

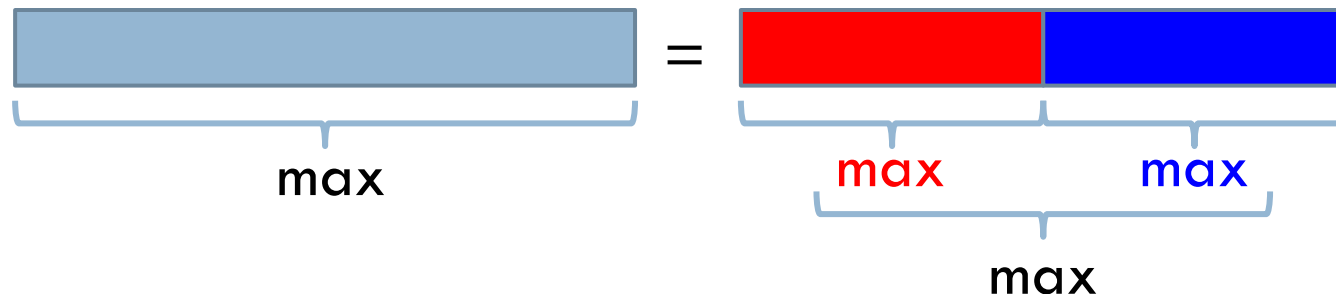
Das **MapReduce** Muster

Parallele Reduktion, MapReduce
Parallelisierungsmuster, Cloud Computing,
Beispiele: Maximumsbestimmung, Wortzählung

Beispiel 1: Maximumsbestimmung

2

- Maximumsbestimmung
 - ▣ Bestimme das Maximum in einer (langen) Werteserie
- Sequenzielle Version
 - ▣ Setze $\text{max} = -\infty$ // **long.MinValue**
 - ▣ Inspiziere ein Element nach dem anderen
 - Falls $\text{Wert} > \text{max}$ ersetze max durch Wert
- „Divide et Impera“ und **Reduktionsschritt**



Reduktionsprogramm mit TPL

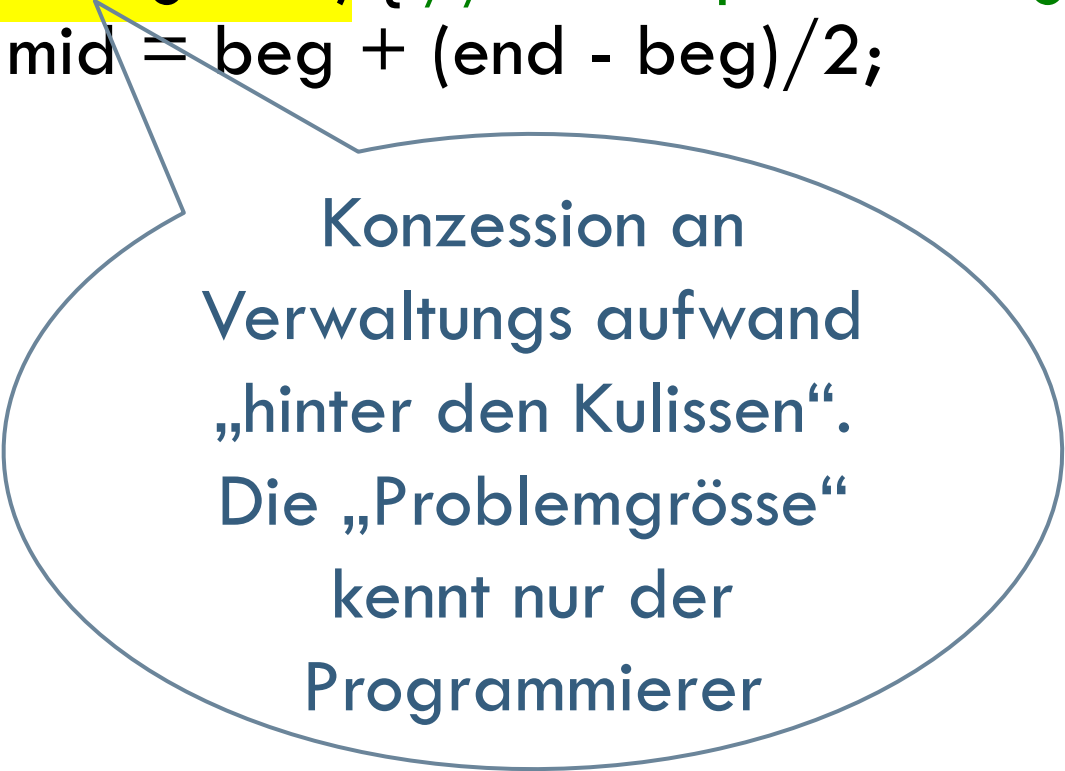
3

```
□ static void CompMax (  
    long beg, long end, out double max) {  
    if (end > beg + ...) { // worth partitioning  
        long mid = beg + (end - beg)/2;  
        double maxL = -∞, maxR = -∞;  
        Parallel.Invoke(  
            () => CompMax(beg, mid, out maxL),  
            () => CompMax(mid, end, out maxR));  
        max = (maxL > maxR)? maxL : maxR; }  
    else { max = -∞; // do it directly  
        for (long i = beg; i < end; i++)  
            if (a[i] > max) max = a[i]; }}
```

Problemgrösse als Kriterium

4

```
□ static void CompMax (  
    long beg, long end, out double max) {  
    if (end > beg + ...) { // worth partitioning  
        long mid = beg + (end - beg)/2;
```



Konzession an
Verwaltungs aufwand
„hinter den Kulissen“.
Die „Problemgrösse“
kennt nur der
Programmierer

Verallgemeinerung

5

- Parallele Reduktion nach divide et impera funktioniert analog für alle *assoziativen Operatoren* op
 - ▣ Falls $A = A1 \cup A2$ dann $op A = (op A1) op (op A2)$

