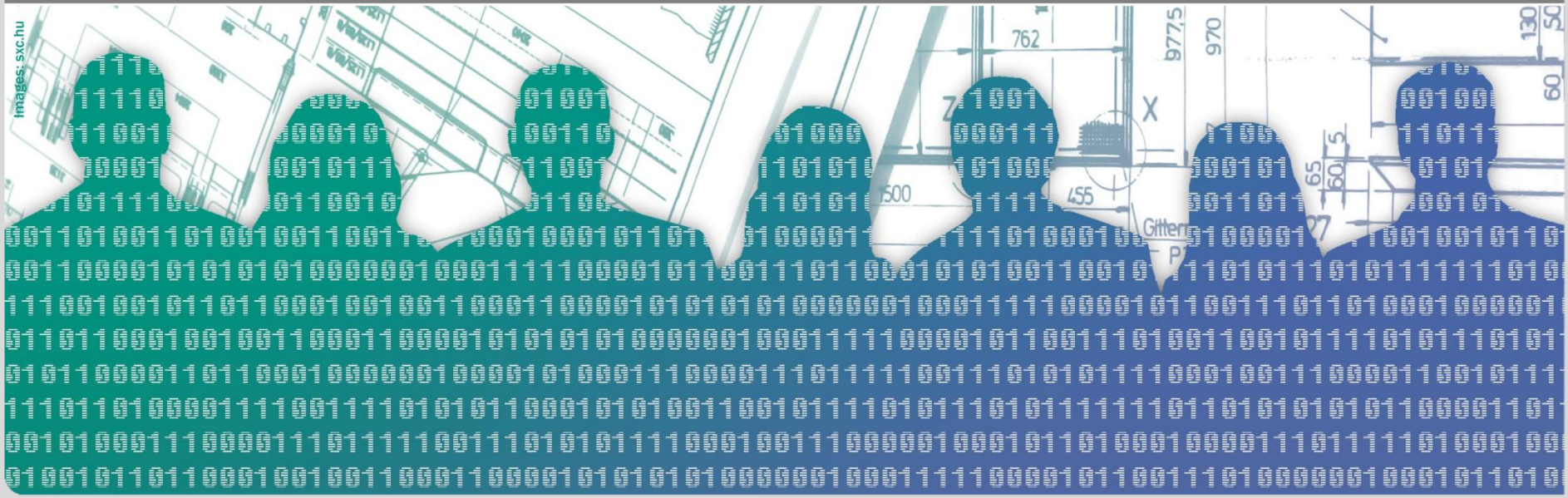
Kapitel 4.2.2 - Parallele Algorithmen

SWT I – Sommersemester 2010

Walter F. Tichy, Andreas Höfer, Korbinian Molitorisz

IPD Tichy, Fakultät für Informatik

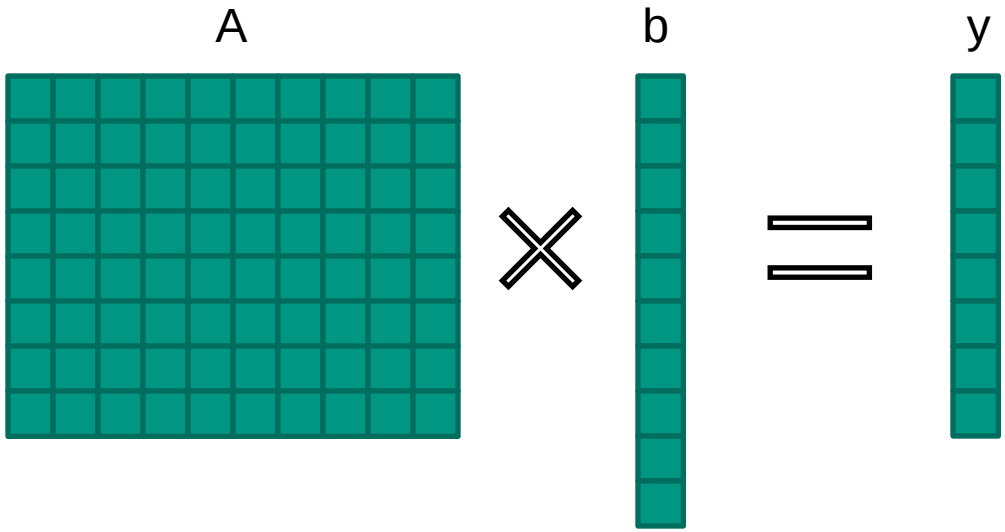


Überblick

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

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: Sequenzieller Algorithmus

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

public int[] multipliziere() {
   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)

