Counting, Sorting, and Distributed Coordination

16.06.2010

Proseminar Nebenläufige Programmierung
Referenten: Benno Kühnl, Alice Nitsche

Unser Anwendungsfall

- Pool von Objekten, die mittels put() und get() hinzugefügt und entnommen werden.

- Mögliche Sicherung bei Nebenläufigkeit: get() und put() als synchronized deklarieren

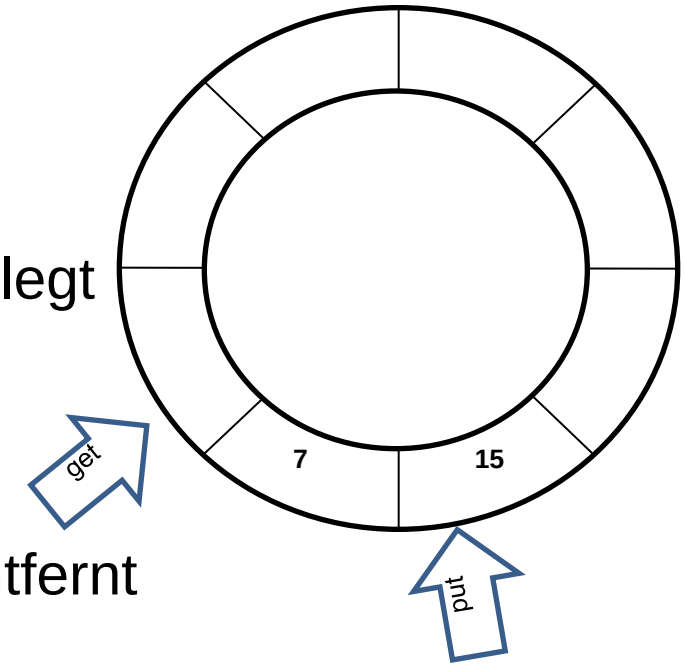
→ Flaschenhals, da sequenziell gearbeitet werden muss

→ Wie geht das besser?

Shared Counting – Ringpuffer

Besser:

- Zwei Zähler und zyklisches Array
- Bei put() wird ein Zähler inkrementiert
  → Array-Index, in das das Objekt abgelegt werden soll
- Bei get() wird der andere Zähler inkrementiert
  → Array-Index, von dem das Objekt entfernt werden soll



→ Ringpuffer ersetzt den Flaschenhals (Lock) durch zwei (Zähler)

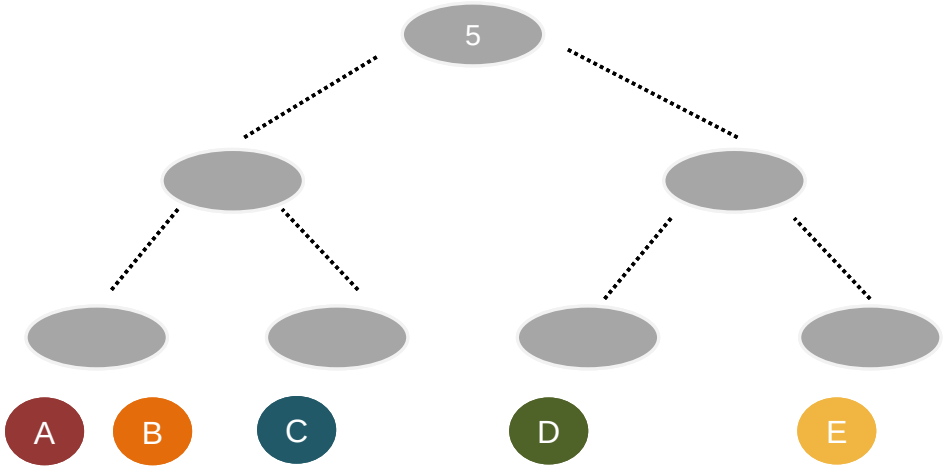
→ **Wie kann man parallelisiert auf diese zugreifen??**

