktivitätsfäden greifen gemeinsam auf Daten zu.
  - Bei einer datenparallelen Lösung hat man Aktivitätsfäden, die sich von Zeit zu Zeit zwischen einzelnen Arbeitsphasen synchronisieren müssen.

Task-Abhängigkeitsgraph

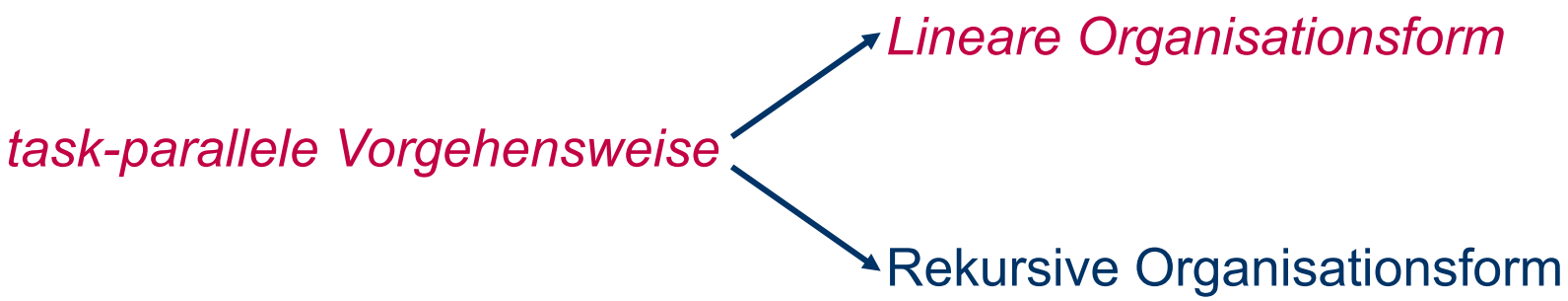
- Ein Ansatz, eine parallele Anwendung zu organisieren, besteht darin, die zu verrichtende Arbeit in Arbeitspakete („tasks“) zu gruppieren.
  - Abgeschlossene, diskrete Einheiten.
  - Unabhängig von Seiteneffekten durch die Abarbeitung anderer Arbeitspakete.
  - Groß genug, um den Verwaltungsaufwand zu rechtfertigen, der mit dem Arbeitspaket und dessen Ausführung verbunden ist.
- Ein Arbeitspaket A
  - kann weitere Arbeitspakete B erzeugen.

Wenn A „schaltet“, wird Voraussetzung für B geschaffen.

  - kann vom Ergebnis eines anderen Arbeitspakets C abhängen.

Das „Schalten“ von C produziert das Ergebnis, das A benötigt.

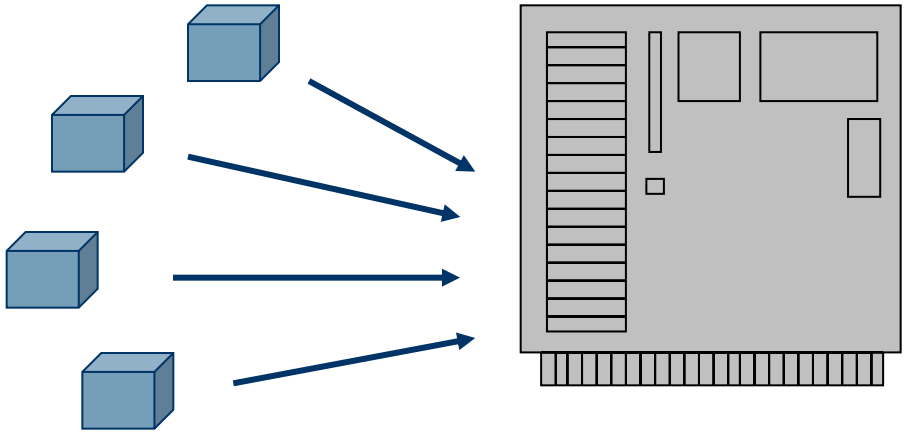
Muster für task-parallele Anwendungen



- Lineare Organisationsform
  - Weitgehend unabhängige Arbeitspakete
    - *→Klient & Dienstleister, „client & server“*
    - Chef & Arbeiter, „master & worker“
      - Arbeitsdiebstahl, „work stealing“
  - Einfache Abhängigkeit (gemeinsame Daten, Reihenfolge)
    - Fließband, Produzent & Konsument, „producer & consumer“
- Rekursive Organisationsform
  - Paralleles teile-und-herrsche

Klient & Dienstleister, „client & server“

- Beispiel: Viele unabhängige Anfragen werden an einen Web-Server gerichtet.
- Klassische Frage für task-paralleles Vorgehen.
- Im Task-Abhängigkeitsgraphen:
  - Jede Anfrage ist ein Arbeitspaket, das von anderen Arbeitspaketen unabhängig ist.
  - Die Bearbeitung ist die Anwendung einer *seiteneffektfreien Funktion* auf die Daten des Arbeitspakets.
  - Synchronisationsaspekte können weitgehend von der Bearbeitungsfunktion getrennt werden.



Unabhängige Arbeitspakete

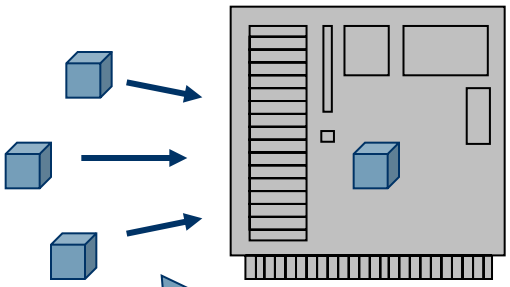
Server

Beispiel Web-Server (1)

- Stellen Sie sich einen einfachen Web-Server vor:

```
class SingleThreadedWebServer {
   public static void main(...) ... {
      ServerSocket socket = new ServerSocket(80);
      while (true) {
         Socket connection = socket.accept();
         handleRequest(connection);
      }
   }
}
```

Anfrage = Arbeitspaket



Warten vor Server

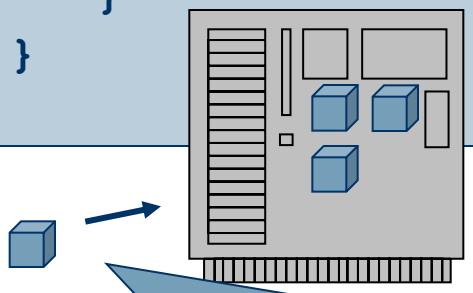
So kann nur eine Anfrage zu einer Zeit beantwortet werden. Nur sinnvoll, wenn die Anfragen sehr selten ankommen.
Wegen der EA-Operationen der Anfragebearbeitung kann selbst mit einem Prozessor mehr Leistung erzielt werden, wenn **`Thread`**-Objekte verwendet werden.
…

Beispiel Web-Server (2)

- Mit einem **`Thread`**-Objekt pro Arbeitspaket:

```
class ThreadPerTaskWebServer {
   public static void main(...) ... {
      ServerSocket socket = new ServerSocket(80);
      while (true) {
         final Socket connection = socket.accept();
         Runnable task = new Runnable() {
               public void run() {
                  handleRequest(connection);
               }
            };
         new Thread(task).start();
      }
   }
}
```

Bearbeitung muss „**`thread`**-sicher“ sein.

