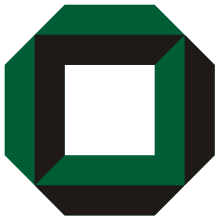


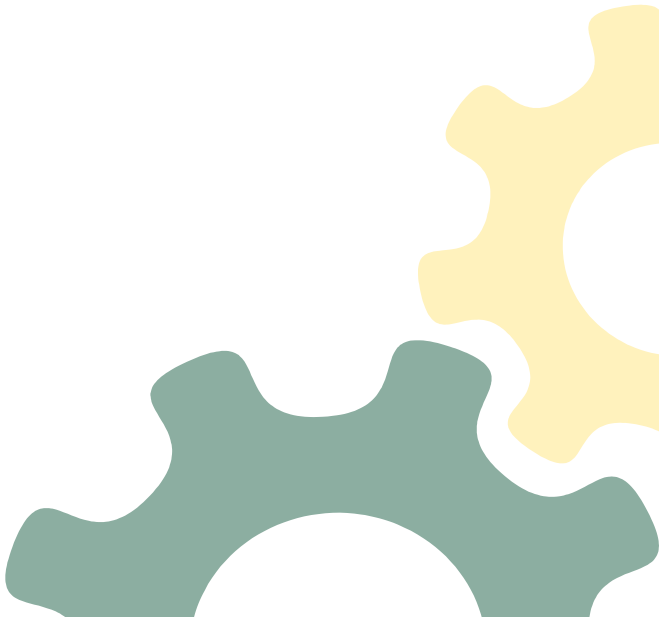
Universität Karlsruhe (TH)
Forschungsuniversität · gegründet 1825

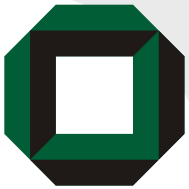
Kapitel 4.2
Parallele Algorithmen

SWT I – Sommersemester 2009

Prof. Dr. Walter F. Tichy
Dipl.-Inform. David J. Meder

Fakultät für **Informatik**
Lehrstuhl für Programmiersysteme



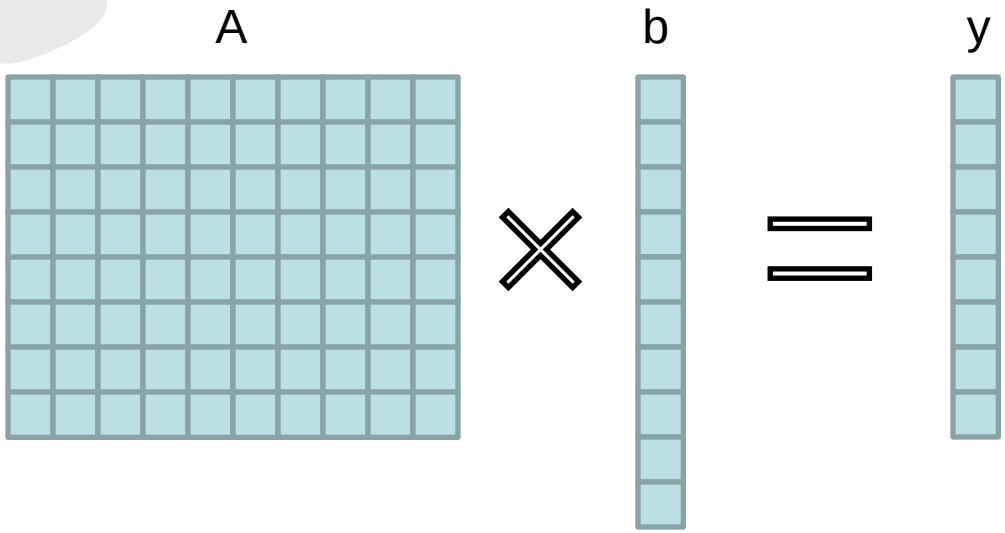


Überblick

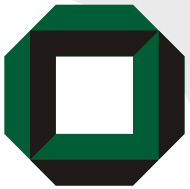
- Matrix-Vektor-Multiplikation
- Matrix-Matrix-Multiplikation
- Numerische Integration
- Bewertung von parallelen Algorithmen

Matrix-Vektor-Multiplikation: Problemstellung

Berechne das Produkt eine n×m-Matrix mit einem Vektor der Länge m.

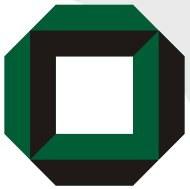


$$y_i = \sum_{j=1}^{m} a_{i,j} * b_j$$



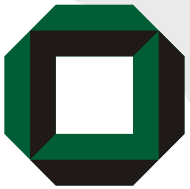
Matrix-Vektor-Multiplikation: Sequentieller Algorithmus

```
// private int[][] a = ...;
// private int[] b = ...;

public int[] multiply() {
int[] y = new int[a.length];

for (int i = 0; i < a.length; i++) {
  for (int j = 0; j < a[i].length; j++) {
      y[i] += a[i][j] * b[j];
    }
  }
return y;
}
```



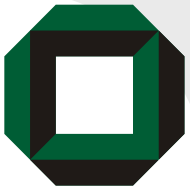
Matrix-Vektor-Multiplikation: Parallelisierung (1)

- Vorüberlegungen:
  - Können verschiedene Prozessoren gleichzeitig auf eine Speicherzelle zugreifen?
    - Vektor b unter den Prozessoren aufteilen?
  - Soll das Ergebnis in eine gemeinsame Variable geschrieben werden?
    - Schreiben u.U. mehrere Prozessoren gleichzeitig den Wert dieser Variablen?
    - Muss der Zugriff auf diese Variable koordiniert werden?
  - Wie sollen die Daten unter den Prozessoren aufgeteilt werden?
    - Matrix A zeilenweise oder spaltenweise unter den Prozessoren aufteilen?
    - Wie groß sollen die Blöcke (Anzahl Zeilen oder Spalten) sein?



Matrix-Vektor-Multiplikation: Parallelisierung (2)

- Annahmen für die folgenden Folien:
  - Variablen (in diesem Beispiel A, b und y) können von allen Prozessoren gleichzeitig gelesen werden.
  - Die beiden Vektoren und die Matrix passen vollständig in den Speicher der Prozessoren.