Map & Reduce = MapReduce

6

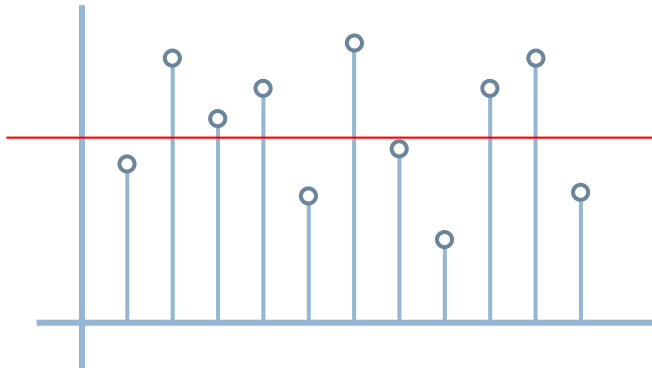
□ Kombination zweier Parallelisierungsmuster

□ $\text{out} = \mathbf{f}(\text{in}_1) / \mathbf{f}(\text{in}_2) / \mathbf{f}(\text{in}_3) / \dots / \mathbf{f}(\text{in}_n)$

■ $\mathbf{f} = \text{map}, / = \text{reduce (assoziativ)}$

□ Beispiel

□ $(f(x_1) + f(x_2) + \dots + f(x_n)) / n$

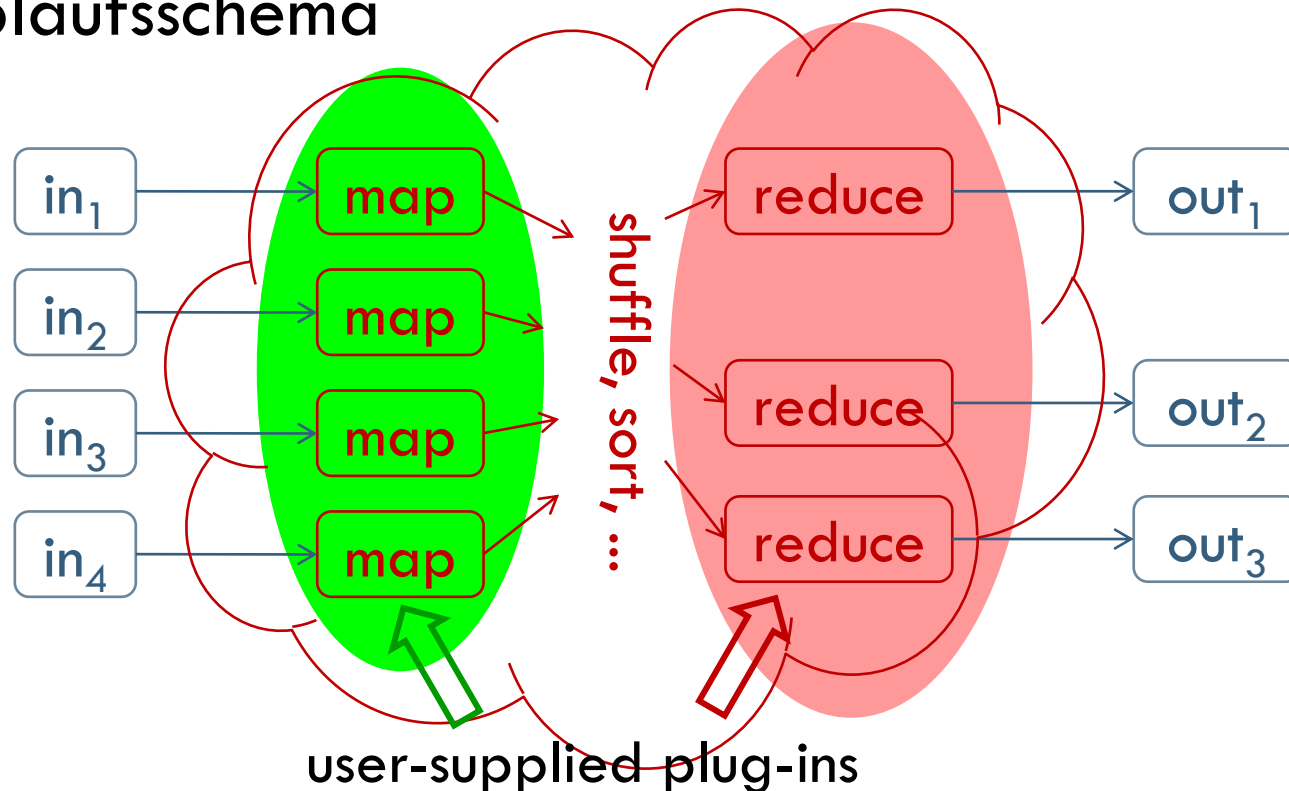


Das MapReduce Modell

MapReduce: Simplified Data Processing on Large Clusters, Jeffrey Dean & Sanjay Ghemawat, Google, Inc.