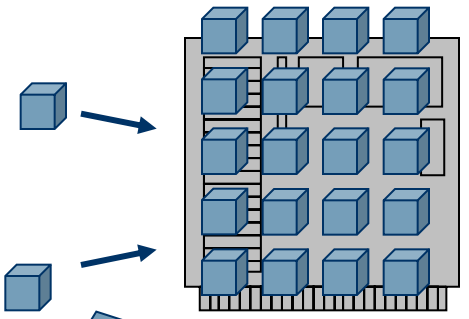


wird sofort angenommen

Sobald die Bearbeitung einer Anfrage in ein **`Thread`**-Objekt ausgelagert wurde, wartet der Server auf den nächsten Auftrag.

Beispiel Web-Server (3)

- Solange die Rate der Anfragen die Kapazität des Servers zu deren Bearbeitung nicht übersteigt, funktioniert die Ein-**`Thread`**-pro-Arbeitspaket-Lösung gut.
- Nachteile der Ein-**`Thread`**-Lösung:
  - Erzeugung und Entfernung der **`Thread`**-Objekte ist nicht umsonst.
  - Jedem Aktivitätsfaden werden zusätzliche Ressourcen (vor allem Hauptspeicher für den Methoden-stapel) zugewiesen.
  - Verknappung produziert zusätzliche Systemlast (z.B. muss der Speicher-bereiniger häufiger laufen).
  - Bei zu vielen Anfragen, kann es zu Abbrüchen kommen (z.B. **`OutOfMemoryError`**).



Noch mehr Aufträge und der Server „platzt“… Grenze: Pseudo-parallelitätsgewinne

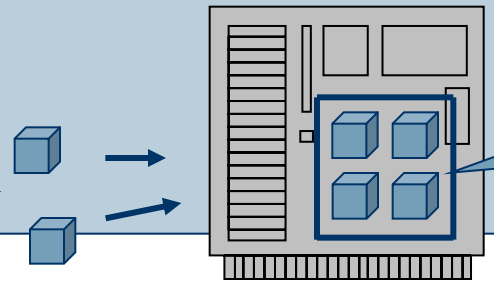
Beispiel Web-Server (4)

- Mit einem **`Thread`**-Pool:

```
class TaskExecutionWebServer {
   private static final int NTHREADS = 100;
   private static final Executor exec =
      Executors.newFixedThreadPool(NTHREADS);
   public static void main(...) ... {
      ServerSocket socket = new ServerSocket(80);
      while (true) {
         final Socket connection = socket.accept();
         Runnable task = new Runnable() {
               public void run() {
                  handleRequest(connection);
               }
            };
         exec.execute(task);
      }
   }
}
```

Bis zu 100 **`Threads`** führen Anfragen aus.

Was geschieht mit zusätzlichen Aufträgen?



Maximalzahl an **`Threads`**.

Interface `BlockingQueue<E>` (1)

- Ein Executor verwendet eine **`BlockingQueue<E>`** zur Verwaltung der anstehenden Arbeitsaufträge.
- **`java.util.concurrent`** bietet passende Klassen:
  - **`LinkedBlockingQueue<E>`**
    - FIFO-Verhalten
    - Maximallänge begrenzt oder unbegrenzt (= default für **`newFixedThreadPool`** und **`newSingleThreadExecutor`**).
  - **`ArrayBlockingQueue<E>`**
    - Als Array realisiert, zyklisch rotierend befüllt.
  - **`PriorityBlockingQueue<E>`**
  - **`SynchronousQueue<E>`**
    - Nur wenn **`Thread`** bereits auf Arbeitspaket wartet (oder ein solcher noch erzeugt werden kann), wird das Arbeitspaket angenommen.
  - …

Schlauer für Web-Server wg. Denial-of-Service-Attacken.

Interface `BlockingQueue<E>` (2)

- Operationen:
  - **`void put(E)`**
    - fügt ein Element **`E`** in die Schlange ein,
    - blockiert, falls die Schlange voll ist; setzt fort, sobald Platz frei ist.
  - **`boolean offer(E)`**
    - fügt ein Element **`E`** in die Schlange ein, liefert dann **`true`**
    - liefert sofort **`false`**, falls die Schlange voll ist.
  - **`E take()`**
    - liefert **`E`** aus Schlange,
    - blockiert, falls die Schlange leer ist; setzt fort, sobald E vorhanden.
  - **`E poll()`**
    - liefert **`E`** aus Schlange, falls E vorhanden,
    - liefert sonst sofort **`null`**.

Interface `RejectedExecutionHandler`

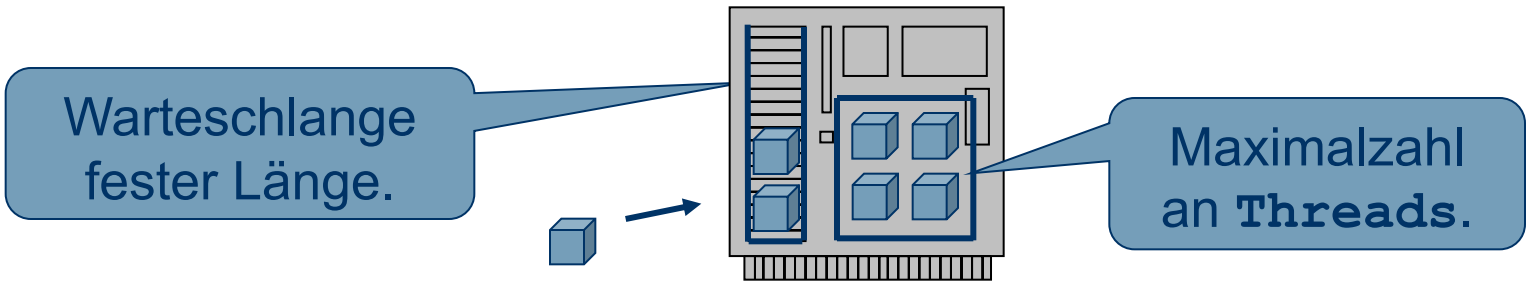
- Der Executor kann festlegen, was mit Arbeitsaufträgen, die von Warteschlangen mit begrenzter Länge nicht aufgenommen werden können, geschehen soll.
  - **`AbortPolicy`**
    - Man erhält bei **`submit`** eine **`RejectedExecutionException`**.
    - Das ist die Standardeinstellung.
  - **`DiscardPolicy`**
    - Neues Arbeitspaket wird stillschweigend ignoriert.
  - **`DiscardOldestPolicy`**
  - **`CallerRunsPolicy`**
    - Der Aktivitätsfaden, der **`submit`** aufgerufen hat, führt selbst das Arbeitspaket aus und kehrt erst danach vom **`submit`**-Aufruf zurück.