Software Combining

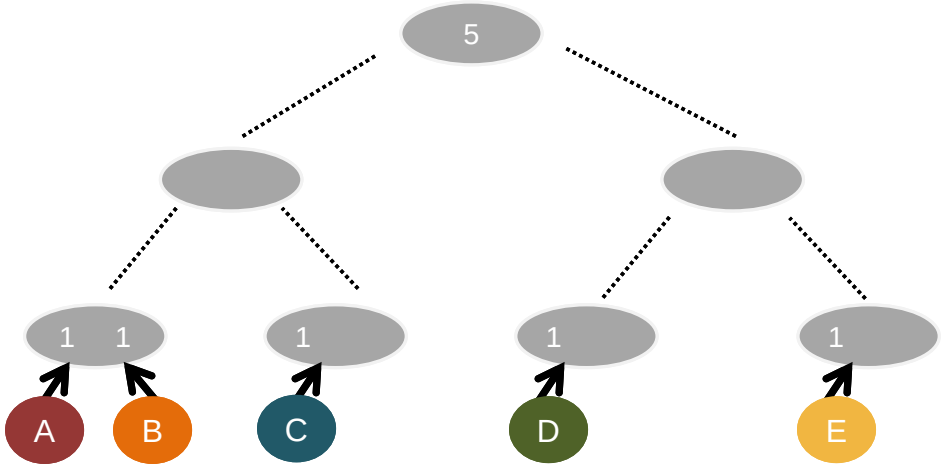
Das Konzept

- Binärer Baum mit Knoten
- Der Wert des Zählers wird in der Wurzel gespeichert
- Max. zwei Threads teilen sich ein Blatt
- Um einen Zähler zu inkrementieren, muss der Thread erst bis zur Wurzel „klettern“
- Treffen sich zwei Threads in einem Knoten, stoppt einer dort, und der andere klettert weiter, nimmt aber die „Nutzlast“ des wartenden mit (d.h. er inkrementiert den Zähler um den Wert von beiden Threads) = Combining

Software Combining – Beispiel

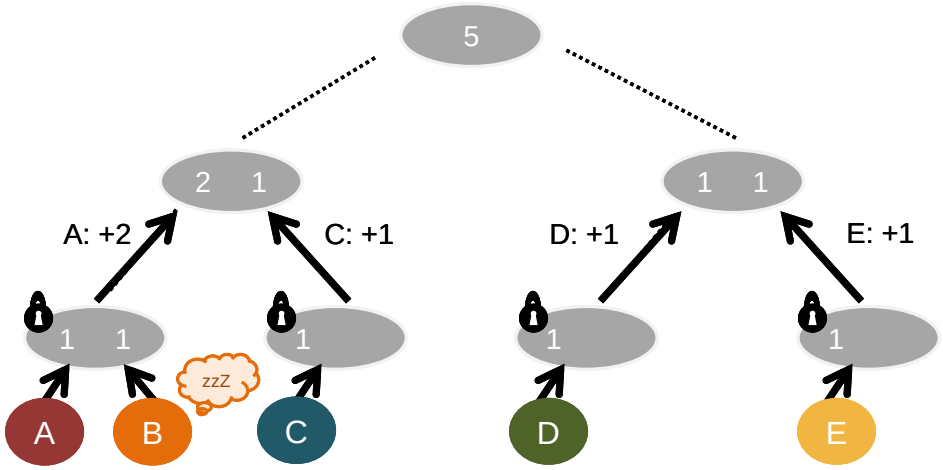


• 5 Threads, die gleichzeitig getAndIncrement() aufrufen wollen

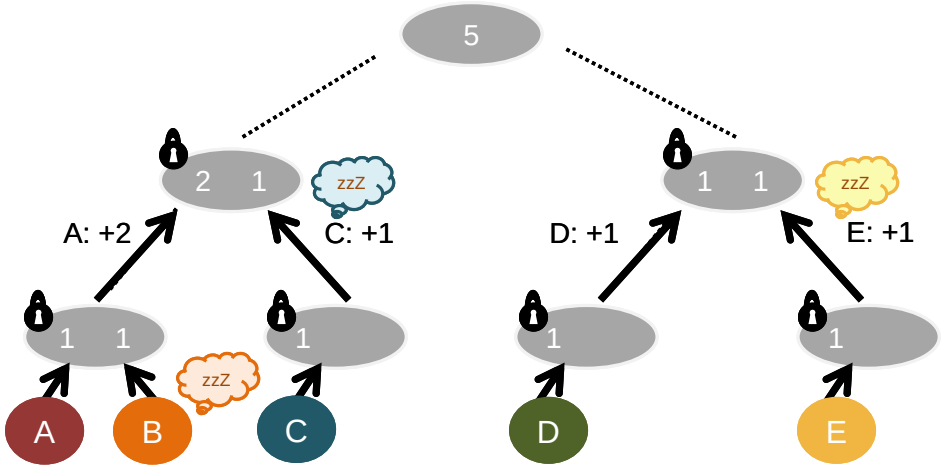


• Besucht ein Thread einen Knoten, trägt er seine „Nutzlast“ ein
•Thread A und Thread B kommen am selben Knoten an.