7

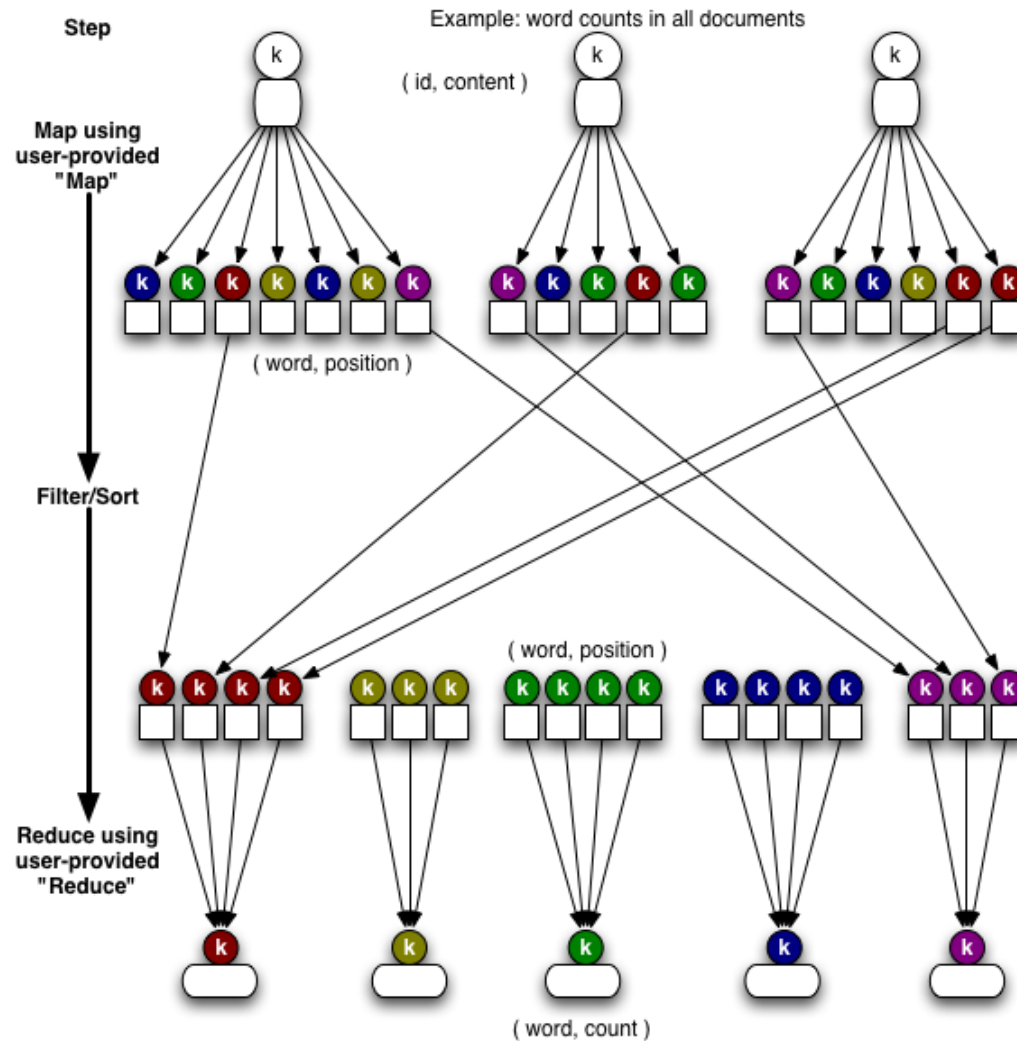
- **map** (k_1, v_1) \rightarrow list (k_2, v_2) (k_i keys, v_i values)
- **reduce** ($k_2, \text{list}(v_2)$) \rightarrow list (v_2) (kleinere Liste)
- Ablaufschema



Beispiel 2: Wortzählung

Ubolonton

8




Map, Reduce Plug-Ins


9

- Bestimmung der Anzahl Vorkommen jedes Wortes in einer grossen Menge von Dokumenten

□ **void** **map** (string id, string content) {
 foreach (word w in content) EmitIntermediate(w, pos);
}



□ **void** **reduce** (string word, IEnumerable occList) {
 int count = 0; // number of occurrences
 foreach (n in occList) count++;
 EmitResult(word, count);
}



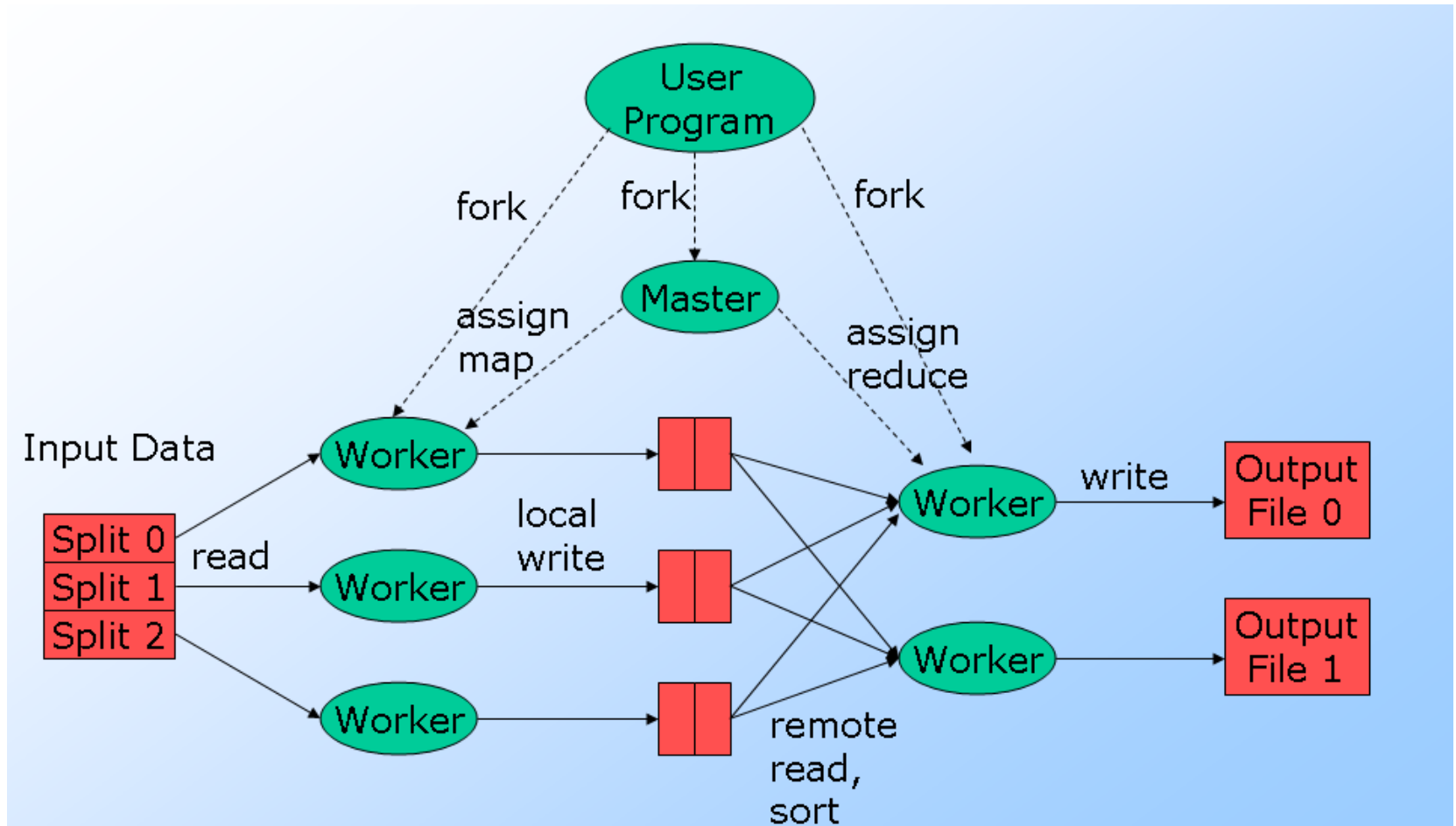
Weitere Anwendungsbeispiele

10

- Bestimmen der Zugriffsverteilung auf URLs
 - ▣ *map* verarbeitet Aktivitätslogs und erstellt (URL, pos) Paare
 - ▣ *reduce* summiert die Zugriffe per URL
- Indexinvertierung
 - ▣ *map* extrahiert Wörter aus Dokumenten und erstellt (word, documentID) Paare
 - ▣ *reduce* erstellt (word, list(documentID)) Paare, sortiert nach documentID