Beispiel Web-Server (5)

- Mit einem speziellen Executor z. B.:

```
private static final int NTHREADS = 100;
private static final int CAPACITY = 100;
private static final Executor exec =
    new ThreadPoolExecutor(
        NTHREADS/2, // übliche Pool-Größe
        NTHREADS, // maximale Pool-Größe
        1L,
        TimeUnit.MINUTES,
        new LinkedBlockingQueue<Runnable>(CAPACITY),
        new ThreadPoolExecutor.CallerRunsPolicy());
```

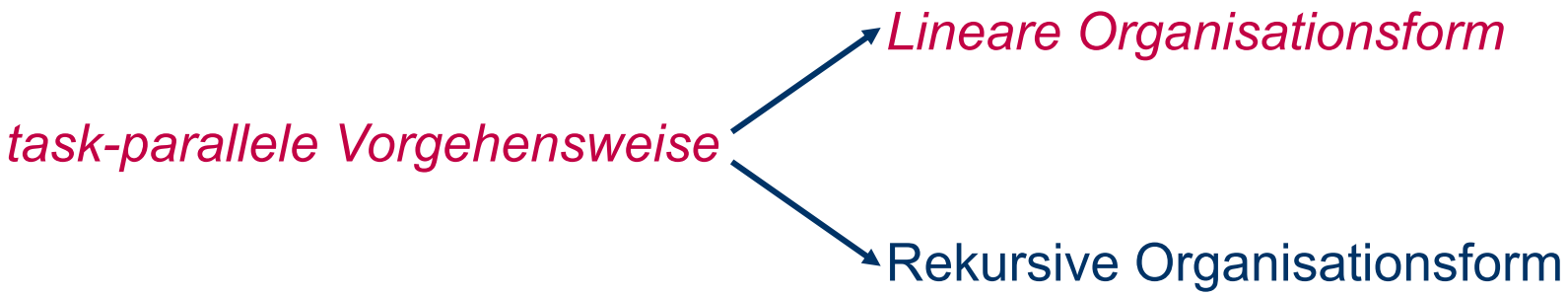
Falls es mehr **Threads** im Pool gibt als die übliche Pool-Größe vorschreibt, dann werden diese so lange vorgehalten, ehe sie entfernt werden.



Klient & Dienstleister: weitere Beispiele

- Datenbank-Server bearbeiten nebenläufig eingehende Anfragen.
- Restaurant: Kellner, Küche, Koch (sequentiell, pseudo-parallel) bzw. Köche (parallel). Synchronisation beim Zugriff auf Töpfe, Herdplatten, …

Muster für task-parallele Anwendungen



- Lineare Organisationsform
  - Weitgehend unabhängige Arbeitspakete
    - Klient & Dienstleister, „client & server“

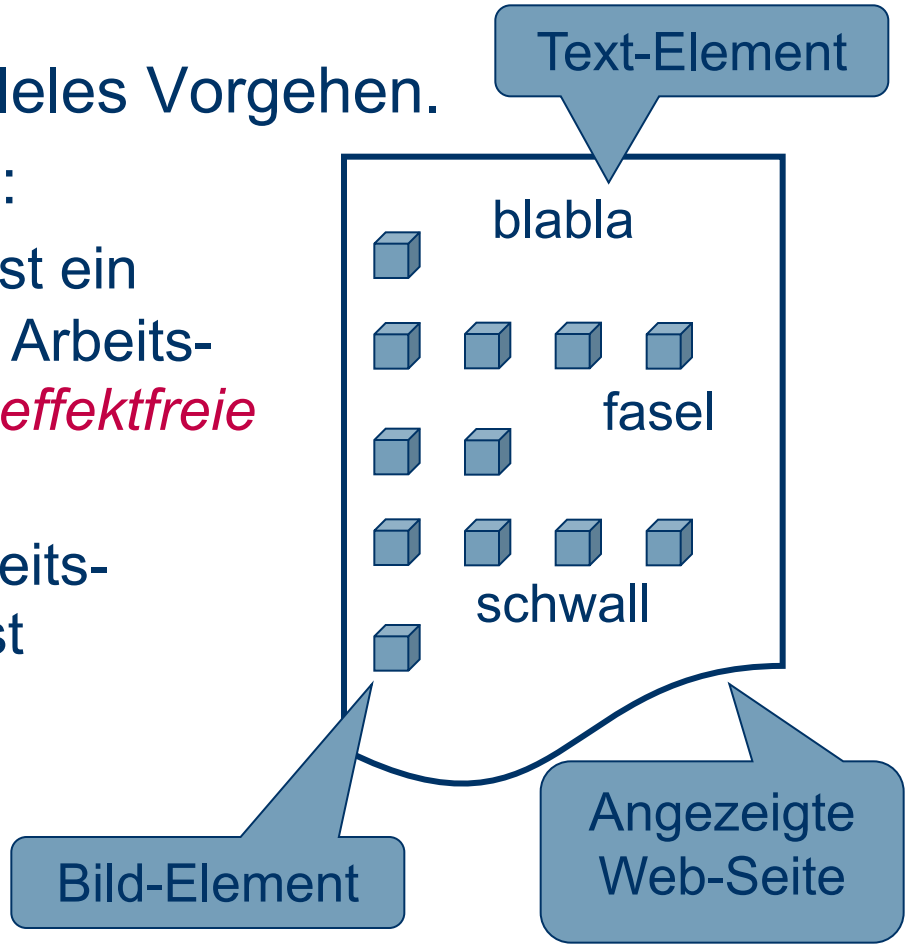
    →Chef & Arbeiter, „master & worker“
      - Arbeitsdiebstahl, „work stealing“
  - Einfache Abhängigkeit (gemeinsame Daten, Reihenfolge)
    - Fließband, Produzent & Konsument, „producer & consumer“
- Rekursive Organisationsform
  - Paralleles teile-und-herrsche