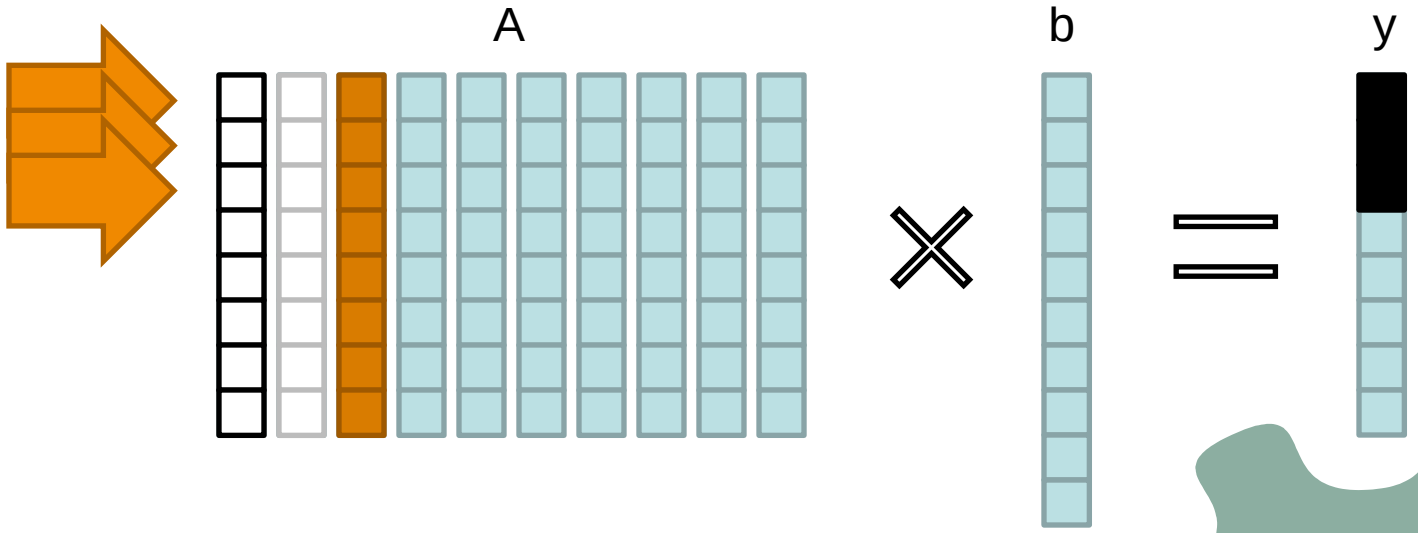


Matrix-Vektor-Multiplikation: Parallelisierung (3)

- Die Zeilenweise Aufteilung der Matrix A hat mehrere Vorteile gegenüber der spaltenweisen Aufteilung:
  - Bei spaltenweiser Aufteilung der Matrix unter den Prozessoren kommt es zu konkurrierenden Zugriffen mehrerer Prozessoren auf des gleiche Element von y.
  - Aufteilung entspricht dem bekannten Multiplikationsalgorithmus $A \times b$.
  - Zeilenweise Aufteilung ist oft performanter als spaltenweise Aufteilung (Cache-Effekt: Eine Cachezeile speichert Elemente mit aufsteigenden Adressen, und holt damit einen Teil einer Zeile auf einmal. Elemente einer Spalte sind dagegen in unterschiedlichen Cachezeilen.).

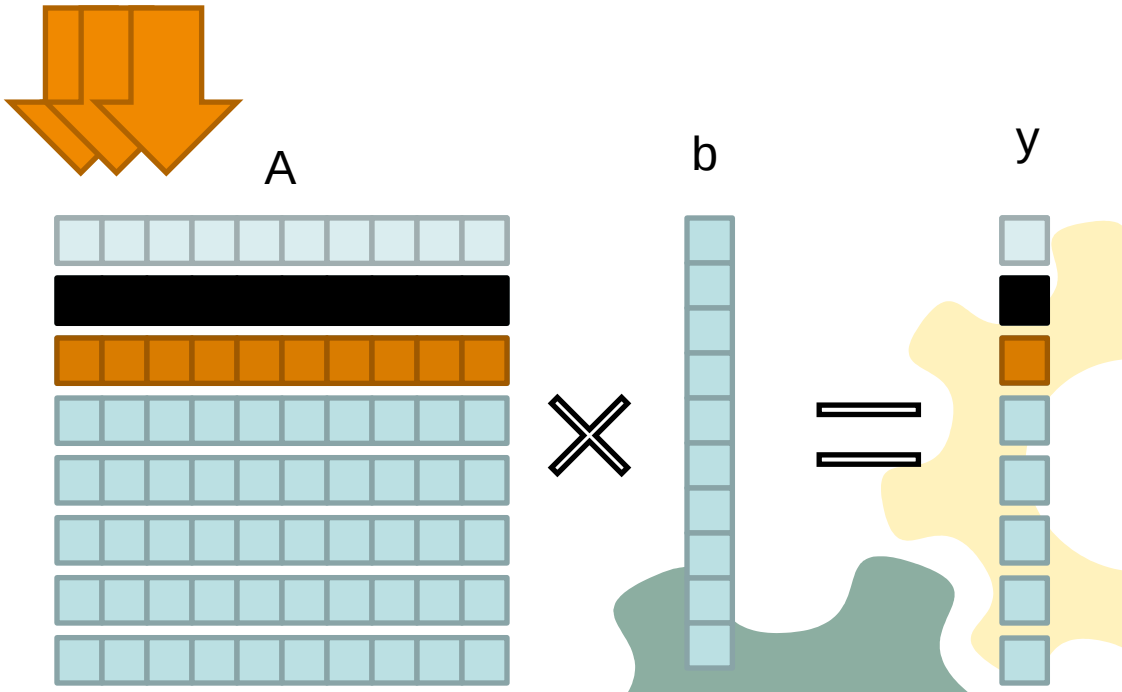
Matrix-Vektor-Multiplikation: Parallelisierung (4)

- Spaltenweise Aufteilung:
  - Mehrere Fäden wollen unter Umständen gleichzeitig das gleiche Element von y Schreiben.
    ➔Zugriff auf y muss koordiniert werden



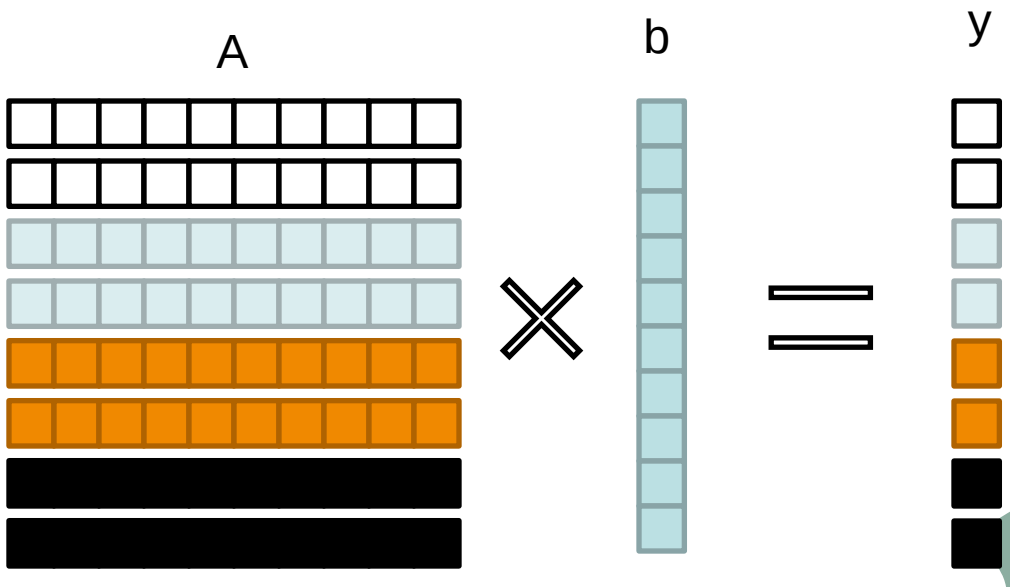
Matrix-Vektor-Multiplikation: Parallelisierung (5)

- Zeilenweise Aufteilung:
  - Jedes Element von y wird nur von einem Prozessor geschrieben.
  - Es ist keine Synchronisation erforderlich.