MapReduce Ablauf in der Cloud

11



Shared Memory Modelle

Shared Memory, Race Conditions, Cache Kohärenz, False Sharing, Stale Values, Zugriffssynchronisation, Atomizität, Interlocked Operationen, Beispiele: Parallele EightQueens, Rucksackfüllung, Branch-and-Bound

Shared Memory Computing Modell

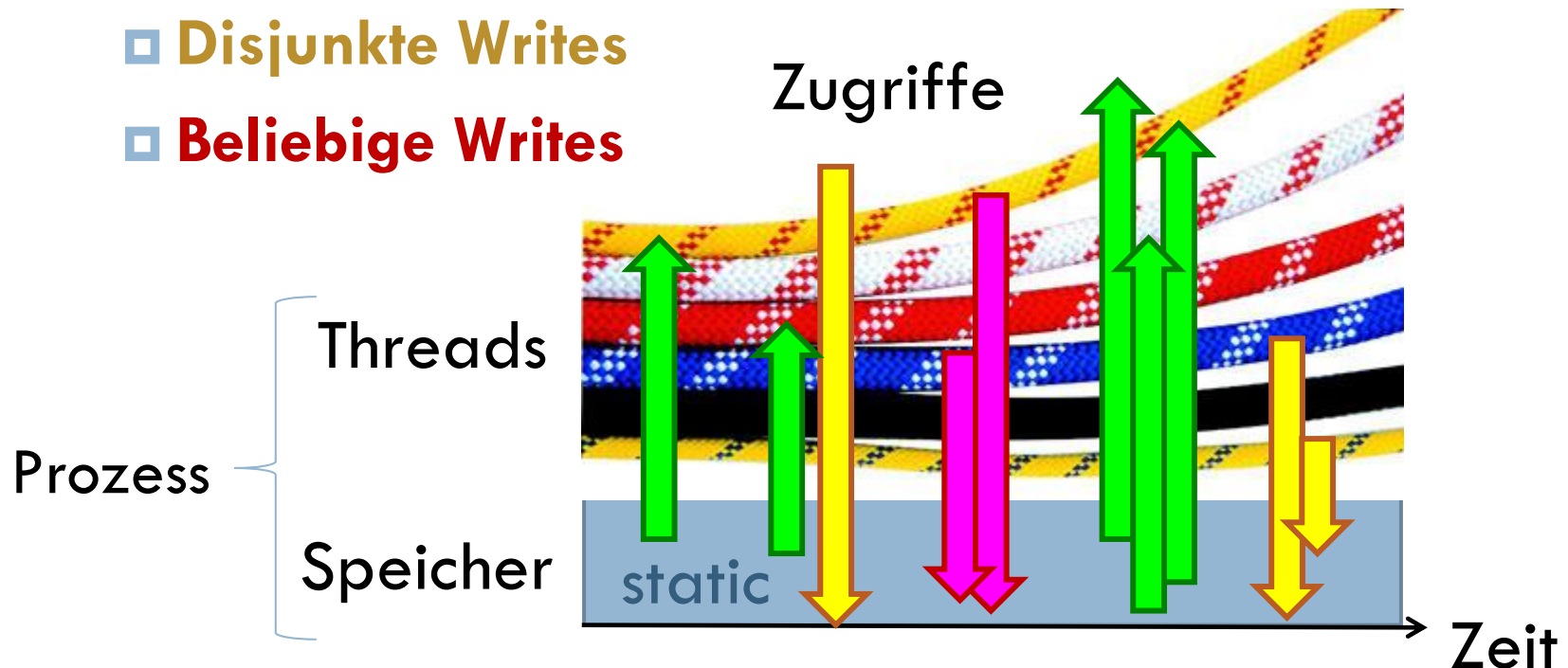
13

- Prozess: Menge von Threads in Lese-/Schreibinteraktion mit der Umgebung (*static*)

- **Reads**

- **Disjunkte Writes**

- **Beliebige Writes**



Problem der *Race Conditions*



14

- Threads T_1 und T_2 führen je eine `i++` Operation auf der Variablen `i` aus
- Mögliche Szenarien mit dem Mikroskop betrachtet

■ Szenarium 1

■ $R_1 \leftarrow i$	} nacheinander
■ $\text{inc}(R_1)$	
■ $R_1 \rightarrow i$	
■ $R_2 \leftarrow i$	
■ $\text{inc}(R_2)$	
■ $R_2 \rightarrow i$	

ok

Szenarium 2


$R_1 \leftarrow i$	} interleaved
$\text{inc}(R_1)$	
$R_2 \leftarrow i$	
$R_1 \rightarrow i$	
$\text{inc}(R_2)$	
$R_2 \rightarrow i$	

f

Atomare Speicherzugriffe



15

	Operation	Synchronisationskonstrukt
Beliebige Granularität	Kritischer Codebereich	Softwarekonstrukte <ul style="list-style-type: none">• Semaphore, Lock, Monitor• Transaktion
Feine Granularität	{ Read; Inspect; Write } 	Maschineninstruktionen <ul style="list-style-type: none">• Test-And-Set• Compare-And-Swap• Interlocked Increment etc.
Feinste Granularität	Read, Write	Memory Controller <ul style="list-style-type: none">• Bus Arbiter