Chef & Arbeiter, „master & worker“

- Zwei Beispiele:
  - →*Anzeige einer Web-Seite im Browser*
    - Warteschlangen zur Begrenzung des Parallelitätsgrads
    - Lastbalance
  - Monte-Carlo-Simulation zur Berechnung von π
    - Leistungsbewertung paralleler Anwendung
- Weitere Beispiele:
  - Strahlenverfolgung („ray tracing“) im Abschnitt „Arbeitsdiebstahl“

Chef & Arbeiter am Beispiel (1)

- Beispiel 1: Anzeige einer Web-Seite durch den Browser. Die Darstellung von Text und allen Bildern wird parallel ausgerechnet.
- Klassische Frage für task-paralleles Vorgehen.
- Im Task-Abhängigkeitsgraphen:
  - Jedes anzuzeigende Element ist ein Arbeitspaket, das von anderen Arbeits-paketen unabhängig ist *(seiteneffektfreie Funktionsauswertung).*
  - Die Reihenfolge, in der die Arbeits-pakete fertig gestellt werden, ist beliebig.
  - Lediglich eine Synchronisation nach Bearbeitung aller Pakete.



„forall“-Muster

- Sequentielle **`for`**-Schleife

```
for (Element i : liste) {
   process(i);
}
```

- Parallele Ausführung „forall“:

```
for (final Element i : liste) {
   exec.execute(new Runnable() {
      public void run() {
         process(i);
      }
   });
}
```

Möglich & sinnvoll wenn:

- einzelne Iterationen voneinander unabhängig,
- Kosten der Iterationen hoch genug, um den Aufwand der „task“-Bearbeitung zu rechtfertigen.

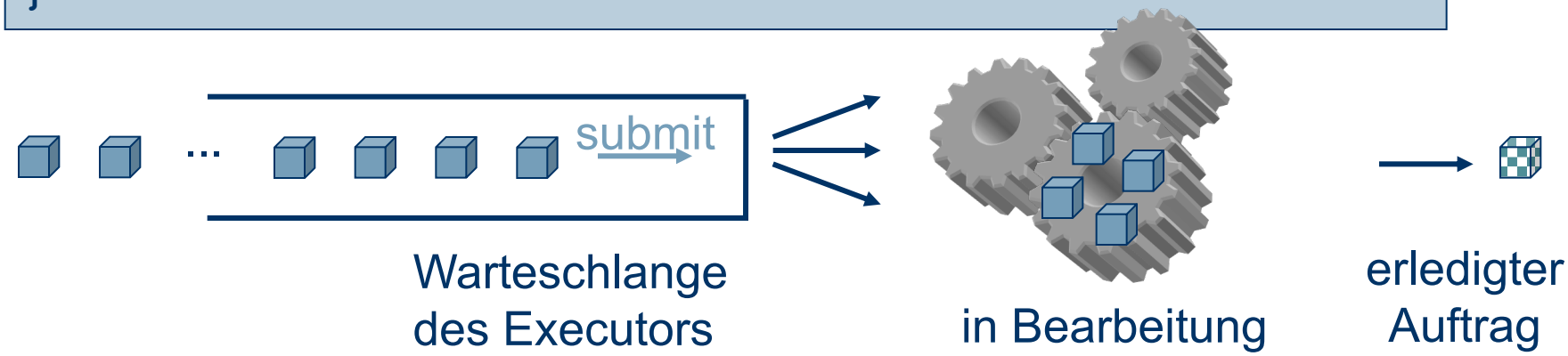
`CompletionService` = `Executor` plus `BlockingQueue`

- Mit **`submit`** werden **`Callable`**-Objekte zur Ausführung bereitgestellt.
- Die Ergebnisse (= **`Future`**-Objekte) werden in einer **`BlockingQueue`** bereitgestellt.
  - Mit **`poll`** kann man nachschauen, ob bereits ein Ergebnis verfügbar ist.
  - **`take`** liefert ein **`Future`**-Objekt oder blockiert, bis eines verfügbar ist.

Beispiel Seitendarstellung im Web-Browser (1)

- Alle in der Web-Seite enthaltenen Grafik-Adressen können nebenläufig geladen werden.
- Anwendung des „forall“-Musters:

```
final List<ImageInfo> liste = ... // gif urls in page
for (final ImageInfo i : liste) {
   completionService.submit(new Callable<ImageData>() {
      public ImageData call() {
         return i.downloadImage();
      }
   });
}
```



Beispiel Seitendarstellung im Web-Browser (2)

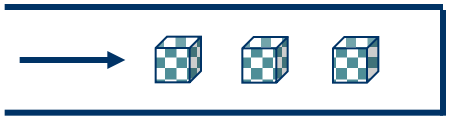
- Sobald die erste Grafik vollständig geladen ist, wird mit der Anzeige begonnen:

```
Executor exec = new ...;
CompletionService<ImageData> completionService =
   new ExecutorCompletionService<ImageData>(exec);

// Nebenläufiger Download der Graphiken (s.o.)

try {
   for (int t = 0; t < info.size(); t++) {
      Future<ImageData> f = completionService.take();
      ImageData i = f.get();
      render(i);
   }
} catch (InterruptedExcetion ie) {...}
```

Ergebnis sofort verfügbar.

Blockiert solange, bis Ergebnis in der Schlange verfügbar ist.

Warteschlange des CompletionService



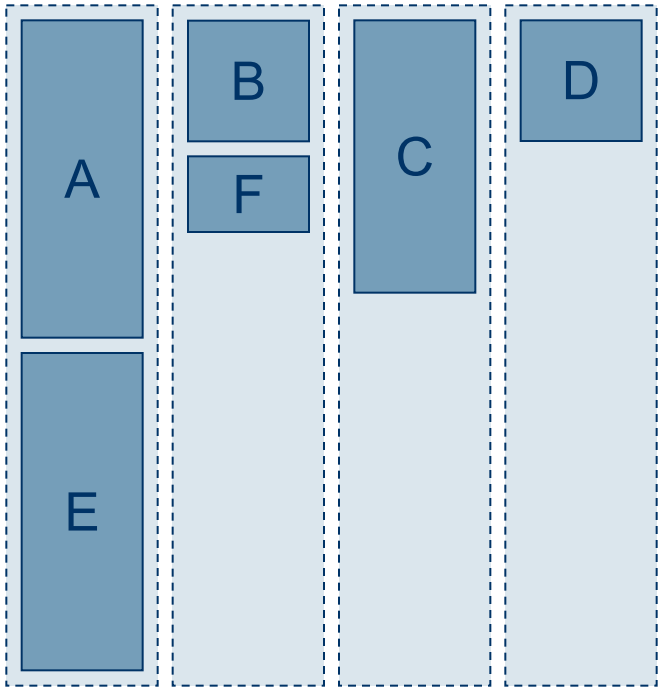
take

Lastverteilung/Lastbalance

- Mehrere unabhängige Arbeitspakete werden auf die verfügbaren Rechenkerne aufgeteilt → Ablaufplanung.