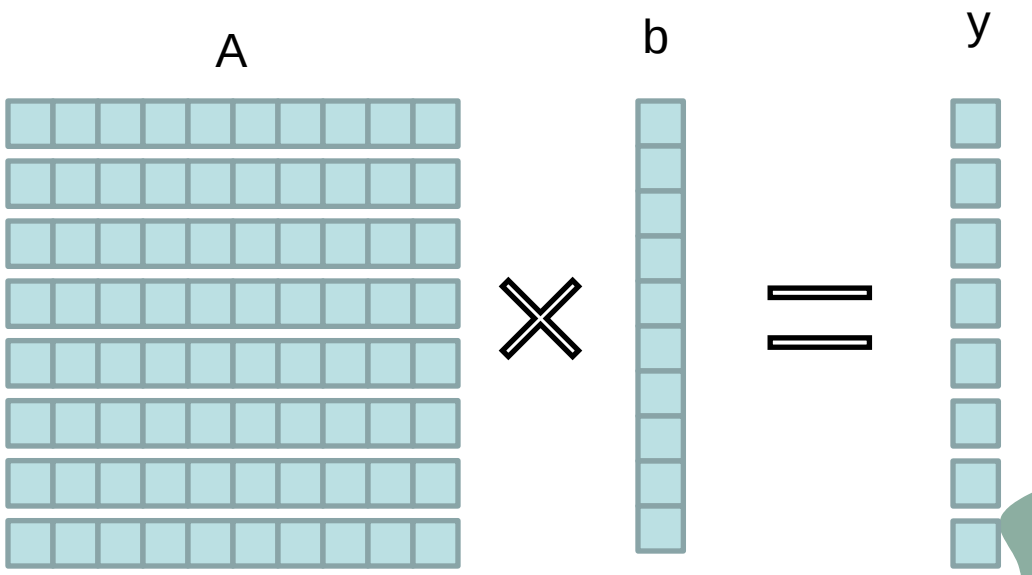
Matrix-Vektor-Multiplikation: Parallelisierung (6)

- Zeilenweise Aufteilung:
  - Statt jedem Prozessor immer nur eine Zeile von A zuzuweisen, können auch mehrere Zeilen einem Prozessor zugewiesen werden.



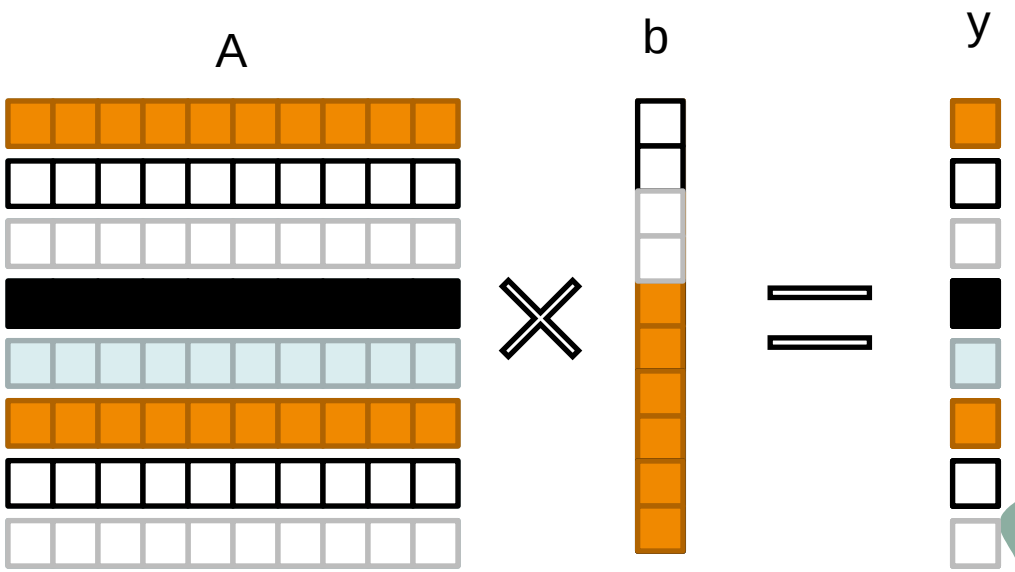
Matrix-Vektor-Multiplikation: Parallelisierung (7)

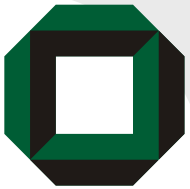
- Blockweise Aufteilung:
  - Wenn Speicherzellen nur von einem Prozessor gleichzeitig gelesen werden können, muss der Zugriff koordiniert werden.



Matrix-Vektor-Multiplikation: Parallelisierung (7)

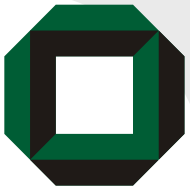
- Zugriffsaufteilung:
  - Wenn Speicherzellen nur von einem Prozessor gleichzeitig gelesen werden können, muss der Zugriff koordiniert werden.





Matrix-Vektor-Multiplikation: Parallele Implementierung (1)

```
public class MatrixVectorMultiplication implements Runnable {
  // Variablen für jeden erstellten Faden
  private int[][] myA = null;
  private int[] myB = null;
  private int[] myY = null;
  private int myStart = 0;
  private int myEnd = 0;

  private static CyclicBarrier barrier = null;

  private static void printVector(int[] vector) {
    for (int i = 0; i < vector.length; i++) {
      System.out.print(vector[i] + "\n");
    }
  }
```



Matrix-Vektor-Multiplikation: Parallele Implementierung (2)

```
private MatrixVectorMultiplication(int[][] a, int[] b, int[] y,
    int start, int end) {
  this.myA = a;
  this.myB = b;
  this.myStart = start;
  this.myEnd = end;
  this.myY = y;
}
// Führe Multiplikation myA*myB auf dem angegebenen Bereich durch
public void run() {
  for (int i = myStart; i < myEnd; i++) {
    for (int j = 0; j < myB.length; j++) {
      myY[i] += myA[i][j] * myB[j];
    }
  }

  try { barrier.await(); }
  catch (InterruptedException e)   { e.printStackTrace(); }
  catch (BrokenBarrierException e) { e.printStackTrace(); }
}
```

Warte so lange, bis alle Fäden diese Barriere erreicht haben.

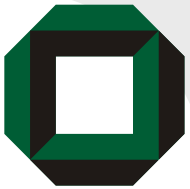
Matrix-Vektor-Multiplikation: Parallele Implementierung (3)

```
public void multiplication(int[][] a, int[] b, int threadCount) {
  MatrixVectorMultiplication[] vmmWorker = new
   MatrixVectorMultiplication[threadCount];
  Thread[] vmmThreads = new Thread[threadCount];

  barrier = new CyclicBarrier(threadCount, new Runnable() {
     public void run() {
       System.out.println("Alle Fäden haben die Barriere erreicht!");
       printVector(myY);
     }});

  myY = new int[a.length]; int start = 0; int end = 0;
  int rowCount = (int) Math.ceil((double) a.length / threadCount);

  for (int i = 0; i < threadCount; i++) {
    start = i * rowCount;
    end = Math.min((i + 1) * rowCount, a.length);

    vmmWorker[i] = new MatrixVectorMultiplication(a, b, myY, start, end);
    vmmThreads[i] = new Thread(vmmWorker[i]);
    vmmThreads[i].start();
  }
}
```

Anzahl Zeilen für jeden Faden

Zeile, ab welcher Faden i multiplizieren soll.