Software Combining – Beispiel

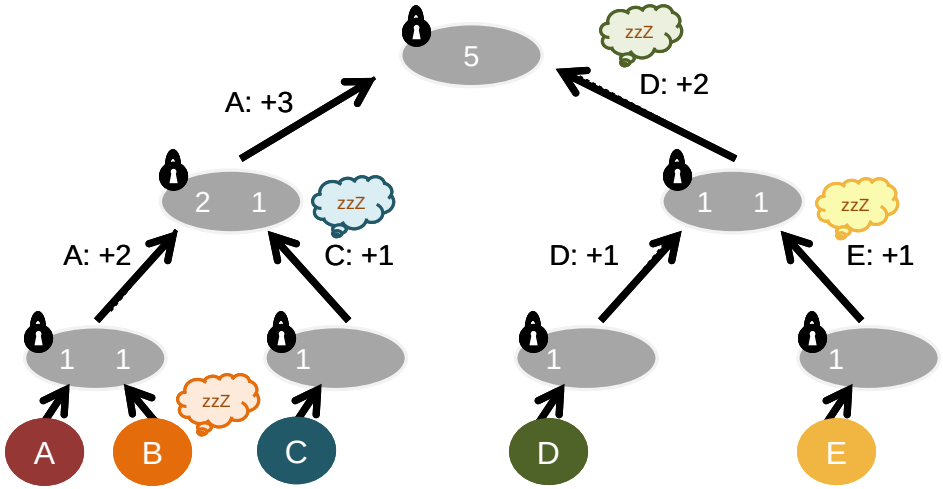


- Einer der am Knoten ankommenden Threads darf weiter nach oben wandern.
- Der jeweils andere muss warten, gibt dem ersten aber seine „Nutzlast“ mit
- Bei A und B ist das A.
- Der Knoten wird gesperrt, sodass kein weiterer Thread weiterkommen kann, solange der erste nicht wieder von oben herabkommt.

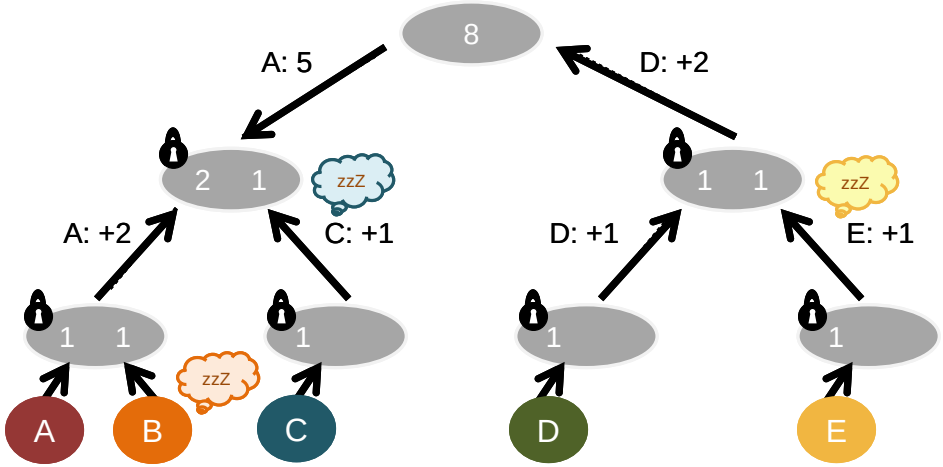


- Der selbe Vorgang erneut in der nächsten Ebene:
- Thread C und E werden schlafen geschickt, A und D werden ihren Wert mitnehmen.
- Die Knoten werden gesperrt.