Schlechte Lastbalance

auf 4 Rechnerkernen



Gute Lastbalance

auf 4 Rechenkernen

Faustregeln zur guten Lastbalance

- Automatische Ablaufplaner erreichen im Allgemeinen gute Ergebnisse, wenn
  - es ausreichend viele Arbeitspakete gibt und
  - die Arbeitspakete einigermaßen gleich groß sind.
- Wenn die Größen der Arbeitspakete erheblich variieren oder nicht vorhersagbar sind, sollte man die Ablaufplanung manuell erledigen, damit man gute Laufzeiten erreicht.

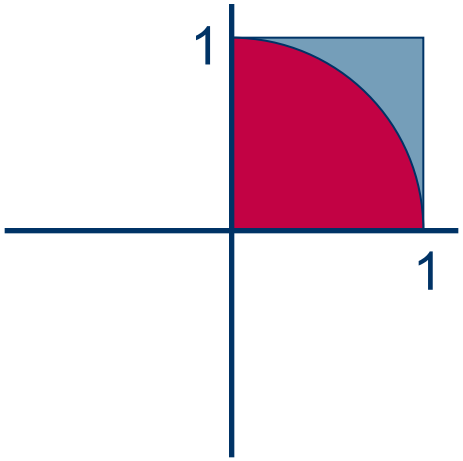
Chef & Arbeiter, „master & worker“

- Zwei Beispiele:
  - Anzeige einer Web-Seite im Browser
    - Warteschlangen zur Begrenzung des Parallelitätsgrads
    - Lastbalance

  →Monte-Carlo-Simulation zur Berechnung von π
    - Leistungsbewertung paralleler Anwendung
- Weitere Beispiele:
  - Strahlenverfolgung („ray tracing“) im Abschnitt „Arbeitsdiebstahl“

Chef & Arbeiter am Beispiel (2)

- Beispiel 2: Monte-Carlo-Simulation zur π-Berechnung
- Idee zur Berechnung von π:
  - Bestimme experimentell den Flächeninhalt eines Kreises.
  - Für den Einheitskreis ist die Fläche bekannt:
    - Formel für die Flächenberechnung: $A = r^2 \pi$
    - Beim Einheitskreis ist der Radius 1; daher ist $A = \pi$
    - Schätzt man die Fläche eines Viertels des Kreises, so kann man die Gesamtfläche (und damit π) hochrechnen.



- Algorithmus:
  - Platziere zufällig viele Punkte im Quadrat (0,0) bis (1,1).
  - Zähle Punkte im Einheitsviertel.
  - Verhältnis der Punkte im Kreis zur Gesamtanzahl der Punkte ergibt π/4.

Monte-Carlo-Simulation zur Berechnung von π – sequentiell (1)

```
import java.util.Random;
public class MonteCarloSerial {
    public static void main(String[] args) {
        int pts = Integer.MAX_VALUE / 32;
        int cnt = 0;
        long start = System.currentTimeMillis();
        Random rnd = new Random();
        for (int i = 0; i < pts; i++) {
            double x = rnd.nextDouble();
            double y = rnd.nextDouble();
            double d = x*x + y*y;
            if (d <= 1.0) cnt++;
        }
        long end = System.currentTimeMillis();
        System.out.println("pi:  " +
                           (((double) cnt)/pts) * 4);
        System.out.println("took " + (end - start) + " ms");
}   }
```

Platziere zufällig Punkte im Quadrat (0,0) bis (1,1) und zähle.

Monte-Carlo-Simulation zur Berechnung von π – sequentiell (2)

```
import java.util.Random;
public class MonteCarloSerial {
    public static void main(String[] args) {
        int pts = Integer.MAX_VALUE / 32;
        int cnt = 0;
        long start = System.currentTimeMillis();
        Random rnd = new Random();
        for (int i = 0; i < pts; i++) {
            double x = rnd.nextDouble();
            double y = rnd.nextDouble();
            double d = x*x + y*y;
            if (d <= 1.0) cnt++;
        }
        long end = System.currentTimeMillis();
        System.out.println("pi:  " +
                           (((double) cnt)/pts) * 4);
        System.out.println("took " + (end - start) + " ms");
}   }
```

Berechne Verhältnis von Punktanzahl im Viertelkreis und außerhalb.

Monte-Carlo-Simulation zur Berechnung von π – sequentiell (3)

```
import java.util.Random;
public class MonteCarloSerial {
    public static void main(String[] args) {
        int pts = Integer.MAX_VALUE / 32;
        int cnt = 0;
        long start = System.currentTimeMillis();
        Random rnd = new Random();
        for (int i = 0; i < pts; i++) {
            double x = rnd.nextDouble();
            double y = rnd.nextDouble();
            double d = x*x + y*y;
            if (d <= 1.0) cnt++;
        }
        long end = System.currentTimeMillis();
        System.out.println("pi:  " +
                           (((double) cnt)/pts) * 4);
        System.out.println("took " + (end - start) + " ms");
}   }
```

Zeitmessung

Zeitmessung

Monte-Carlo-Simulation zur Berechnung von π — parallel (1)

- Eine Analyse der Hauptschleife zeigt, dass man sie parallel berechnen kann:
  - Jeder Arbeiter erzeugt für sich Punkte und zählt diese.
  - Am Ende werden alle Zwischensummen von allen Arbeitern aufsummiert.
- Die Berechnung von lokalen Zwischenergebnissen mit anschließendem Zusammenführen zum globalen Gesamtergebnis heißt auch *Reduktion* (später mehr dazu).
  - Informell: Die Zwischenergebnisse werden zum Gesamtergebnis reduziert.
  - Achtung: Operation muss *assoziativ und kommutativ* sein, sonst kann sich das parallel errechnete Ergebnis vom sequentiell errechneten unterscheiden.