Letzte Zeile für Faden i.



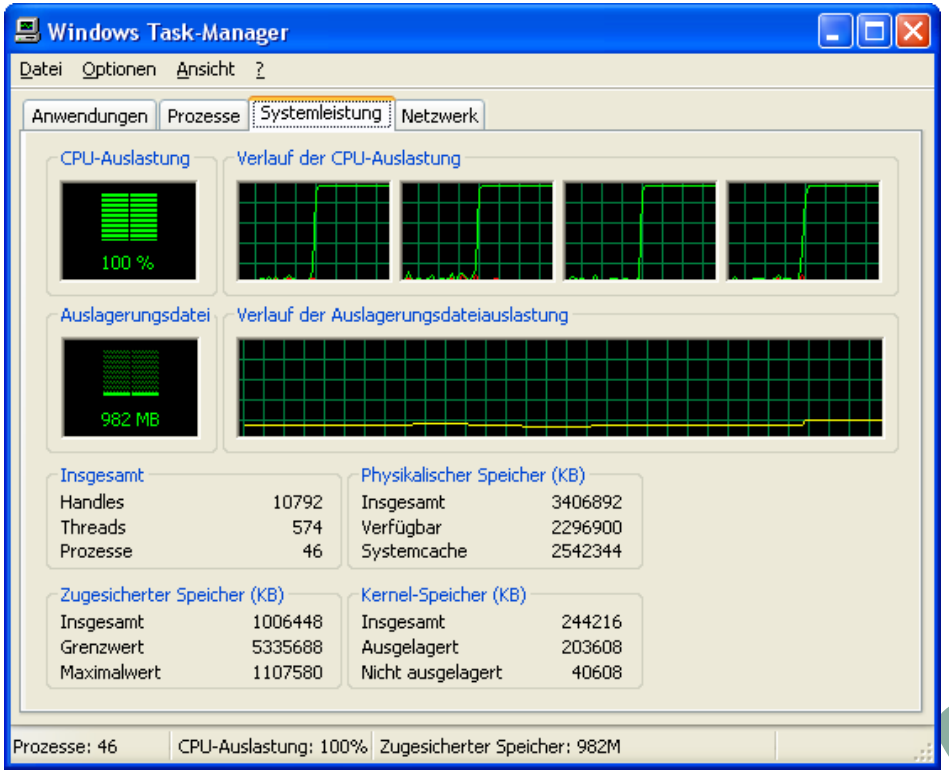
Matrix-Vektor-Multiplikation: Parallele Implementierung (4)

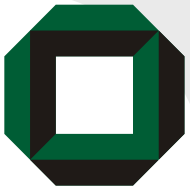
```
  public static void main(String[] args) {
    MatrixVectorMultiplication mvm = new MatrixVectorMultiplication();
    int[][] a = new int[][] {
                { 1, 1, 1, 1 },
                { 0, 2, 2, 2 },
                { 0, 0, 3, 3 },
                { 0, 0, 0, 4 } };
    int[] b = new int[] {1, 1, 1, 1};
    int threadCount = 8;

    mvm.multiplication(a, b, threadCount);
    System.out.println("Warte auf Ergebnis...");
  }
}
```

Ist es möglich, dass das Ergebnis der Multiplikation erst nach der Ausgabe von „Warte auf Ergebnis..:“ ausgegeben wird?

Matrix-Vektor-Multiplikation: Aufteilung der Rechenlast

- Verteilung von 4 Fäden auf 4 Prozessorkerne:





Matrix-Matrix-Multiplikation: Mögliche Algorithmen

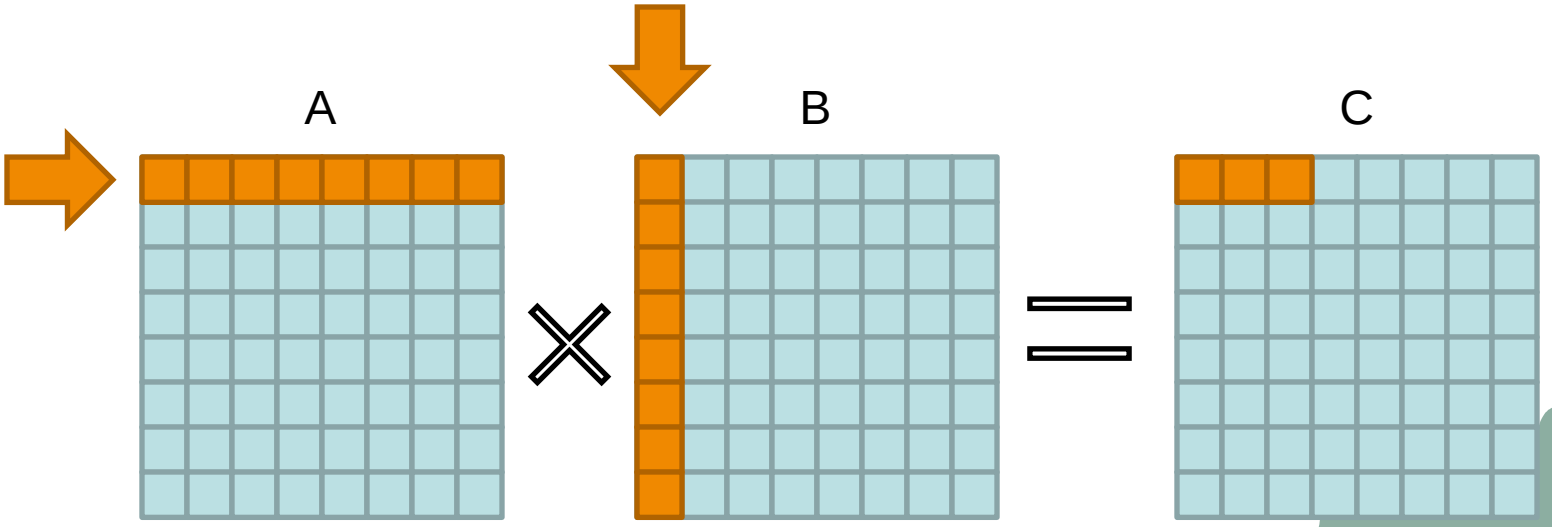
- Zur Multiplikation zweier Matrizen können verschiedene Algorithmen verwendet werden:
  - ijk-Algorithmus
    - Analog dazu: kij-Algorithmus, ikj-Algorithmus
  - Systolischer Algorithmus
  - Cannon-Algorithmus
  - …

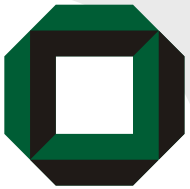
Matrix-Matrix-Multiplikation: ijk-Algorithmus