Software Combining – Beispiel

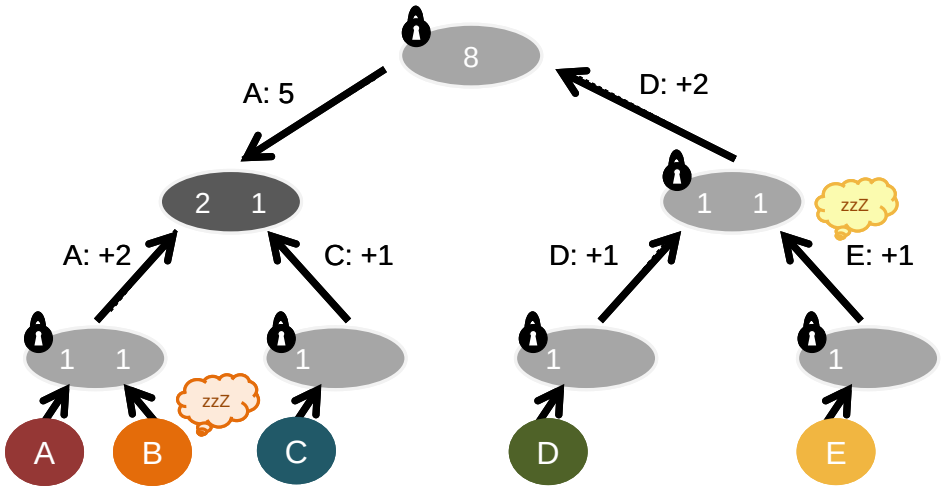


- Die Threads A und D kommen an der Wurzel an.
- A darf zuerst zugreifen und sperrt sie für alle anderen.
- A merkt sich den alten Wert der Wurzel (5).
- D wartet.

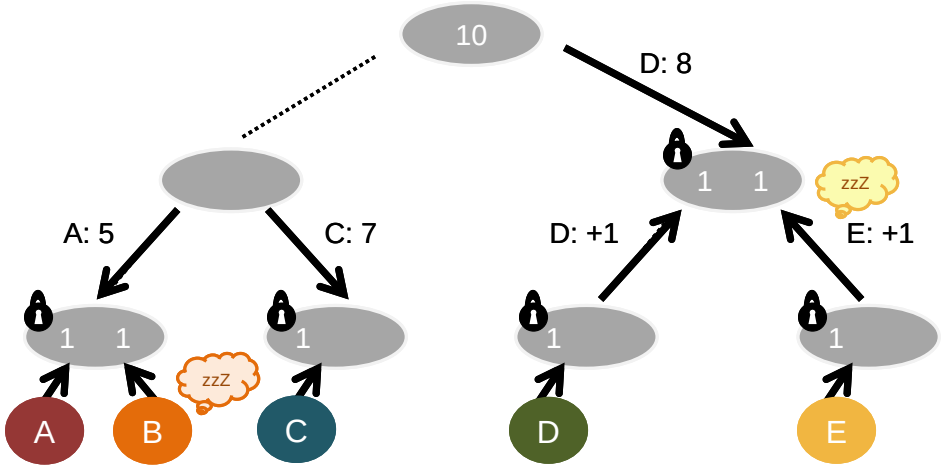


- Thread A hat den Wert der Wurzel erhöht und macht sich mit deren altem Wert auf den Rückweg.
- Da A nun die Wurzel entsperrt hat, kann Thread D zugreifen.

Software Combining – Beispiel

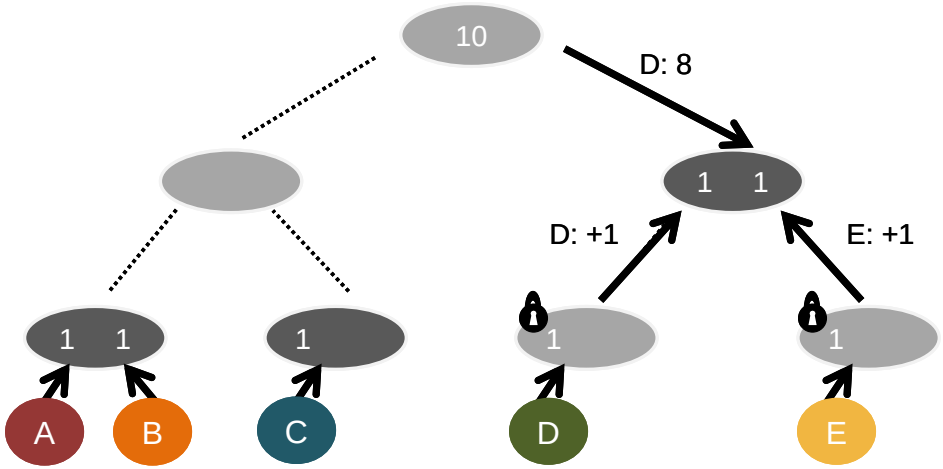


• Thread A besucht den ersten Knoten auf seinem Rückweg, und teilt dem wartenden Thread mit, dass er erfolgreich war.
• Thread D sperrt die Wurzel und merkt sich ihren alten Wert (8).