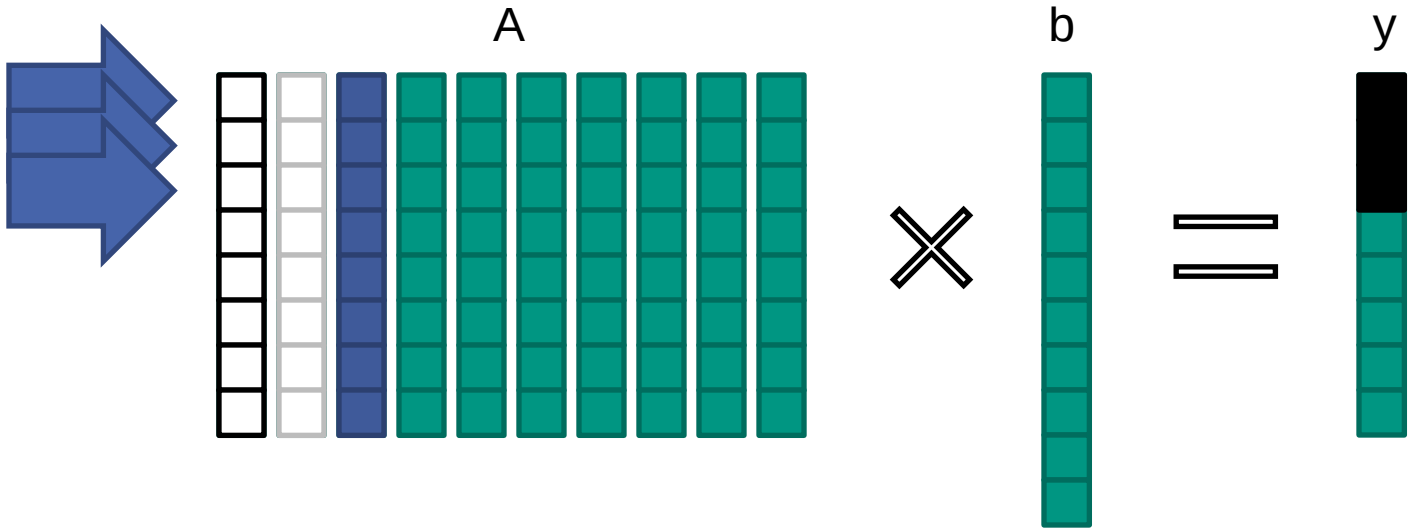
- Annahmen für die folgenden Folien:
  - Variablen (in diesem Beispiel A, b und y) können von allen Prozessoren gleichzeitig gelesen werden.
  - Die beiden Vektoren und die Matrix passen vollständig in den Speicher der Prozessoren.

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.).

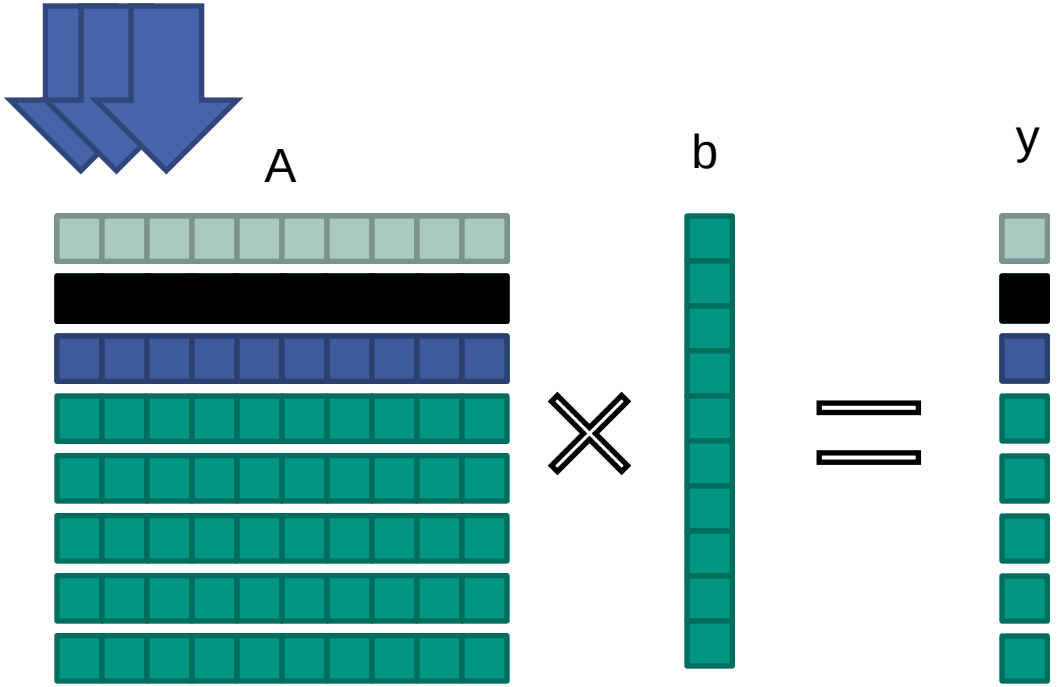
Matrix-Vektor-Multiplikation: Parallelisierung (4)

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



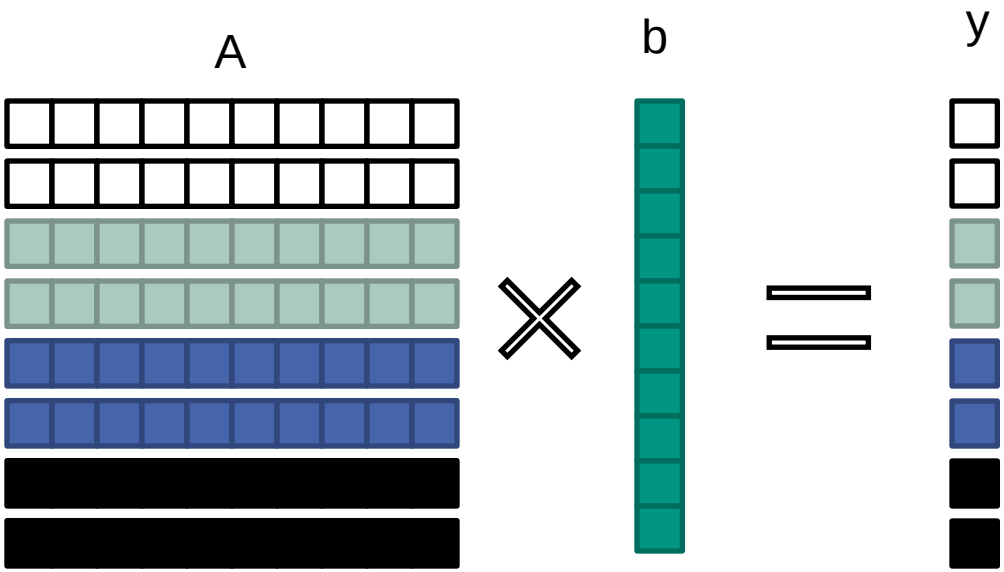
Matrix-Vektor-Multiplikation: Parallelisierung (5)