„Multipliziere jede Zeile von A mit jeder Spalte von B“

- Der ijk-Algorithmus ist der „klassische“ Algorithmus zur Multiplikation zweier Matrizen:

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} * B_{k,j}$$



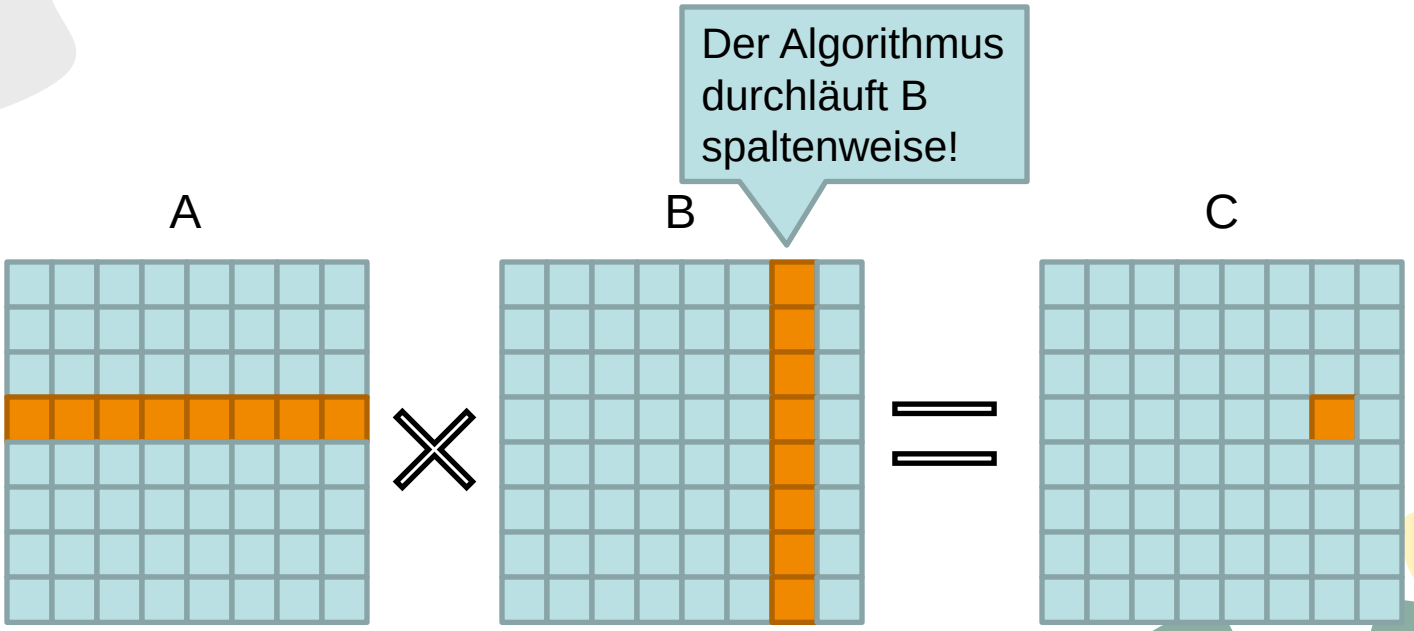


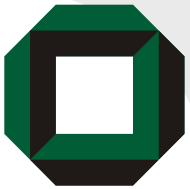
Matrix-Matrix-Multiplikation: ijk-Algorithmus

```
private final int N = ...;

public int[][] matrixMult(int[][] a, int[][] b) {
int[][] c = new int[N][N];


for (int i = 0; i < N; i++) {
  for (int j = 0; j < N; j++) {
    for (int k = 0; k < N; k++) {
        c[i][j] += a[i][k] * b[k][j];
      }
    }
  }

return c;
}
```

Matrix-Matrix-Multiplikation: Nachteil des ijk-Algorithmus

- Der ijk-Algorithmus ist, bedingt durch die Anordnung der Schleifen, nicht Cache-freundlich.





Matrix-Matrix-Multiplikation: Optimierung: ikj-Algorithmus

- Umordnung der Schleifen des ijk-Algorithmus:

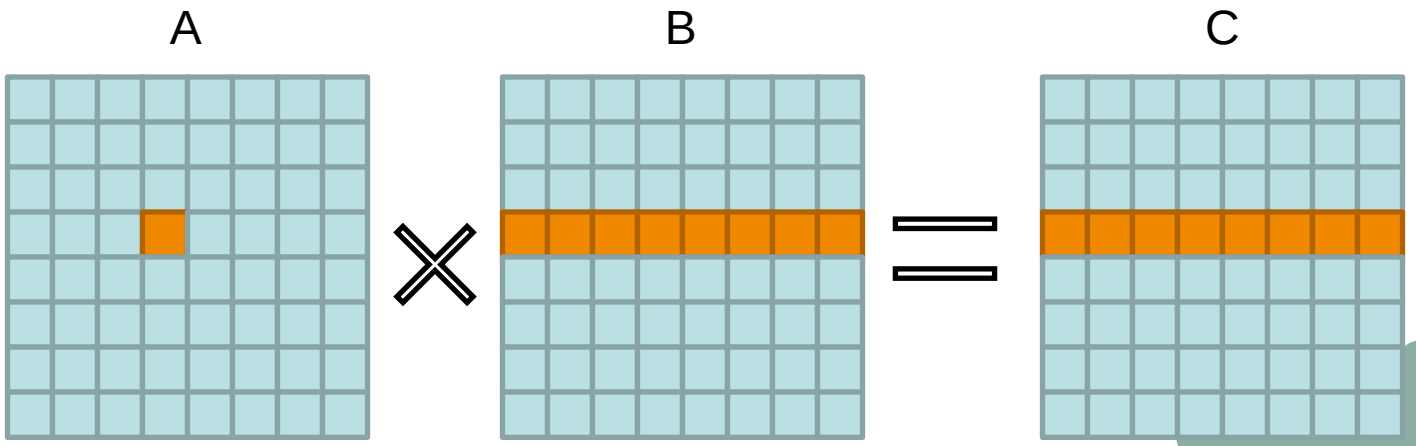
```
for (i=0; i < N; i++)
  for (k=0; k < N; k++)
    for (j=0; j < N; j++)
      c[i][j] += a[i][k] * b[k][j];
```

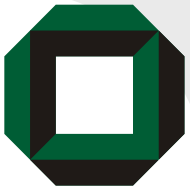
- Herausziehen des schleifeninvarianten Elements:

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

Matrix-Matrix-Multiplikation: ikj-Algorithmus (1)

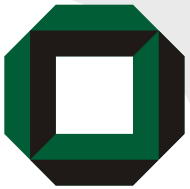
```
for (i=0; i < N; i++)
  for (k=0; k < N; k++) {
    r = a[i][k];
    for (j=0; j < N; j++)
      c[i][j] += r * b[k][j];
  }
```





Matrix-Matrix-Multiplikation: ijk- vs. ikj-Algorithmus (2)

- Der ikj-Algorithmus nutzt die Eigenschaften des Prozessor-Caches und der Abbildung der Felder (Matrix, Vektor) im Speicher aus:
  - Die Elemente der Zeile einer Matrix oder eines Vektors liegen hintereinander im Speicher.
  - Wenn das erste Element einer Zeile oder eines Vektors in den Cache geladen wird, werden je nach Größe einer Cache-Zeile gleich mehrere Elemente geladen, die dann hintereinander genutzt werden sollten. Das reduziert die Anzahl der Zugriffe auf den langsamen Hauptspeicher und somit die Ausführungszeit des Algorithmus.