Monte-Carlo-Simulation zur Berechnung von π – parallel (2)

```
import java.util.Random;
public class MonteCarloSerial {
    public static void main(String[] args) {
        int pts = Integer.MAX_VALUE/32;
        int cnt = 0;
        long start = System.currentTimeMillis();
        Random rnd = new Random(System.currentTimeMillis());
        for (int i = 0; i < pts; i++) {
            double x = rnd.nextDouble();
            double y = rnd.nextDouble();
            double d = x*x + y*y;
            if (d <= 1.0) cnt++;
        }
        long end = System.currentTimeMillis();
        System.out.println("pi:  " +
                           (((double) cnt)/pts) * 4);
        System.out.println("took " + (end - start) + " ms");
}   }
```

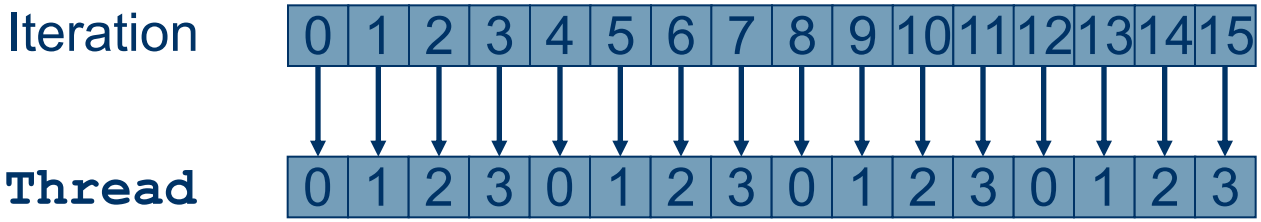
Dieser Teil des Programms kann auf **Threads** verteilt werden.

Monte-Carlo-Simulation zur Berechnung von π – parallel (3)

- Vorgehen:
  - Lagere die Schleife in eine Klasse aus, die **`Runnable`** implementiert.
  - Jeder **`Thread`** arbeitet nur einen Teil der Schleife ab.
  - Wichtig:
    - Jede Schleifeniteration darf nur einmal berechnet werden.
    - Alle Schleifeniterationen müssen berechnet werden.

Monte-Carlo-Simulation zur Berechnung von π – parallel (4)

- Einfachste Aufteilung, die kein Lastungleichgewicht erzeugt:
  - Weise jedem **`Thread`** eine ID zu.
  - Jeder **`Thread`** startet die Schleife bei der Iteration, die seiner ID entspricht.
  - Ändere Schrittweite der Ursprungsschleife auf Anzahl der **`Threads`**.
- Dieses Vorgehen ergibt folgende Zuteilung:



Monte-Carlo-Simulation zur Berechnung von π — `Threads` (1)

```
class MonteCarloWorker implements Runnable {
    private int id;
    private int numThreads;
    private int pts;
    private AtomicInteger globalSum;
    private Random rnd;

    public MonteCarloWorker(int id, int numThreads,
            int pts, AtomicInteger globalSum) {
        this.id = id;
        this.numThreads = numThreads;
        this.pts = pts;
        this.globalSum = globalSum;
        this.rnd = new Random();
    }

    // weiter: nächste Folie
```

Initialisierung der Parameter für die Berechnung.

Monte-Carlo-Simulation zur Berechnung von π — `Threads` (2)

```
    // Fortsetzung von MonteCarloWorker

    public void run() {
        int mySum = 0;
        for (int i = id; i < pts; i += numThreads) {
            double x = rnd.nextDouble();
            double y = rnd.nextDouble();
            double d = x*x + y*y;
            if (d <= 1.0)
                mySum++;
        }
        globalSum.addAndGet(mySum);
    }

}
```

Zähle zunächst nur selbst erzeugte Punkte.

Addiere eigene Zählung zum Gesamtergebnis.

Monte-Carlo-Simulation zur Berechnung von π – Programm (1)

```
import java.util.Random;
import java.util.concurrent.atomic.AtomicInteger;

public class MonteCarloParallel {
    public static void main(String[] args) throws Exception {
        int numThr = Integer.parseInt(args[0]);
        int pts = Integer.MAX_VALUE / 32;
        Thread[] threads = new Thread[numThr];
        AtomicInteger sum = new AtomicInteger(0);
        long start = System.currentTimeMillis();
        for (int i = 0; i < numThr; i++) {
            MonteCarloWorker worker =
                new MonteCarloWorker(i, numThr, pts, sum);
            threads[i] = new Thread(worker);
            threads[i].start();
        }
        // weiter auf der nächsten Folie
```

Erzeugen der **Threads**.

Monte-Carlo-Simulation zur Berechnung von π – Programm (2)

```
        // Fortsetzung vom Hauptprogramm

        for (int i = 0; i < numThr; i++) {
            threads[i].join();
        }

        long end = System.currentTimeMillis();

        System.out.println("pi:   " +
                           (((double) sum.get())/pts) * 4);
        System.out.println("computation took " +
                           (end - start) + " ms");
    }
}
```

Warte bis alle **Threads** fertig sind.

Leistungsmaße

- Die zwei wesentlichen Gründe für den Einsatz paralleler Programmiersprachen:
  - Applikationen laufen schneller, wenn man Teile parallel macht.
  - Größere Probleme können in gleicher Zeit bearbeitet werden.
- Für die Bewertung der Leistung sequentiellen Codes haben wir den O-Kalkül kennen gelernt.
- Hier: Wie bewertet man parallelen Code?