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



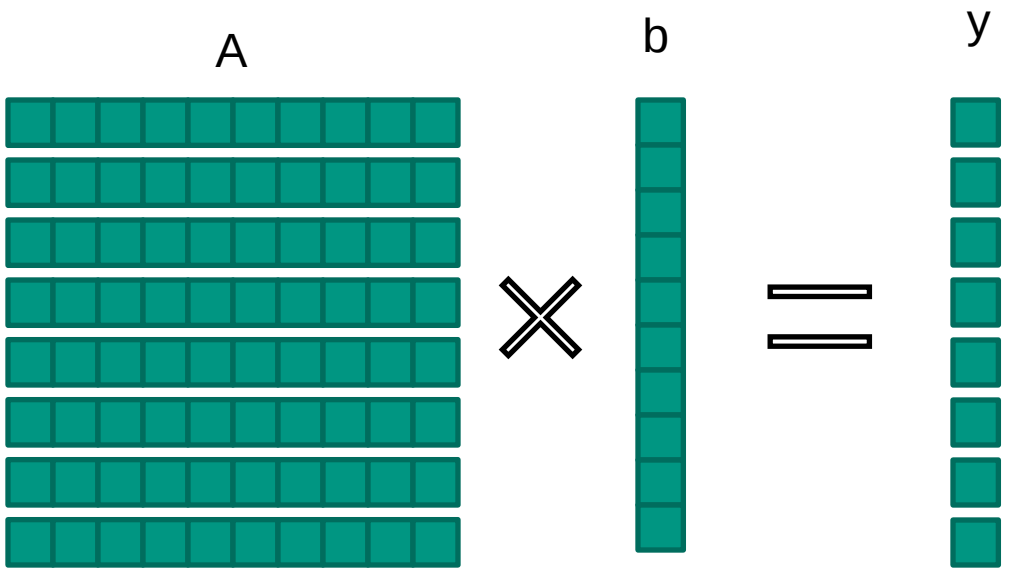
Matrix-Vektor-Multiplikation: Parallelisierung (6)

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



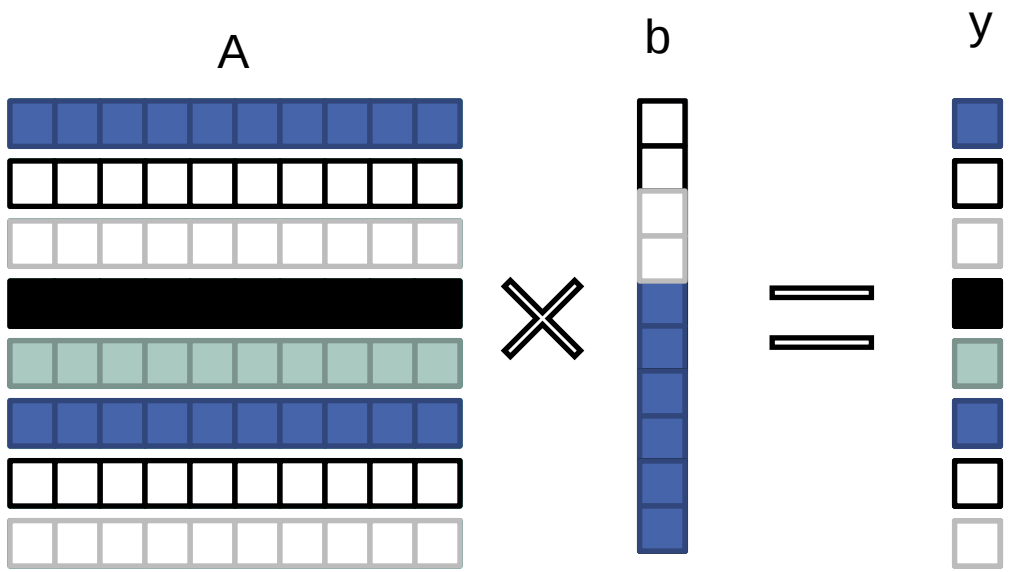
Matrix-Vektor-Multiplikation: Parallelisierung (7)

- 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 MatrixVektorMultiplikation implements Runnable {
  // Variablen für jeden erstellten Faden
  private int[][] meinA = null;
  private int[] meinB = null;
  private int[] meinY = null;
  private int meinStart = 0;
  private int meinEnde = 0;

  private static CyclicBarrier barriere = null;

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

Matrix-Vektor-Multiplikation: Parallele Implementierung (2)

```
private MatrixVektorMultiplikation(int[][] a, int[] b, int[] y,
                                   int start, int ende) {
  this.meinA = a;
  this.meinB = b;
  this.meinStart = start;
  this.meinEnde = ende;
  this.meinY = y;
}
// Führe Multiplikation meinA*meinB auf dem angegebenen Bereich durch
public void run() {
  for (int i = meinStart; i < meinEnde; i++) {
    for (int j = 0; j < meinB.length; j++) {
      meinY[i] += meinA[i][j] * meinB[j];
    }
  }

  try { barriere.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 multiplikation(int[][] a, int[] b, int anzahlFäden) {
  MatrixVektorMultiplikation[] vmmArbeiter = new
    MatrixVektorMultiplikation[anzahlFäden];
  Thread[] vmmFäden = new Thread[anzahlFäden];

  barriere = new CyclicBarrier(anzahlFäden, new Runnable() {
     public void run() {
       System.out.println("Alle Fäden haben die Barriere erreicht!");
       printVektor(meinY);
     }});

  meinY = new int[a.length]; int start = 0; int end = 0;
  int anzahlZeilen = (int) Math.ceil((double) a.length / anzahlFäden);

  for (int i = 0; i < anzahlFäden; i++) {
    start = i * anzahlZeilen;
    ende = Math.min((i + 1) * anzahlZeilen, a.length);

    vmmArbeiter[i] = new MatrixVektorMultiplikation(a, b, meinY, start, ende);
    vmmFäden[i] = new Thread(vmmArbeiter[i]);
    vmmFäden[i].start();
  }
}
```

Schwellwert für die Barriere

Anzahl Zeilen für jeden Faden.