Matrix-Matrix-Multiplikation: ijk- vs. ikj-Algorithmus (3)

- Aufbau des Experiments:
  - ijk- und ikj-Matrixmultiplikation in Java (mit Java Threads)
  - Verwendung von **2048x2048** Matrizen
  - Multiplikation sequentiell sowie parallel
  - Parallele Versionen mit 2..64 Fäden
  - Äußere Schleife wurde parallelisiert
- Verwendete Rechner:
  - Sun Fire T5120 mit Ultra Sparc T2 Prozessor (1,17 GHz, 8 Kerne, 8 HW-Fäden pro Kern)
  - Intel Core 2 Quad Q6600 (2,4GHz, 4 Kerne, 1 Hardware-Faden pro Kern)

Matrix-Matrix-Multiplikation: ijk- vs. ikj-Algorithmus (4)

- **Sun Fire T5120**, Matrixgröße: 2048x2048
- Laufzeit T (in Sekunden):

	sequentiell	parallel (#Fäden)				
		2	4	8	32	64
ijk-Multiplikation	665,3s	330,5s	165,3s	82,7s	19,8s	14,8s
ikj-Multiplikation	266,7s	133,3s	66,9s	33,5s	13,3s	8,2s

Faktor 2,49

- Maximale Beschleunigung (T(1)/ T(n)):

	ijk-Multiplikation	ikj-Multiplikation
Maximale Beschleunigung	45,0	32,5

Matrix-Matrix-Multiplikation: ijk- vs. ikj-Algorithmus (5)

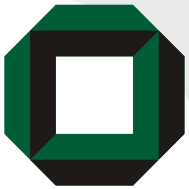
- **Intel Q6600**, Matrixgröße: 2048x2048
- Laufzeit T (in Sekunden):

	sequentiell	parallel (#Fäden)				
		2	4	8	32	64
ijk-Multiplikation	118,3s	59,3s	31,3s	31,5s	31,3s	31,3s
ikj-Multiplikation	62,7s	31,4s	16,1s	16,1s	15,9s	15,8s

Faktor 1,9

- Maximale Beschleunigung (T(1)/ T(n)):

	ijk-Multiplikation	ikj-Multiplikation
Maximale Beschleunigung	3,8	4,0



Numerische Integration

- Die **numerische Integration** bezeichnet das näherungsweise Berechnen von Integralen.
- **Beispiel:** Kann für eine Funktion f keine Stammfunktion angegeben werden, wird die numerische Integration verwendet.

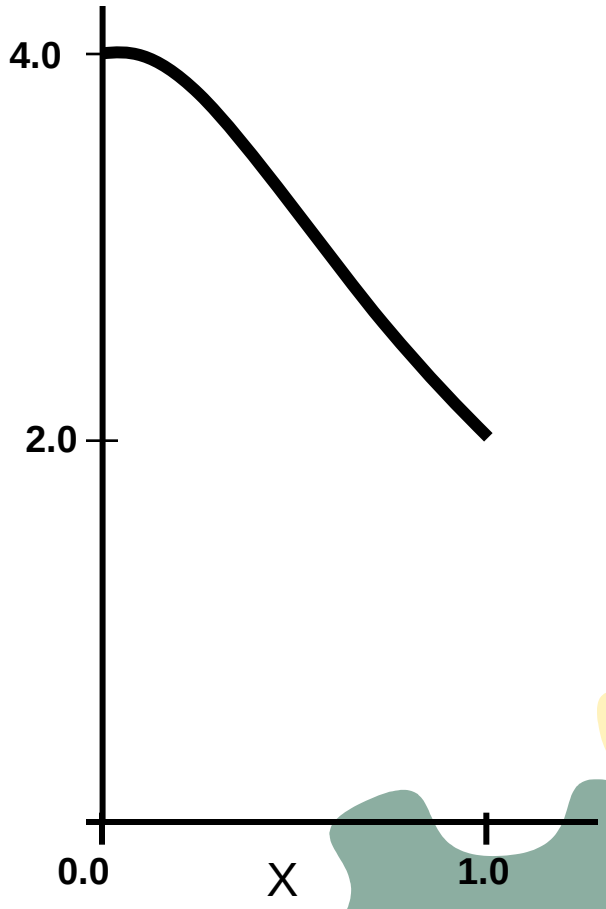
Numerische Integration: Vorgehen (1)

- Gegeben sei folgende Funktion:

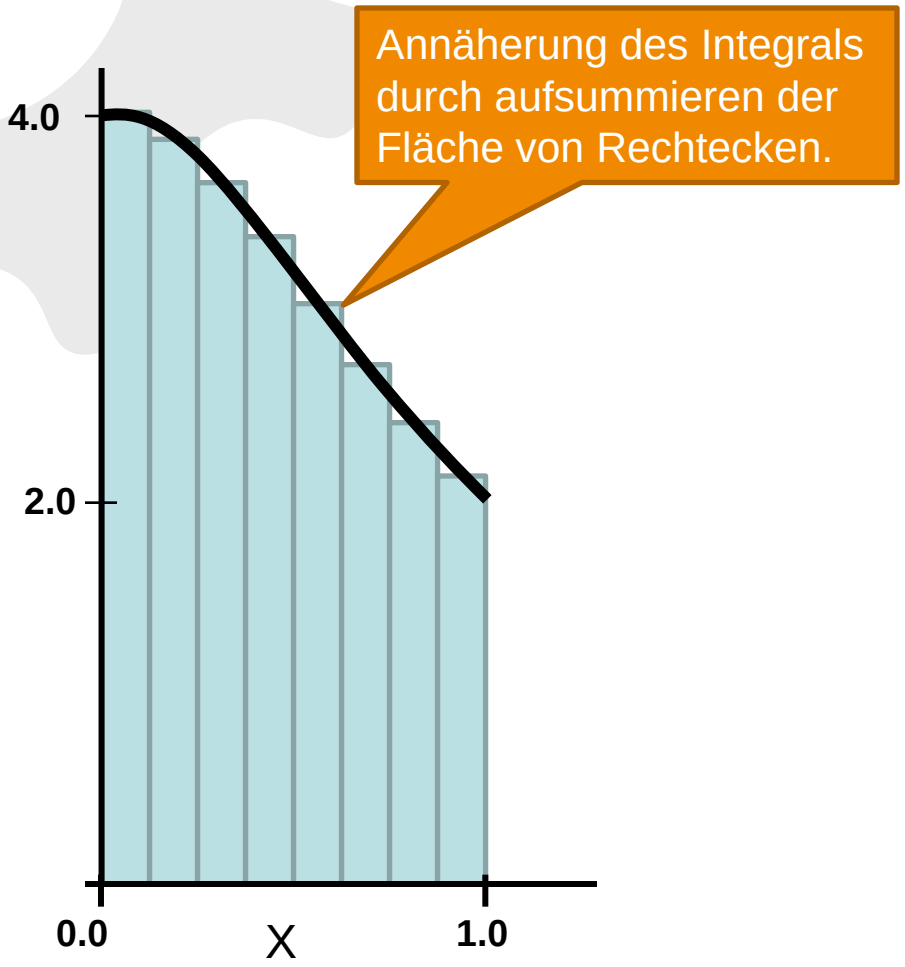
$$f(x) = \frac{4}{1+x^2}$$

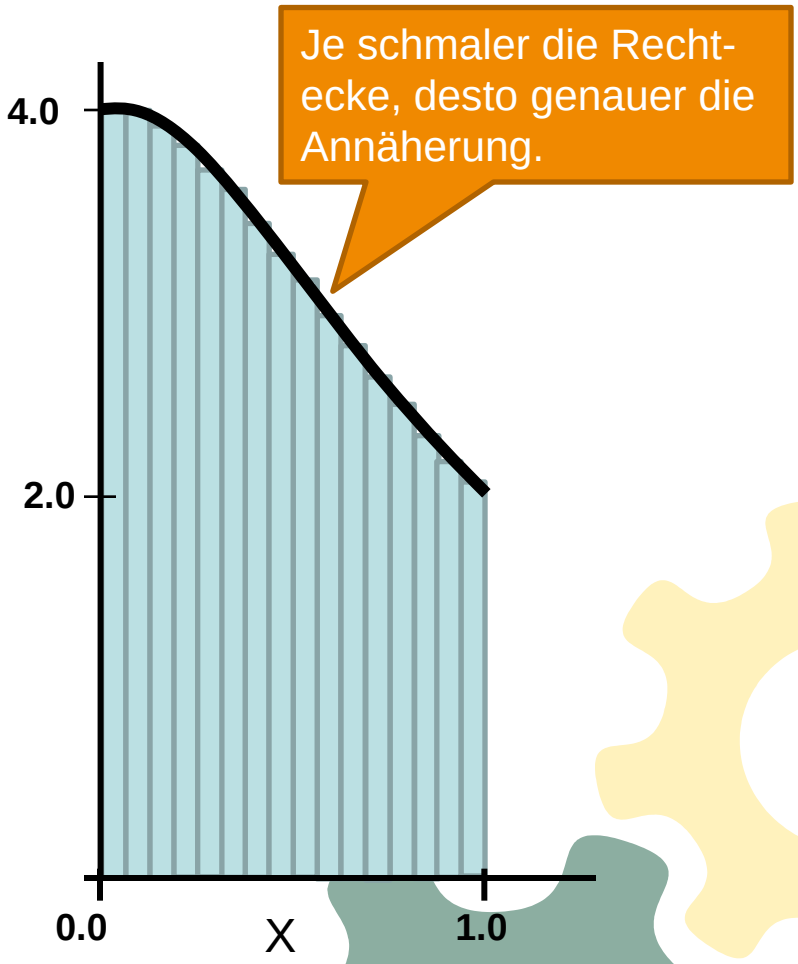
- Gesucht ist folgendes Integral:

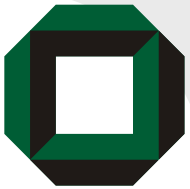
$$\int_0^1 \frac{4}{1+x^2}\,dx$$



Numerische Integration: Vorgehen (2)

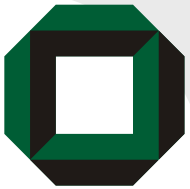






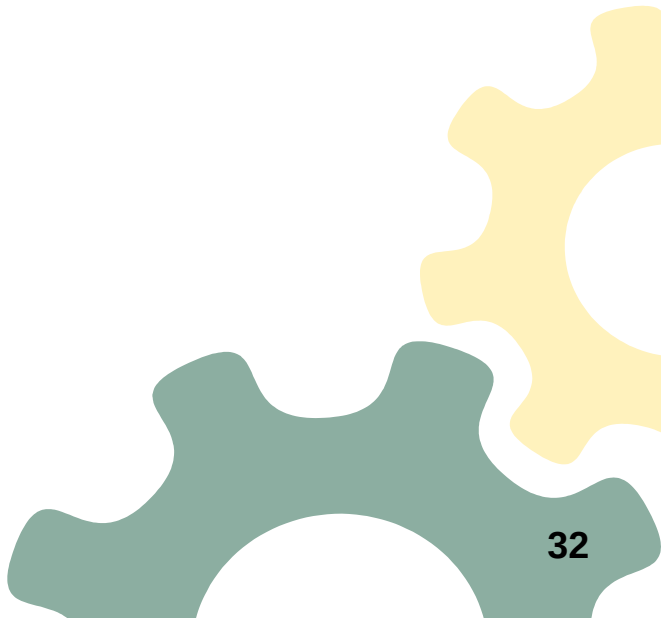
Numerische Integration: Sequentieller Algorithmus

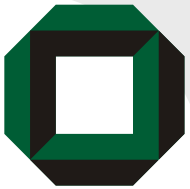
```
private final long RECT_COUNT = 1000000;
private final double RECT_SIZE = 1.0/(double)RECT_COUNT;

public double calculate() {
  double x = 0.0;
  double sum = 0.0;

  for (long i = 0; i< RECT_COUNT; i++) {
    x = (i + 0.5) * RECT_SIZE;
    sum += 4.0/(1.0 + x*x);
  }

  double result = RECT_SIZE * sum;
  return result;
}
```



Numerische Integration: Parallelisierung (1)

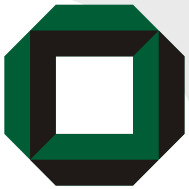
- Idee:
  - Gib jedem Prozessor einen bestimmten Bereich der Funktion, die er integrieren soll.
  - Sammle alle Teilergebnisse ein und führe sie zum Gesamtergebnis zusammen.





Numerische Integration: Beispiel (1)

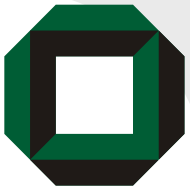
```
public class Integral implements Runnable {

  private double myResult = 0.0;
  private long myStart = 0;
  private long myEnd = 0;
  private double myStepSize = 0.0;

  private final static int DEFAULT_THREAD_COUNT = 4;
  private final static int DEFAULT_STEP_COUNT = 1000000;

  private Integral() {}

  private Integral(long start, long end, double stepSize) {
    this.myStart = start;   this.myEnd = end;
    this.myStepSize = stepSize;
  }

  public static void main(String[] args) {
    Integral inte = new Integral();
    System.out.println("Integral(f(x)) = " +
      inte.calculate(DEFAULT_STEP_COUNT, DEFAULT_THREAD_COUNT));
  }
```



Numerische Integration: Beispiel (2)

```
public double calculate(long steps, int threadCount) {
  Integral[] integralWorker = new Integral[threadCount];
  Thread[] integralThreads = new Thread[threadCount];
  long start = 0; long end = 0;

  long stepsPerThread = (int) Math.ceil((double)steps /
threadCount);
  double stepSize = 1.0 / (double) steps;

  for (int i = 0; i < threadCount; i++) {
    start = i * stepsPerThread;
    end = Math.min((i + 1) * stepsPerThread, steps);

    integralWorker[i] = new Integral(start, end, stepSize);
    integralThreads[i] = new Thread(integralWorker[i]);
    integralThreads[i].start();
  }
```

Anfang und Ende für jeden Faden berechnen.