Interlocked Operationen

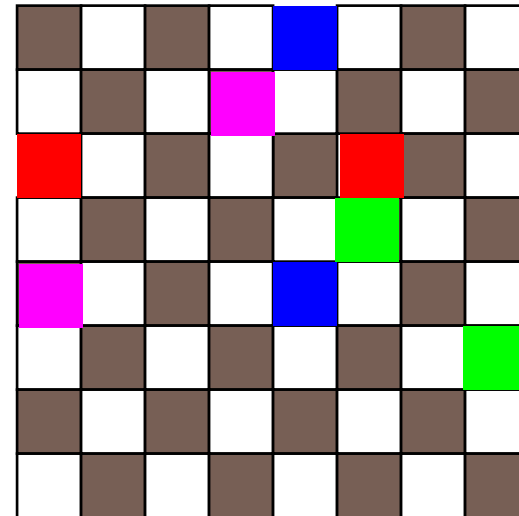
16

- Atomares Inkrementieren
 - ▣ `Interlocked.Increment(ref c)`
 - ▣ Atomar: Erhöhe Zähler bei `c` um 1
- Atomares Vergleichen & Ersetzen
 - ▣ `Interlocked.CompareExchange (ref x, new, exp)`
 - ▣ Atomar: Vergleiche Wert bei `x` mit `exp`; falls gleich, ersetze diesen Wert durch `new`

Beispiel 1: Eight-Queens Problem

17

- Das Acht-Damenproblem (Eight Queens)
 - ▣ Wieviele Konstellationen mit 8 Schachdamen auf einem Schachbrett ohne gegenseitige Bedrohung gibt es?
 - ▣ Konfliktarten
 - Zeilenkonflikt
 - Spaltenkonflikt
 - Hauptdiagonalenkonflikt
 - Nebendiagonalenkonflikt



Algorithmus

18

□ Strategie



- **Branch-And-Bound (B&B)**, spaltenweise

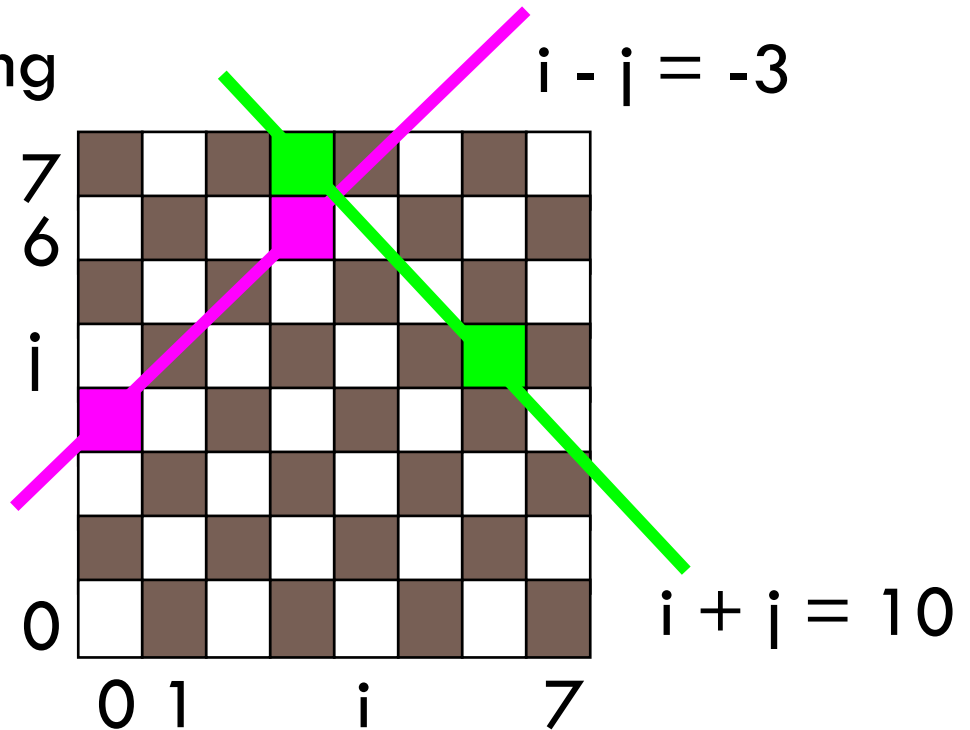
□ Diagonalencodierung

- Hauptdiagonale (diag1)

■ $i - j = -7 \dots 7$

- Nebendiagonale (diag2)

■ $i + j = 0 \dots 14$



Sequenziell B&B ($N = \#$ Spalten)

19

```
□ static void TryNextCol (int n, QueensRequest r) {  
    for (int i = 0; i < N; i++) {  
        if (r.rowFree[i] && r.diag1Free[n - i + N - 1]  
            && r.diag2Free[n + i]) { r.row[n] = i;  
            r.rowFree[i] = false;  
            r.diag1Free[n - i + N - 1] = false;  
            r.diag2Free[n + i] = true;  
            if (n != N) TryNextCol(n + 1, r); // shared via ref  
            else nofSol++; // globaler Lösungszähler  
            r.diag2Free[n + i] = true;  
            r.diag1Free[n - i + N - 1] = true;  
            r.rowFree[i] = true; }}  
}
```

Parallel B&B (N = # Spalten)

20

```
□ static void TryNextCol (int n, QueensRequest r) {  
    List<Task> locTasks = new List<Task>();  
    for (int i = 0; i < N; i++) {  
        if (r.rowFree[i] && r.diag1Free[n - i + N - 1]  
            && r.diag2Free[n + i]) { ...  
            if (n != N) {  
                QueensRequest qr = new QueensRequest(r);  
                locTasks.Add(Task.Factory.StartNew(  
                    () => TryNextCol(n + 1, qr))); }  
            else Interlocked.Increment(ref nofSol);  
            ...  
        }  
    }  
    Task.WaitAll(locTasks.ToArray()); }
```

Clone

Barriere

Atomares
Zählen

Laufzeitvergleich

21

- Die parallele Version (auf Dual Core mit Hyperthreading) läuft viel langsamer als die sequenzielle Version!