Zeile, ab welcher Faden i multiplizieren soll.

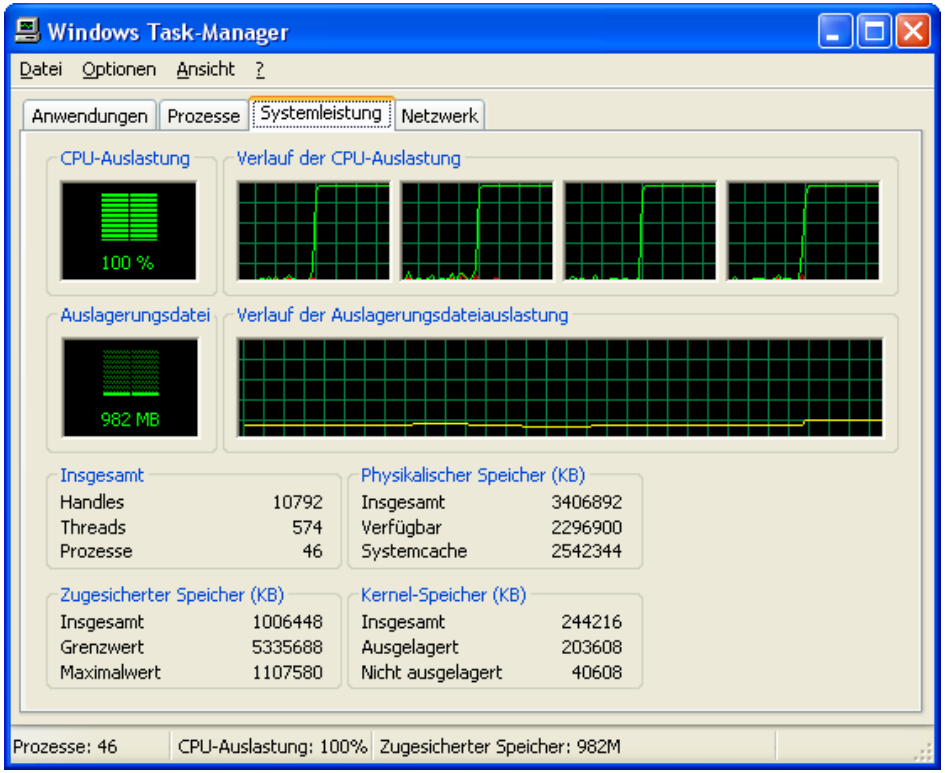
Letzte Zeile für Faden i.