„Speedup“

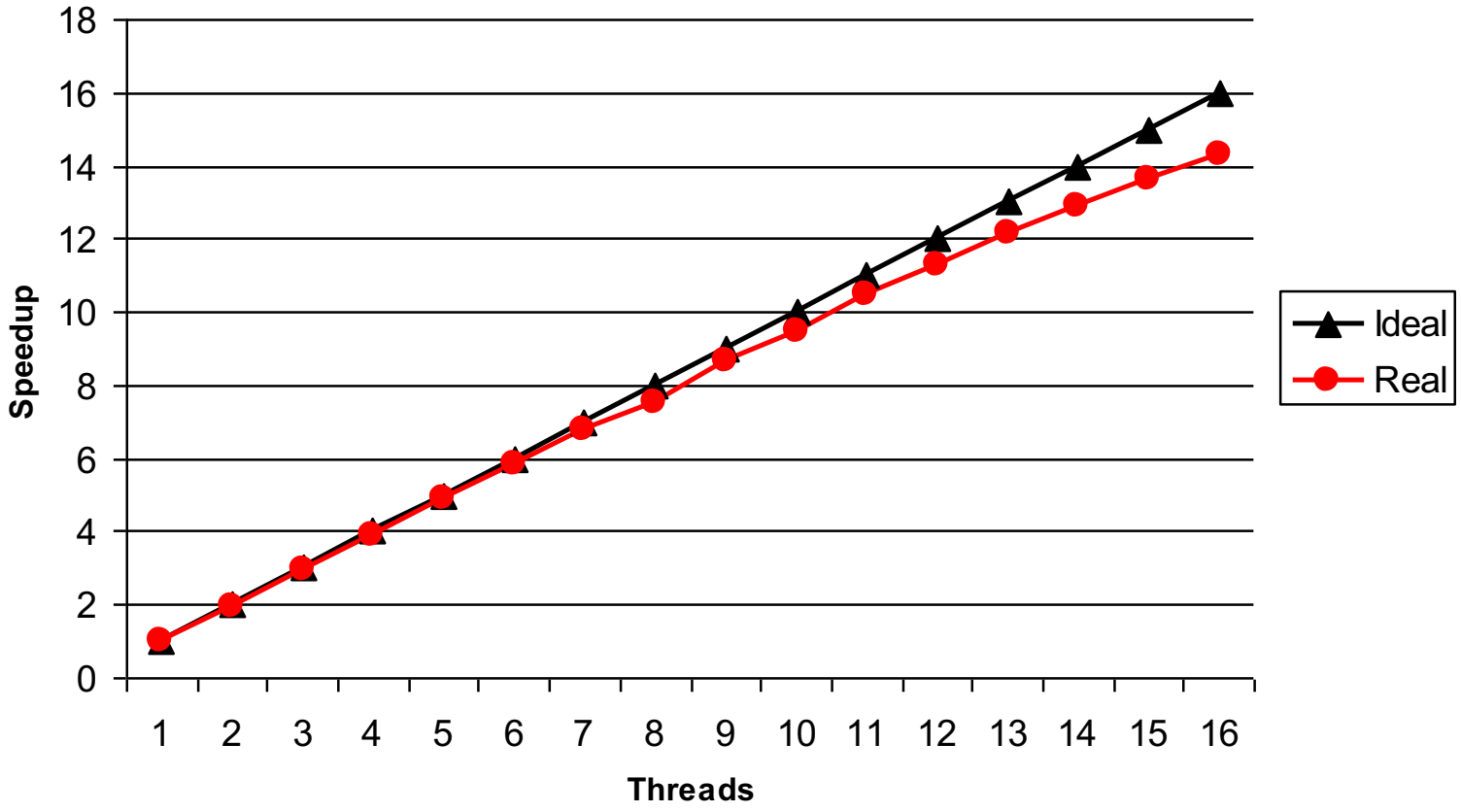
- Wenn man das Programm auf doppelt so vielen Prozessoren ausführt, dann erwartet/hofft man, dass
  - das Programm nur noch die halbe Zeit braucht, also
  - dabei doppelt so schnell rechnet.
- Für diesen Zusammenhang definiert man den *Speedup S(n)*:

$$S_p(n) = \frac{T^*(n)}{T_p(n)}$$

  - $S_p(n)$ drückt eine relative Geschwindigkeitssteigerung aus.
  - $T^*(n)$ ist die Laufzeit des schnellsten sequentiellen Algorithmus für das Problem bei Eingabegröße n.
  - $T_p(n)$ ist die Laufzeit des parallelen Programms mit p Prozessoren (bei Eingabegröße n).
- *Idealer Speedup* bedeutet, dass der Speedup eines Programms mit der Prozessorzahl übereinstimmt.

Ausführung der Monte-Carlo-Simulation

- *Idealer Speedup* bedeutet, dass der Speedup eines Programms mit der Prozessorzahl übereinstimmt.
- Ausführung der Monte-Carlo-Simulation auf realem Rechner:



Gesetz von Amdahl (1)

- Warum fällt der Speedup bei steigender **`Thread`**-Anzahl langsam ab?
- Ein Programm lässt sich wie folgt zerlegen:
  - in sequentielle Anteile *s(n)*, die *nicht parallelisierbar* sind,
  - in parallele Anteile *p(n)*, die auf mehrere **`Threads`** verteilt werden können.
  - Die Anteile *s(n)* und *p(n)* lassen sich in Prozent ausdrücken, d.h. $s(n), p(n) \in [0,1]$ und $s(n) + p(n) = 1$.
- Da man nur die parallelisierbaren Programmanteile parallel ausführen kann, ergibt sich für den Speedup $S_P(n)$:

$$S_P(n) = \frac{T^*(n)}{s(n) \cdot T^*(n) + \frac{p(n)}{P} \cdot T^*(n)}$$

Anteilige sequentielle Ausführungszeit.

Anteilige parallele Ausführungszeit auf *P* Prozessoren.

Gesetz von Amdahl (2)

- Für steigende Prozessorzahlen lässt sich mit der Formel der vorherigen Folie der Speedup nach oben abschätzen:

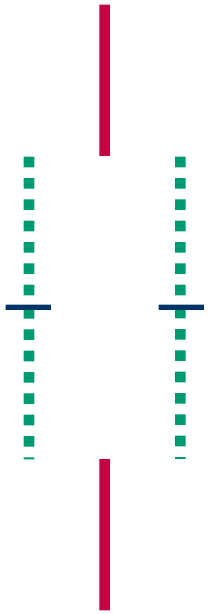
$$S_P(n) = \frac{T^*(n)}{s(n) \cdot T^*(n) + \frac{p(n)}{P} \cdot T^*(n)} \leq \frac{1}{s(n)}$$

- Die Abschätzung sagt, dass der maximale Speedup eines Programms durch den sequentiellen Anteil beschränkt ist.
  - Nicht nur ein theoretisches Limit, sondern ein praktisches (wie die Monte-Carlo-Simulation zeigt).
  - Besonders problematisch, da heutige Höchstleistungsrechner (und kommende Multi-Core-Rechner) hunderte oder gar tausende Prozessoren besitzen.