Anzahl Lösungen	Sequenzielle Version	Parallele Version	Speedup
92	4431	20675	0.21

- Hauptgrund: Die geleistete Arbeit pro Task im Verhältnis zum Verwaltungsaufwand zu gering

Cutoff und grössere Schachbretter

22

- Mit cutoff der Parallelität bei Spalte 2
- $N = 8, 10, 12, 14$

N	Anzahl Lösungen	Sequenziel le Version	Parallele Version	Speedup
8	92	2661	52162	0.051
10	724	16829	56613	0.30
12	14200	358979	241924	1.48
14	365596	12200496	4968290	2.46

Optimierter Algorithmus

23

- A priori Beschränkung auf Permutationen
- Start with permutation $\{ 1, 2, \dots, N-1, 0 \}$;
 do {
 check current permutation for conflicts;
 if no conflict increase *nofSol*;
 iterate to next permutation }
 while (not at starting permutation again)

Optimierungsebenen

24

□ Optimierungsebenen

- Algorithmen

- Parallelisierung

□ Vergleich

Algorithmus	Sequenzielle Version	Parallele Version	Speed up
original	1 220 049 6	4 968 290	2.46
optimiert	673 274 3	2 779 472	2.42
Speedup		1.79	4.34

Algorithmen Diskussion

25

- Verbesserung des Algorithmus durch Beschränkung der Probekonfigurationen auf Permutationen
- Laufzeit serielle vs. parallele Version (mit cutoff)

Anzahl Lösungen	Sequenzielle Version	Parallele Version	Slowdown
92	2865	73025	25.5

Beispiel 2: Das Rucksackproblem

26

- Gegebene Menge von Gegenständen mit Attributen *Gewicht* und *Wert*
- Bestimme die Rucksackpackung maximalen Wertes unter Berücksichtigung einer *Gewichtslimite*
- Naheliegende Verfahren funktionieren nicht

Gegenstand	Gewicht	Wert	Limite
0	70	75	80 ✓
1	45	45	100
2	45	45	

Verwende Branch-And-Bound

27

□ Traversiere Lösungsbaum

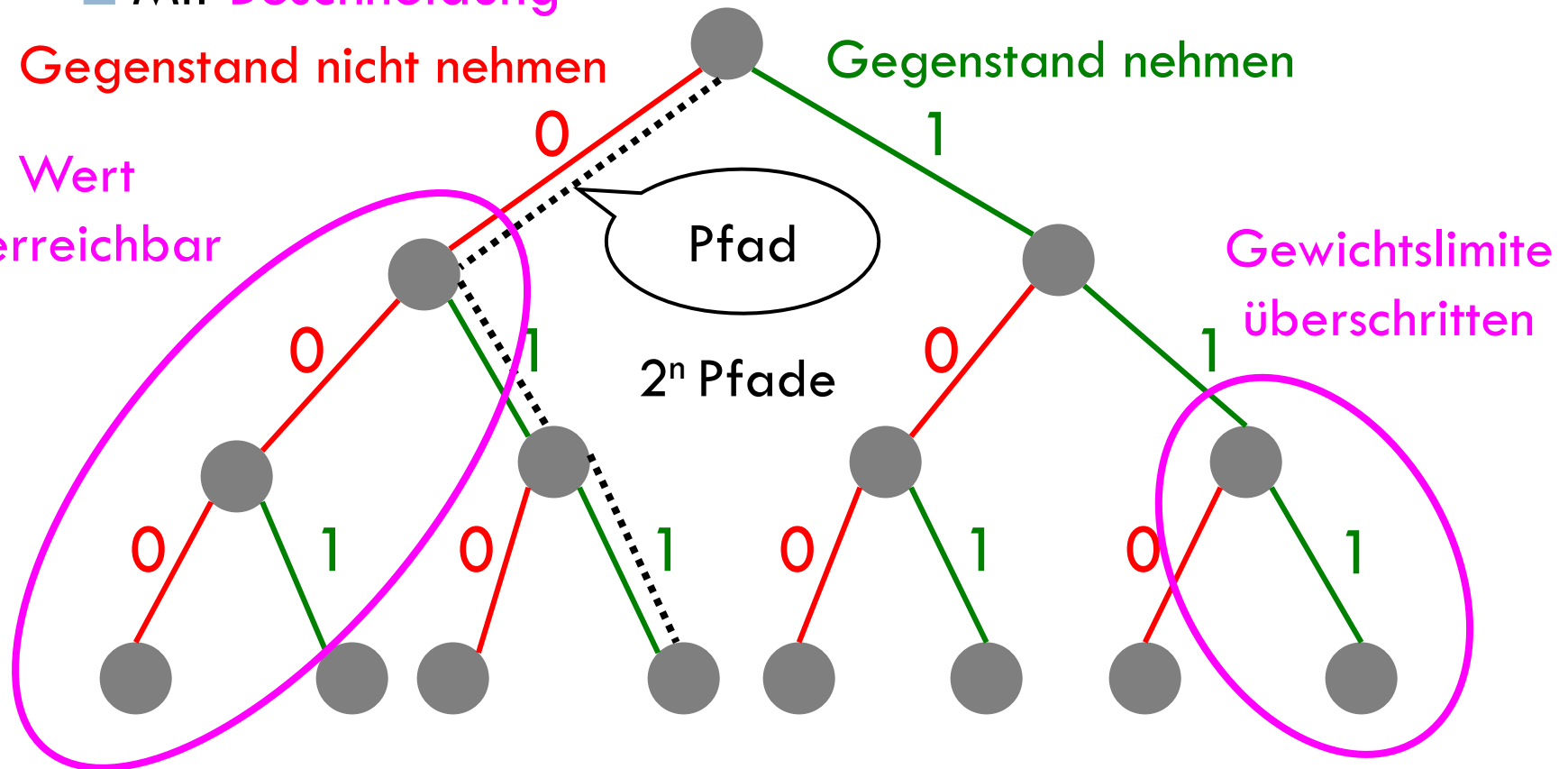
▣ Mit Beschneidung

Gegenstand nicht nehmen

Gegenstand nehmen

Wert
unerreichbar

Gewichtslimite
überschritten



Datenstruktur

28

```
□ class ItemSet {  
    public ItemSet() {  
        elems = new bool[nofItems]; }  
    public ItemSet(ItemSet s) { // cloning  
        elems = new bool[nofItems];  
        for (int i = 0; i < nofItems; i++) elems[i] = s.elems[i];  
    }  
    public bool [] elems;  
}  
  
□ class Cand {  
    public Cand(ItemSet s, double v, double w) {  
        this.s = new ItemSet(s); this.v = v; this.w = w; }  
    public ItemSet s;  
    public double v, w;  
}
```