Matrix-Vektor-Multiplikation: Parallele Implementierung (4)

```
  public static void main(String[] args) {
    MatrixVektorMultiplikation mvm = new MatrixVektorMultiplikation();
    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 anzahlFäden = 8;

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

Matrix-Vektor-Multiplikation: Aufteilung der Rechenlast

- Verteilung von 4 Fäden auf 4 Prozessorkerne:



Matrix-Matrix-Multiplikation: Mögliche Algorithmen

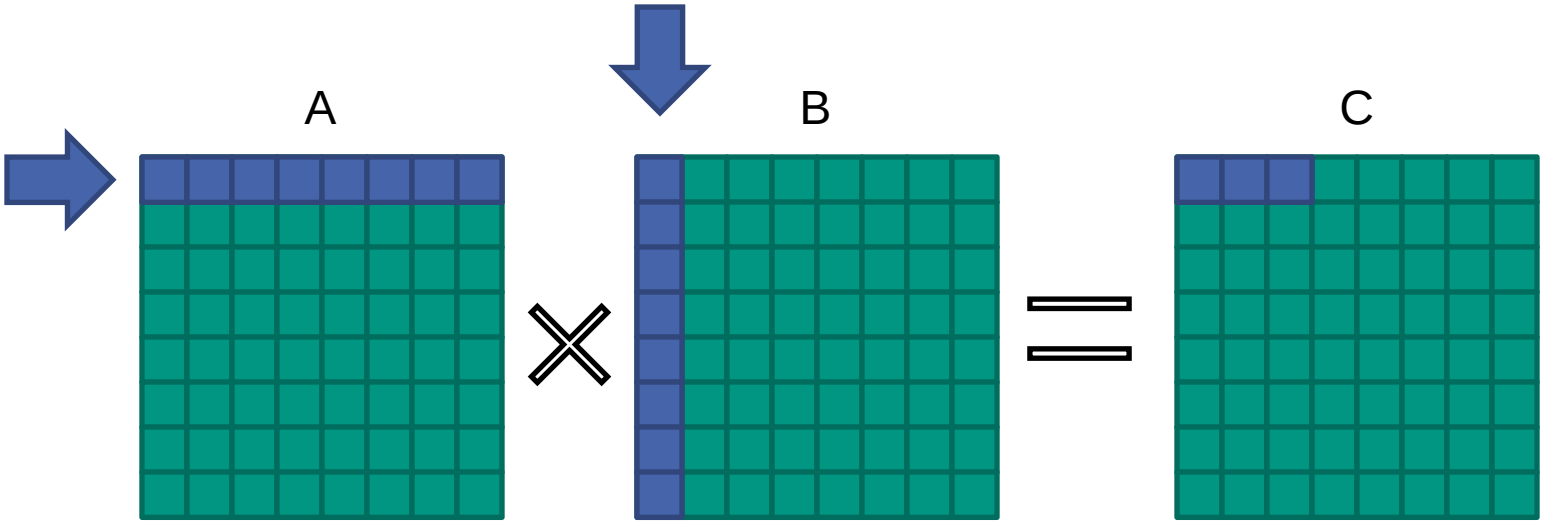
- Zur Multiplikation zweier Matrizen können verschiedene Algorithmen verwendet werden:
  - ijk-Algorithmus
    - Analog dazu: kij-Algorithmus, ikj-Algorithmus
  - Systolischer Algorithmus
  - Cannon-Algorithmus
  - …

Matrix-Matrix-Multiplikation: ijk-Algorithmus

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

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

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

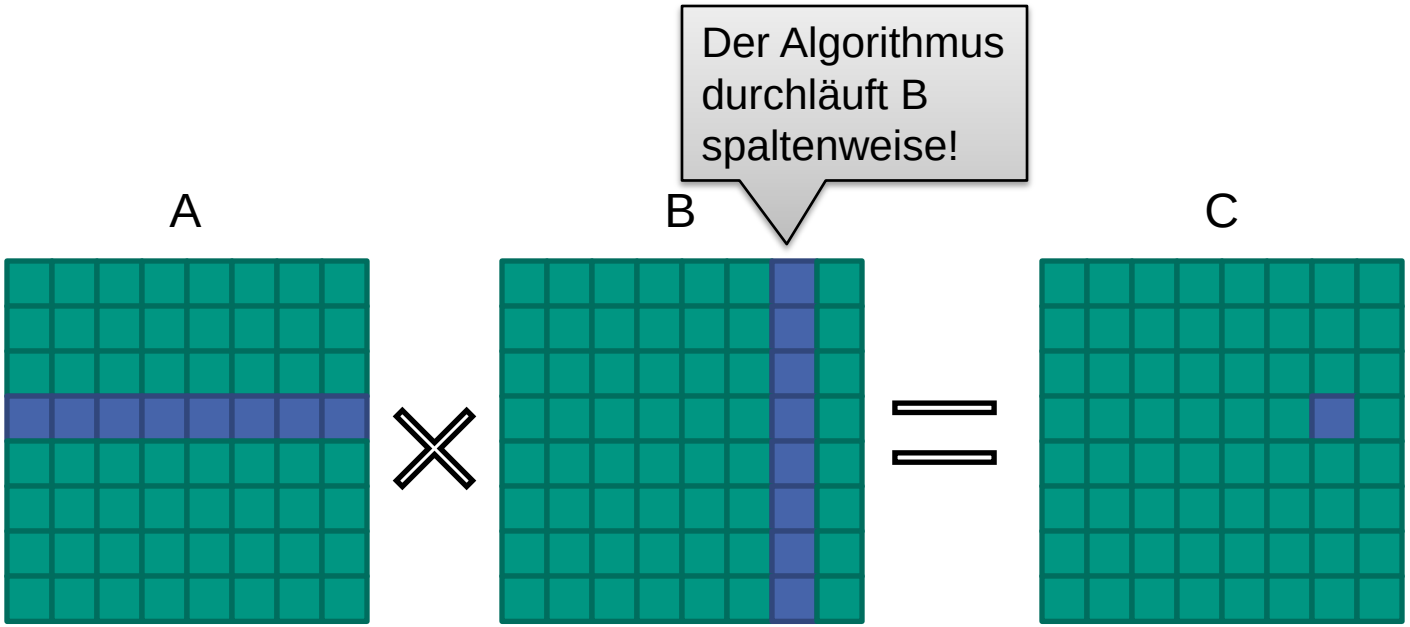


Matrix-Matrix-Multiplikation: ijk-Algorithmus

```
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:

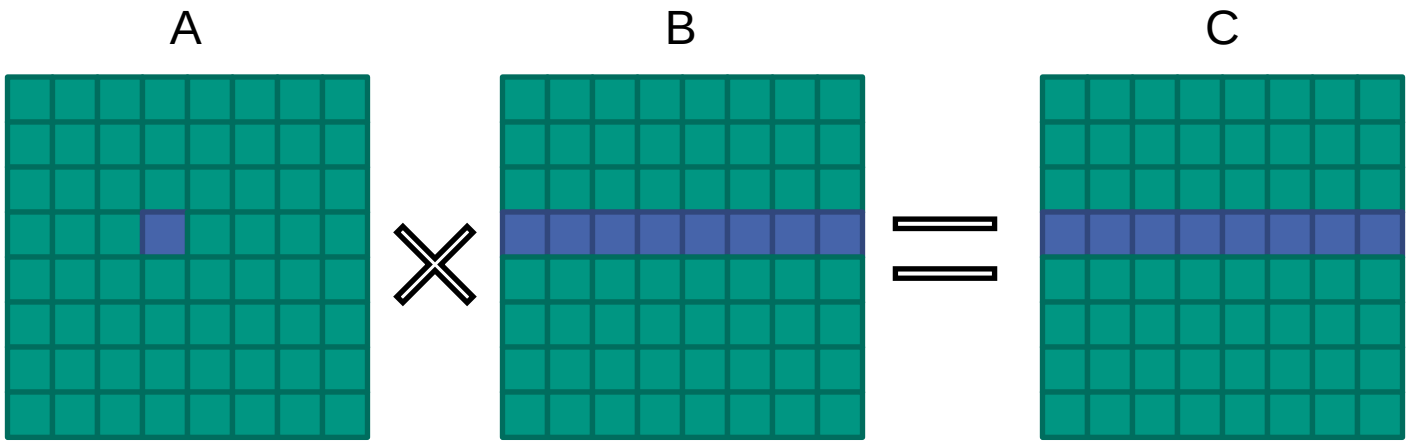
```
for (i=0; i < N; i++)
  for (k=0; k < N; k++)
    for (j=0; j < N; j++)
      c[i][j] += a[i][k] * b[k][j];
```

- 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):

Faktor 2,49

	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

- Maximale Beschleunigung (T(1)/ T(64)):

	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):

Faktor 1,9

	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

- Maximale Beschleunigung (T(1)/ T(64)):

	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.