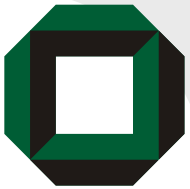


Numerische Integration: Beispiel (3)

```
    try {
      for (int i = 0; i < threadCount; i++) {
        integralThreads[i].join();
        myResult += integralWorker[i].myResult;
      }
    } catch (InterruptedException e) { e.printStackTrace(); }

    return myResult;
  }

  public void run() {
    double x = 0.0;
    for (long i = myStart; i < myEnd; i++) {
      x = (i + 0.5) * myStepSize;
      myResult += 4.0 / (1.0 + x * x);
    }
    myResult *= myStepSize;
  }
}
```

Auf Ende jedes Fadens warten und Teilergebnisse einsammeln.

Teilergebnis für den vorgegebenen Bereich berechnen.



Bewertung von parallelen Algorithmen (1)

- Betrachten wir ein Algorithmus mit
  - einem sequentiellen Anteil, der sich nicht parallelisieren lässt und
  - einem parallelisierbaren Rest, der gleichmäßig auf mehrere homogene Prozessoren (bzw. Kernen) aufgeteilt werden kann.
    - σ: Zeit für Ausführung des sequentiellen Teils
    - π: Zeit für Ausführung des parallelen Teils
    - Dann ist die **Gesamtlaufzeit T(p)** auf p Prozessoren (bzw. Kernen):

$$T(p) = \sigma + \frac{\pi}{p}$$

Bewertung von parallelen Algorithmen (2)

- Die **Beschleunigung (engl. Speedup) S(p)** gibt an, um wie viel schneller der Algorithmus mit p Prozessoren im Vergleich zur besten sequenziellen Ausführung wird:

$$S(p) = \frac{T(1)}{T(p)}$$

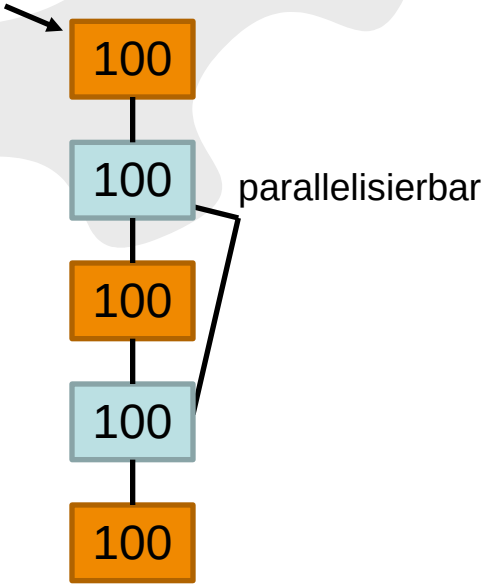
- Die **Effizienz E(p)** gibt den Anteil an der Ausführungszeit an, die jeder Prozessor mit nützlicher Arbeit verbringt:

$$E(p) = \frac{T(1)}{p * T(p)} = \frac{S(p)}{p}$$

- Im Idealfall ist S(p) = p und E(p) = 1

Bewertung von parallelen Algorithmen (3)

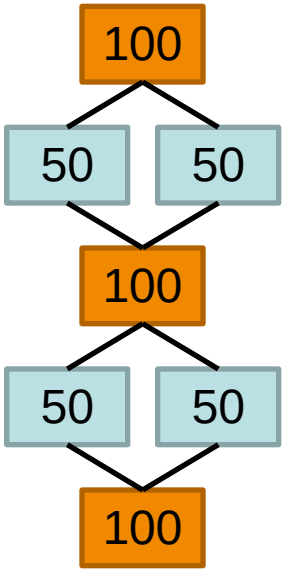
Anweisungsblock
mit Ausführungszeit 100



Seq. Ausführung: T(1): 500
Par. Ausführung: T(1): 500

Beschleunigung

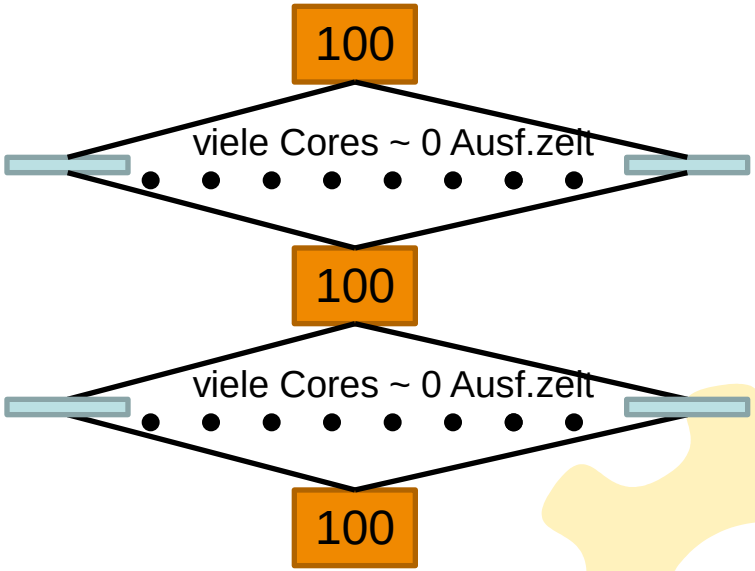
1



Seq. Ausführung: T(1): 500
Par. Ausführung: T(2): 400

Beschleunigung 500/400

1.25



Seq. Ausführung: T(1): 500
Par. Ausführung: T(n): 300

Beschleunigung 500/300=

1.7 maximal

Bewertung von parallelen Algorithmen (4)

- **Amdahls Gesetz:**

$$S(p) \leq \frac{1}{f}$$

- f ist der sequentielle, d.h. nicht parallelisierbare, Anteil des Algorithmus:

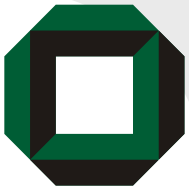
$$f = \frac{\sigma}{\sigma + \pi}$$

Bewertung von parallelen Algorithmen (5)

- **Beispiel:**
- Der sequentielle Anteil an der Ausführungszeit eines Algorithmus beträgt 25 Zeiteinheiten, der parallele Anteil 75 Zeiteinheiten.

$$f = \frac{\sigma}{\sigma + \pi} = \frac{25}{25+75} = 0{,}25$$

$$S(p) \leq \frac{1}{f} = \frac{1}{0{,}25} = 4$$



Ausblick

- Bis jetzt wurden folgende Verfahren zur Parallelisierung von Algorithmen vorgestellt:
  - Gebietszerlegung (siehe Matrix-Vektor- und Matrix-Matrix-Multiplikation)
  - Teile-und-Herrsche (Aufgabe 4, Übungsblatt3: Merge-Sort)
- Neben diesen Verfahren gibt es noch weitere Verfahren zur Ausnutzung der Datenparallelität, auf welche hier nicht weiter eingegangen wird.