Umgebung

29

```
□ static Cand maxCand;  
  static double[] weight = {  
    28.0, 39.5, 12.8, 45.0, 17.3,  
    ...  
  };  
  static double[] val = {  
    150.8, 390.5, 44.0, 210.0, 79.0,  
    ...  
  };
```

Sequenzieller Programmkernel

30

```
□ static void TryItem(int i, ItemSet s,  
    double av, double w) { // av = still attainable value  
    if (av - val[i] > maxCand.v) { s.elems[i] = false;  
        if (i == nofItems - 1)  
            maxCand = new Cand(s, av - val[i], w);  
        else TryItem(i + 1, s, av - val[i], w); }  
    if (w + weight[i] <= wmax) { s.elems[i] = true;  
        if (i == nofItems - 1) {  
            if (av > maxCand.v)  
                maxCand = new Cand(s, av, w + weight[i]); }  
        else TryItem(i + 1, s, av, w + weight[i]);  
    }  
}
```

Interaktionen mit der Umgebung

31

- Spezifikationen sind als globale Konstanten *nofItems*, *val*, *weight*, *wmax* realisiert
- Bisher beste gefundene Lösung wird in einer globalen Variablen *maxCand* registriert
- Boundkriterien sind als globale Konstante *wmax* bzw. globale Variable *maxCand.v* realisiert
- Die aktuelle Konstellation *s* wird entlang der Methodenaufruflskette weitergereicht

Interaktionen im Programmcode

32

```
□ static void TryItem(int i, ItemSet s,  
    double av, double w) {  
    if (av - val[i] > maxCand.v) { s.elems[i] = false;  
        if (i == nofItems - 1)  
            maxCand = new Cand(s, av - val[i], w);  
        else TryItem(i + 1, s, av - val[i], w); }  
    if (w + weight[i] <= wmax) { s.elems[i] = true;  
        if (i == nofItems - 1) {  
            if (av > maxCand.v)  
                maxCand = new Cand(s, av, w + weight[i]); }  
        else TryItem(i + 1, s, av, w + weight[i]);  
    }  
}
```


Parallelisierungshindernisse

33

- Interferenzen bei der Benutzung von *s*
 - ▣ *Aliasing*: Weiterreichen via Referenz macht *s* zur gemeinsamen Variablen
- Interferenzen beim Testen und Setzen von *maxCand*
 - ▣ *Stale Value Problem*: Benutzung eines veralteten Wertes führt zu suboptimaler Beschneidung des Baumes
 - ▣ *Race Conditions*: Verzögertes Schreiben eines neuen Maximums kann ein noch besseres Resultate überschreiben
- Abnehmende Problemgrösse entlang der Rekursion

Lösungsansätze

34

- Aliasing: Cloning von *s* bei der Parameterübergabe vermeidet gemeinsame Benutzung
- Stale Value: Nur Effizienzbeeinträchtigung
- Race Conditions
 - ▣ Atomares Testen & Setzen (Vergleichen & Ersetzen)
 - *Interlocked.CompareExchange (ref x, newVal, expVal)*
 - ▣ *maxCand* als *Hint*. Verzögerte Maximumsbestimmung via Kandidatenmenge (lazy evaluation)
 - *Collection<Cand> c*
- Problemgrösse: Cutoff nach *seqN* Rekursionen

Atomares Vergleichen & Ersetzen

35

- **if** ($av - val[i] > \text{maxCand.v}$) { $s.\text{elems}[i] = \text{false};$
 if ($i == \text{nofItems} - 1$)
 UpdateMax(new Cand($s, av - val[i], w$)); ...
- **static void** UpdateMax (Cand newCand) {
 Cand cur = maxCand;
 while (newCand.v > cur.v) {
 Cand cur1 = Interlocked.CompareExchange(
 ref maxCand, newCand, cur);
 if (cur1 == cur) break;
 cur = cur1; }
}

Verzögerte Max Bestimmung (1)

36

```
□ static void TryItemPar (  
    int l;  
    ItemSet s,  
    Collection<Cand> c,  
    double v, double w) {  
    Parallel.Invoke(  
        () => { ... }, // inclusion of next item  
        () => { ... } // exclusion of next item  
    ); }  
}
```

Verzögerte Max Bestimmung (2)

37

- Einschluss des nächsten Items (Ausschluss analog)
 - ▣ `Collection<Cand> c1 = new Collection<Cand>()`
if (av - val[i] >= maxCand.v) { // catch all candidates
 ItemSet s1 = new ItemSet(s); s1.elems[i] = false;
 if (i < seqN)
 TryItemPar(i + 1, s1, c1, av - val[i], w);
 else TryItemSeq(i + 1, s1, c1, av - val[i], w); }
- Synchrone Vereinigung der Kandidaten aus beiden Teilen (Einschluss und Ausschluss)
 - ▣ `foreach (Cand cand in c1.Union(c2)) c.Add(cand);`

Verzögerte Max Bestimmung (3)



38

- Neuer Maximumskandidat gefunden

- ▣ if (i == nofltems - 1) {

- Cand newCand = **new** Cand(s, av - val[i], w);
 - maxCand = newCand; **c.Add(newCand);** }

- Lösungssuche in der Kandidatenmenge c

- ▣ **double** maxVal = c.Max(cand => cand.v);

- Cand [] maxCands = c.**Where**(
cand => cand.v == maxVal).ToArray();

- foreach** (Cand cand in maxCands) { ...