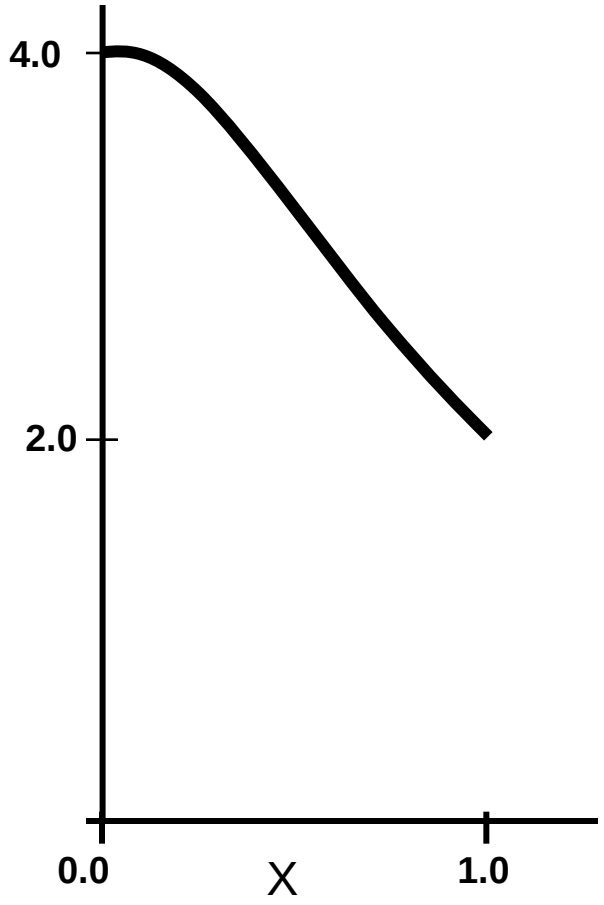
Numerische Integration: Vorgehen (1)

- Gegeben sei folgende Funktion:

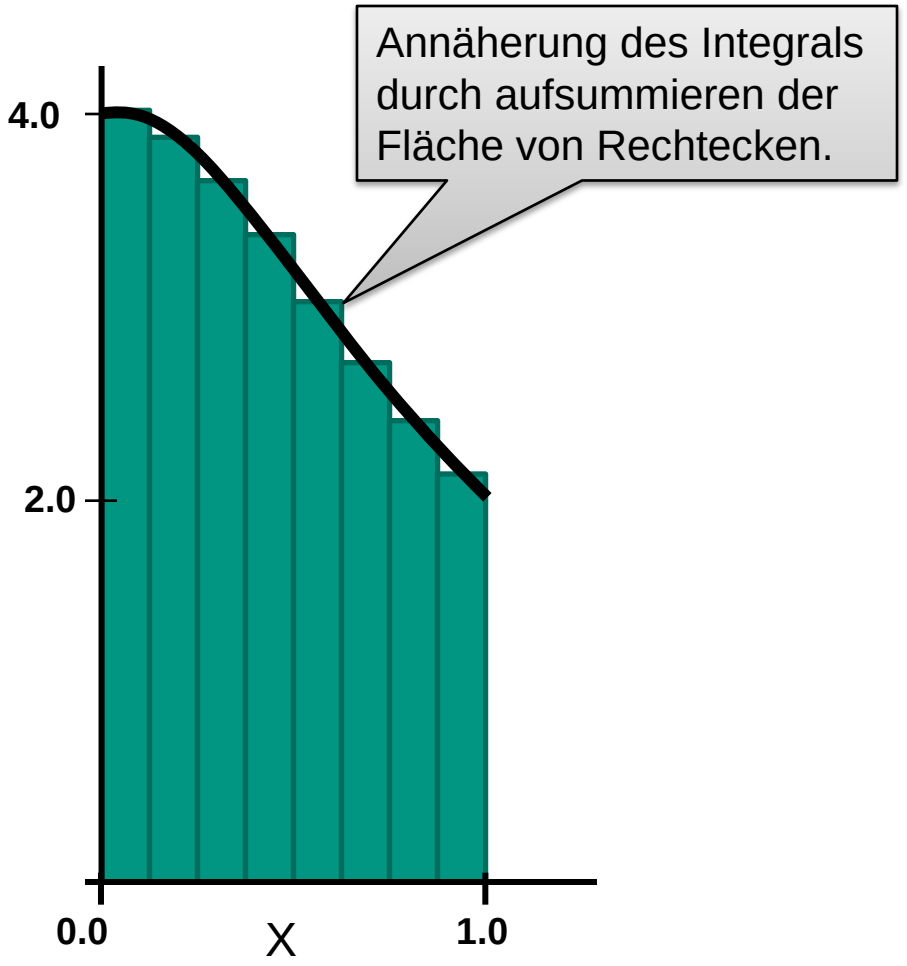
$$f(x) = \frac{4}{1+x^2}$$

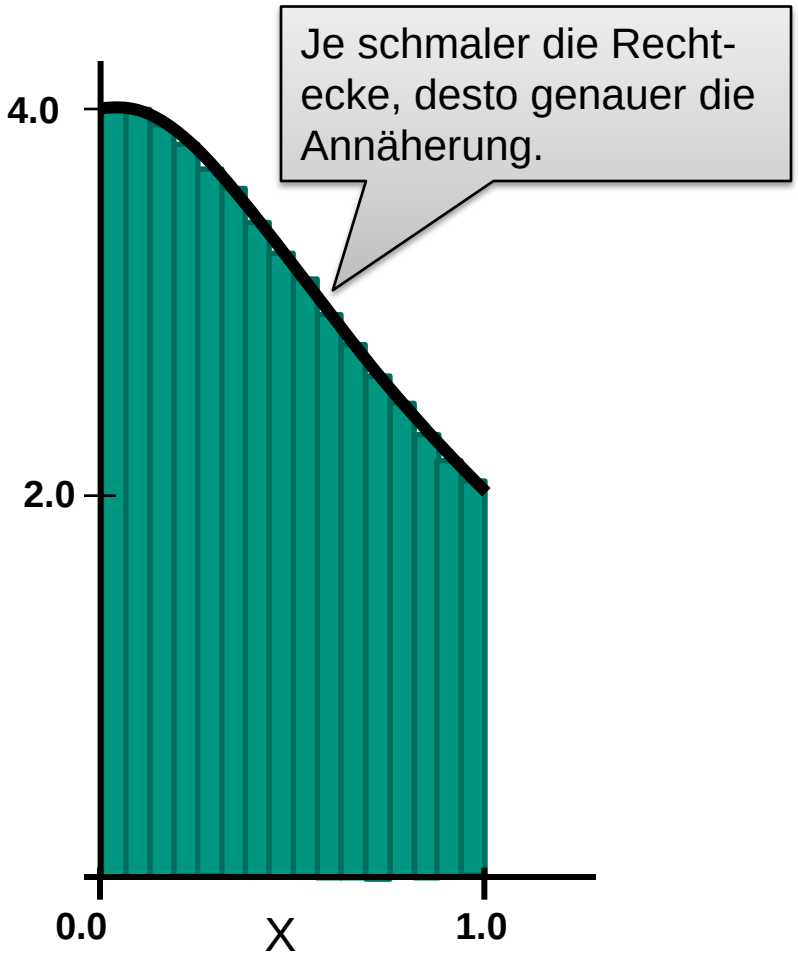
- Gesucht ist folgendes Integral:

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



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 integriere() {
  double x = 0.0;
  double summe = 0.0;

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

  double ergebnis = RECT_SIZE * summe;
  return ergebnis;
}
```

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 meinErgebnis = 0.0;
  private long meinStart = 0;
  private long meinEnde = 0;
  private double meineSchrittweite = 0.0;

  private final static int ANZAHL_FÄDEN = 4;
  private final static long ANZAHL_SCHRITTE = 1000000;

  private Integral() {}

  private Integral(long start, long ende, double schrittweite) {
    this.meinStart = start;   this.meinEnde = ende;
    this.meineSchrittweite = schrittweite;
  }

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

Numerische Integration: Beispiel (2)

```
public double integriere(long schritte, int anzahlFäden) {
  Integral[] integralArbeiter = new Integral[anzahlFäden];
  Thread[] integralFäden = new Thread[anzahlFäden];
  long start = 0; long ende = 0;

  long schritteProFaden = (int)
Math.ceil((double)schritte/anzahlFäden);
  double schrittweite = 1.0 / (double) sc

  for (int i = 0; i < anzahlFäden; i++) {
    start = i * schritteProFaden;
    ende = Math.min((i + 1) * schritteProFaden, schritte);

    integralArbeiter[i] = new Integral(start, end, schrittweite);
    integralFäden[i] = new Thread(integralArbeiter[i]);
    integralFäden[i].start();
  }
```

Anfang und Ende für jeden Faden berechnen.

Numerische Integration: Beispiel (3)

```
    try {
      for (int i = 0; i < anzahlFäden; i++) {
        integralFäden[i].join();
        meinErgebnis += integralArbeiter[i].meinErgebnis;
      }
    } catch (InterruptedException e) { e.printStackTrace(); }