Gesetz von Amdahl (3)

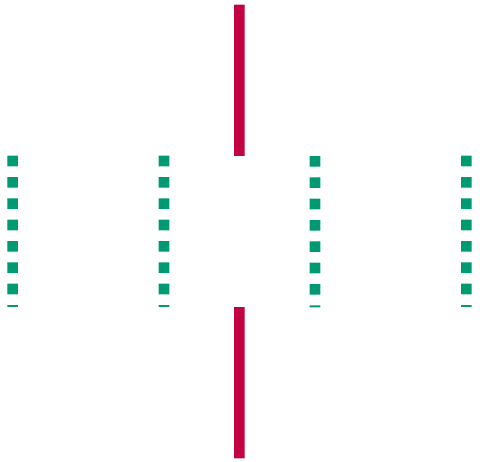
Sequentiell

2 **Threads**



4 Zeiteinheiten
Speedup von 1,5

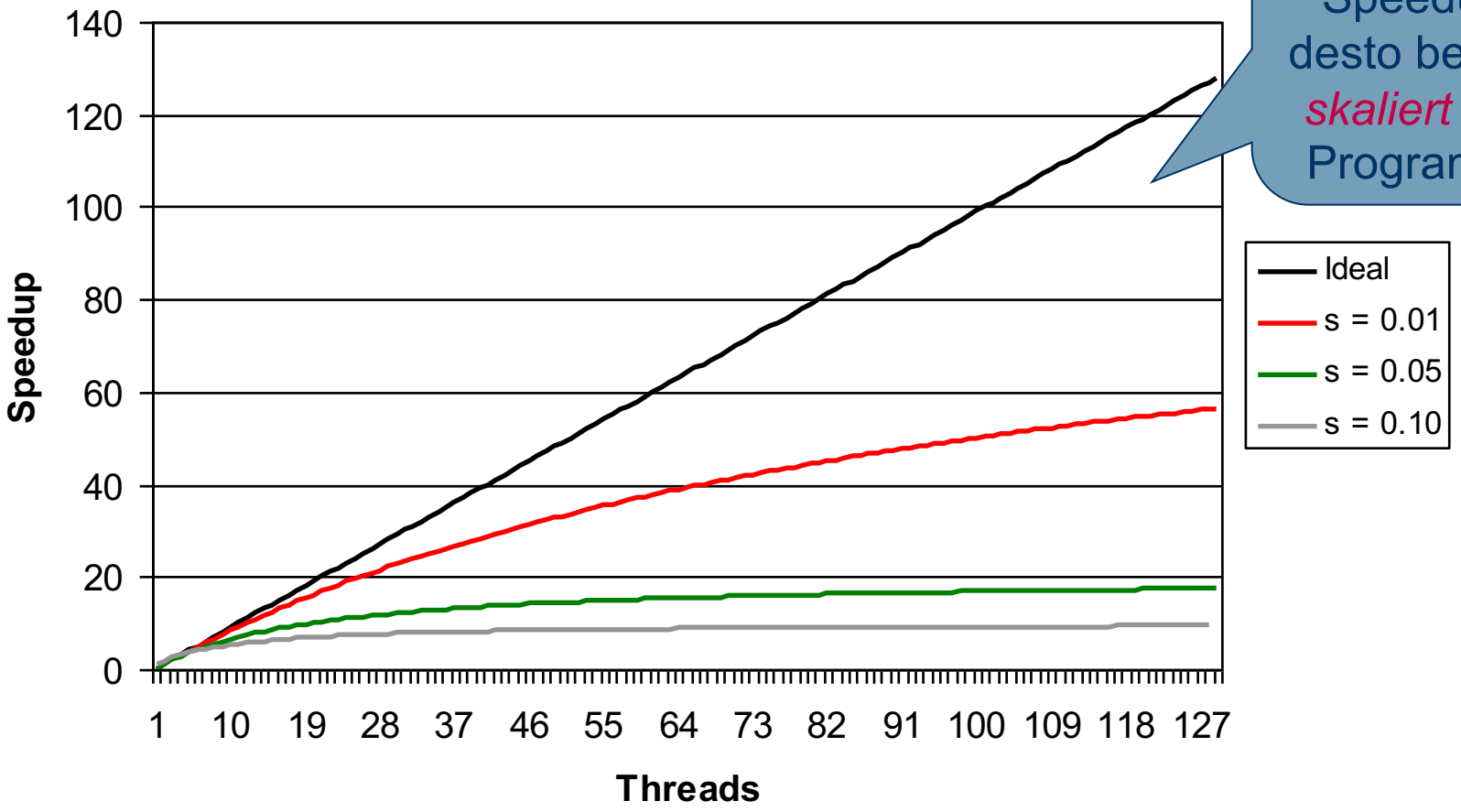
4 **Threads**



3 Zeiteinheiten
Speedup von 2

Gesetz von Amdahl (4)

- Auswirkung des Gesetzes von Amdahl für unterschiedliche Werte von *s*:



Je näher am idealen Speedup, desto besser *skaliert* das Programm.

Sequentielle Anteile in der Monte-Carlo-Simulation (1)

```
import java.util.Random;
import java.util.concurrent.atomic.AtomicInteger;

public class MonteCarloParallel {
    public static void main(String[] args) throws Exception {
        int numThreads = Integer.parseInt(args[0]);
        int pts = Integer.MAX_VALUE/32;
        Thread[] threads = new Thread[numThreads];
        AtomicInteger sum = new AtomicInteger(0);
        long start = System.currentTimeMillis();
        for (int i = 0; i < numThreads; i++) {
            MonteCarloWorker worker =
                new MonteCarloWorker(i, numThreads, pts, sum);
            threads[i] = new Thread(worker);
            threads[i].start();
        }
        // weiter auf der nächsten Folie
```

Sequentielle Anteile in der Monte-Carlo-Simulation (2)

```
        // Forsetzung vom Hauptprogramm

        for (int i = 0; i < numThreads; i++) {
            threads[i].join();
        }

        long end = System.currentTimeMillis();

        System.out.println("pi:   " +
                           (((double) sum.get())/pts) * 4);
        System.out.println("computation took " +
                           (end - start) + " ms");
    }
}
```

Parallele Anteile in der Monte-Carlo-Simulation

```
    // Fortsetzung von MonteCarloWorker

    public void run() {
        int mySum = 0;
        for (int i = id; i < pts; i += numThreads) {
            double x = rnd.nextDouble();
            double y = rnd.nextDouble();
            double d = x*x + y*y;
            if (d <= 1.0)
                mySum++;
        }
        globalSum.addAndGet(mySum);
    }

}
```

Parallele Effizienz

- Ein Programm A, das schlechter skaliert als ein anderes Programm B, erscheint weniger effizient parallelisiert.
  - Diese intuitive Aussage lässt sich über die Formel zur Berechnung des Speedups formalisieren.
  - Man spricht von *paralleler Effizienz.*
  - Sie ist ein Maß für die relative Zeit, die ein Programm in parallelen Programmteilen verbringt.
- Berechnung der parallelen Effizienz:
  - Die parallele Effizienz soll 1 (= 100%) ergeben, wenn das Programm einen idealen Speedup erreicht.
  - Folgende Formel leistet dies:

$$E_p(n) = \frac{S_p(n)}{p} = \frac{T^*(n)}{p \cdot T_p(n)}$$

Nenner: Von allen Prozessoren gemeinschaftlich verbrauchte Rechenzeit.