- // print solution

- }

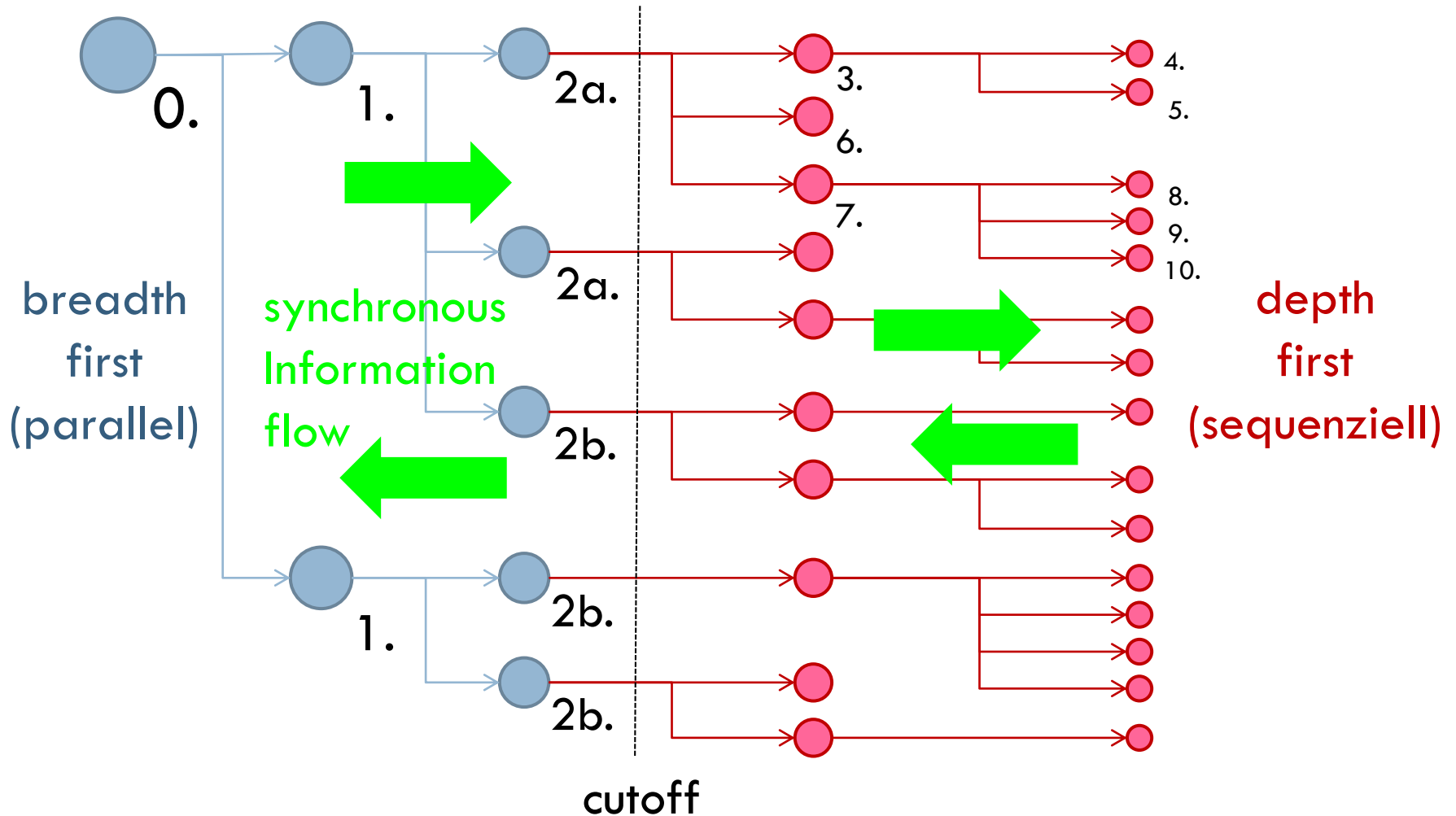


□ Laufzeitenvergleich

N	seq	par (atomic)	par (lazy)	speed up (best)
20	2855	90487	21272	0.13
30	184621	152175	126785	1.46
40	16251569	6292223	6058040	2.79

B&B Parallelisierungsmuster

40



Weitere Shared Memory Probleme

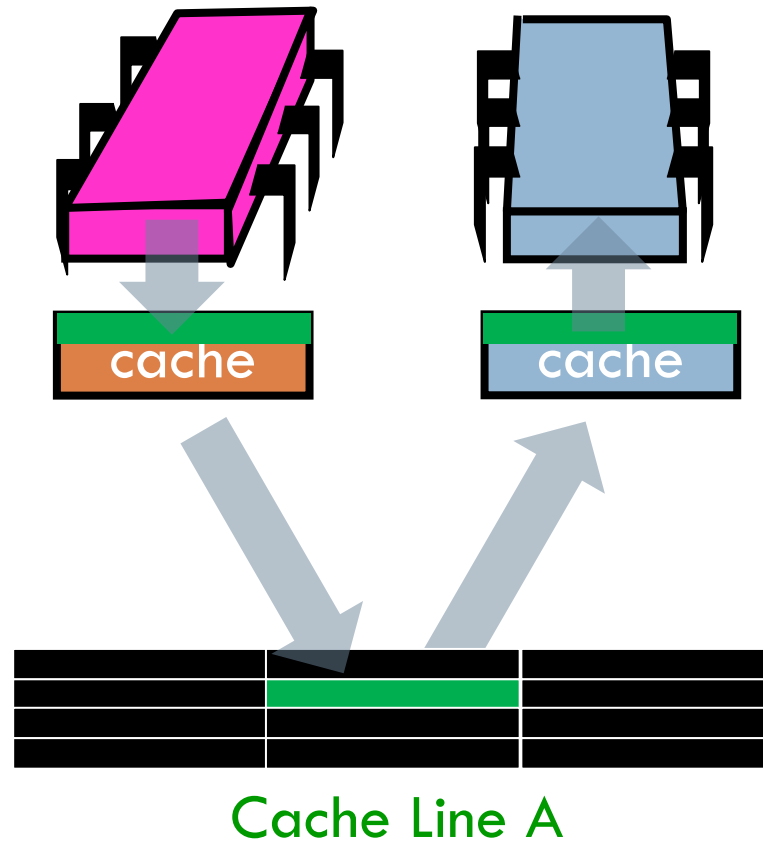
41

- Korrektheit & Effizienz
 - ▣ Cache Kohärenz (Write, HW)
 - ▣ False Sharing (Write, SW)
 - ▣ Speicherengpass (Bottleneck)

Cache Kohärenz

42

- Threads T_1, T_2
 - Ausgeführt von P_1 und P_2
 - T_1 schreibt nach A
 - T_2 liest von A
 - Cache Kohärenz Protokoll nötig



False Sharing

43

- Threads T_1, T_2
 - Ausgeführt von P_1 und P_2
 - T_1 schreibt nach D_1
 - T_2 schreibt nach D_2
 - Cache Line A ist sowohl an D_1 als auch an D_2 beteiligt, obwohl D_1 und D_2 disjunkt sind