    return meinErgebnis;
  }

  public void run() {
    double x = 0.0;
    for (long i = meinStart; i < meinEnde; i++) {
      x = (i + 0.5) * meineSchrittweite;
      meinErgebnis += 4.0 / (1.0 + x * x);
    }
    meinErgebnis *= meineSchrittweite;
  }
}
```

Auf Ende jedes Fadens warten und Teilergebnisse einsammeln.

Teilergebnis für den vorgegebenen Bereich berechnen.

Bewertung von parallelen Algorithmen (1)

- 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

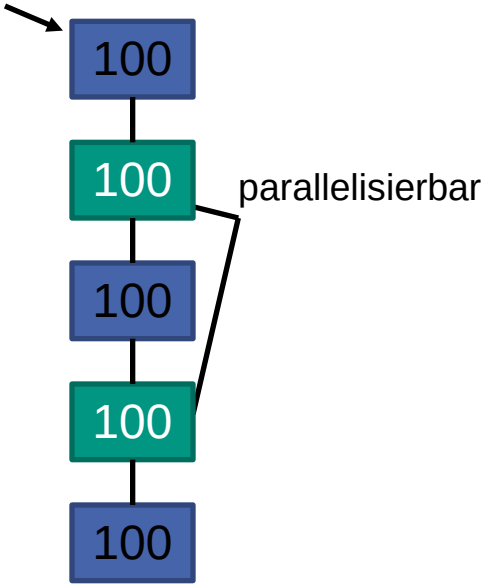
Bewertung von parallelen Algorithmen (2)

- 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 die sequentielle Ausführung des parallelisierbaren Teils
    - Dann ist die **Gesamtlaufzeit T(p)** auf p Prozessoren (bzw. Kernen):

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

Bewertung von parallelen Algorithmen (3)

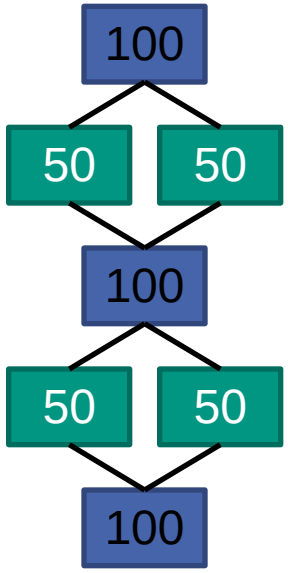
Anweisungsblock
mit Ausführungszeit 100



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

Beschleunigung

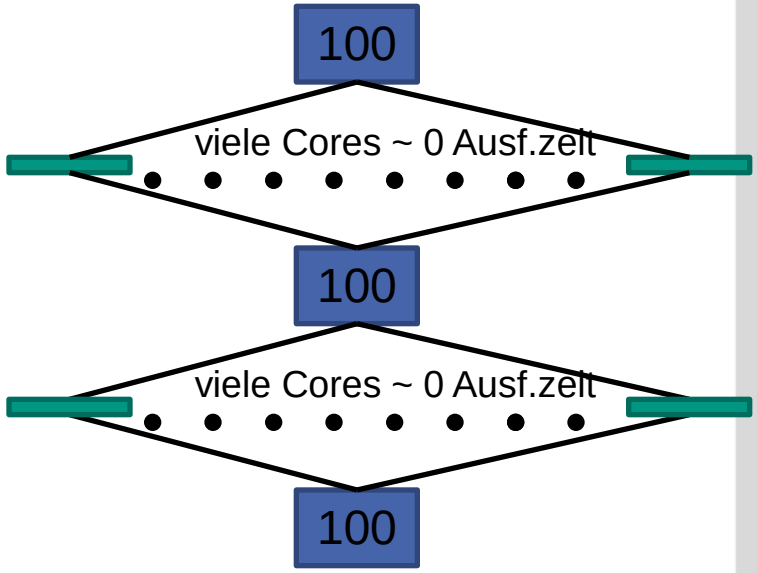
1



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:

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

Dateiindizierung

- Problem: Erzeuge einen invertierten Index aller Dateien
  - Für jedes Wort (Term) liefert der invertierte Index die Liste der Dateien, die das Wort enthalten
  - Inv. Index wird erzeugt, in dem man alle Dateien durchsucht und für jeden Term t in Datei f das Paar (t,f) zum Index hinzufügt
  - Dann kann für ein geg. Wort schnell nach Dateien gesucht werden

  - Wie und was parallelisieren?
  - Was dauert am längsten? Ist evtl. die Platte der Flaschenhals?

Dateiindizierung: Ohne Daten geht's nicht!

- Erzeuge eine Testmenge an Dateien zur Messung
  - Testfallmenge: 50,000 Dateien, 900MB, nur reine Textdateien, ohne spezielle Formate wie pdf, docx, etc.)
- Baue sequenziellen Indizierer
- Vermesse seq. Indizierer auf Zielplattform
- Beispiel: 4-Kern-Plattform
  - Dateinamen-Erzeugung: 5 s
  - Reines Lesen, Zeichen für Zeichen, der Dateien (ohne Termextraktion): 77 s
  - Zusatz für Termextraktion: 11 s
  - Indexaktualisierung: 22 s
- Offensichtlich ist die Platte noch nicht der Flaschenhals, also könnte sich Parallelisierung lohnen.

Warum nicht?

Dateiindizierung

- Idee: Setze mehrere Fäden für die Termextraktion ein
- Anzahl der Fäden unbekannt
  - Durch Experimente zu bestimmen
- Dateinamen → Termextraktion kritisch:
  - Ein synchronisierter Puffer zwischen Dateinamenerzeuger und Termextraktor war sehr langsam (warum?)
    - Statt dessen:
      - Dateinamenerzeuger speichert die Dateinamen im Umlaufverfahren in n Puffer, wobei n die Zahl der Termextraktoren ist
      - Dateinamenerzeuger läuft sequenziell und komplett vor Termextraktion. Vorteil: Keine Synchronisation
  - Lastbalance (große Dateien brauchen längere Zeit zur Bearbeitung als kurze) war kein Problem wegen der großen Zahl an Dateien

Dateiindizierung

- Kommunikation zwischen Termextraktoren → Indexaktualisierung ebenfalls sehr langsam
  - Lösung: Übergabe aller Terme einer Datei anstatt einzelner Terme
- Indexaktualisierung ist selbst eine Flaschenhals
  - Lösung: Benutzung mehrerer Indexaktualisierer, die unabhängig voneinander laufen
- Erzeugte Indizes können...
  - am Schluss verschmolzen werden (Reduktion) oder
  - unverschmolzen verbleiben und parallel durchsucht werden.

Dateiindizierung, beste Konfigurationen

Rechner	Dateinamen-erzeuger	Termextraktion	Index-aktualisierung	Verschmelzung	Beschleunigung
4 Kerne	1	3	2	0	**4,7**
8 Kerne	1	6	2	0	**2,1**
32 Kerne	1	9	4	0	**3,5**

Anzahl an Fäden in Konfiguration

- Anmerkung:
  - Bei der Verschmelzung der Indizes sind die Beschleunigungswerte etwas kleiner

Ausblick

- Bis jetzt wurden folgende Verfahren zur Parallelisierung von Algorithmen vorgestellt:
  - Gebietszerlegung (siehe Matrix-Vektor- und Matrix-Matrix-Multiplikation)
  - Auftraggeber/Auftragenehmer (Master/Worker) (siehe Entwurfsmuster)
  - Erzeuger/Verbraucher mit Puffer
  - Fließband mit Puffern zwischen Stufen und eigenen Fäden pro Stufe (siehe Entwurfsmuster)
  - Dateiindizierung
- Neben diesen Verfahren gibt es noch weitere Verfahren zur Parallelisierung
  - Paralleles Teile-und-Herrsche: Teilaufgaben werden parallel verarbeitet, bis zu einer gewissen Zerlegungstiefe oder Problemgröße, ab dann sequenziell (z.B. paralleles Verschmelzungssortieren).
  - Map/Reduce: Eine Berechnung wird an einer Menge von Elementen ausgeführt, dann die Ergebnisse zusammengeführt (z.B. addiert)