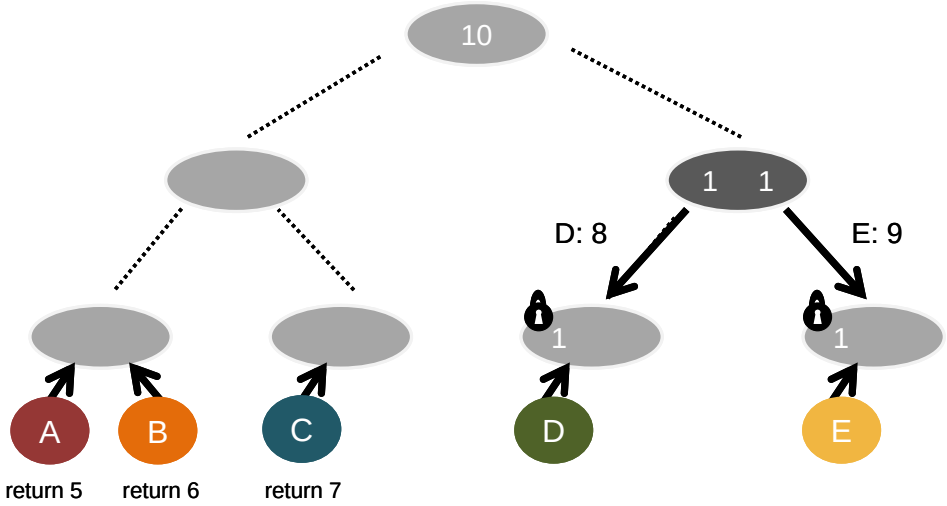


• Thread A erinnert sich an den Wert, den die Wurzel bei seinem Besuch zuerst hatte (5), zählt seinen (Zähl-)Wert (2) hinzu, und gibt ihn Thread C mit auf den Rückweg.
• Thread A selbst steigt auch abwärts, und nimmt dabei den (get-)Wert 5 mit.
•Thread D hat den Wert in der Wurzel aktualisiert und steigt mit dem alten Wert ab.

Software Combining – Beispiel

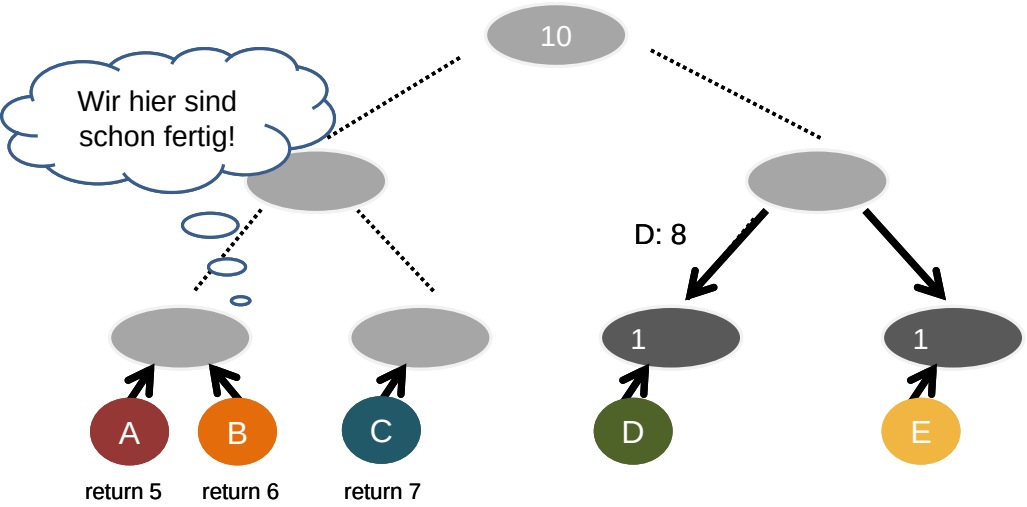


• Thread A ist bei seinem Blatt angekommen, teilt dem wartenden B mit, dass er erfolgreich war, und gibt B den gemerkten Wert + den eigenen (Zähl-)Wert (5+1=6).
• C ist bei seinem Blatt angekommen und hat dort mangels Partner nichts zu tun.
• Thread D ist an seinem ersten Rückwegsknoten angekommen und teilt dem wartenden E mit, dass er erfolgreich war.

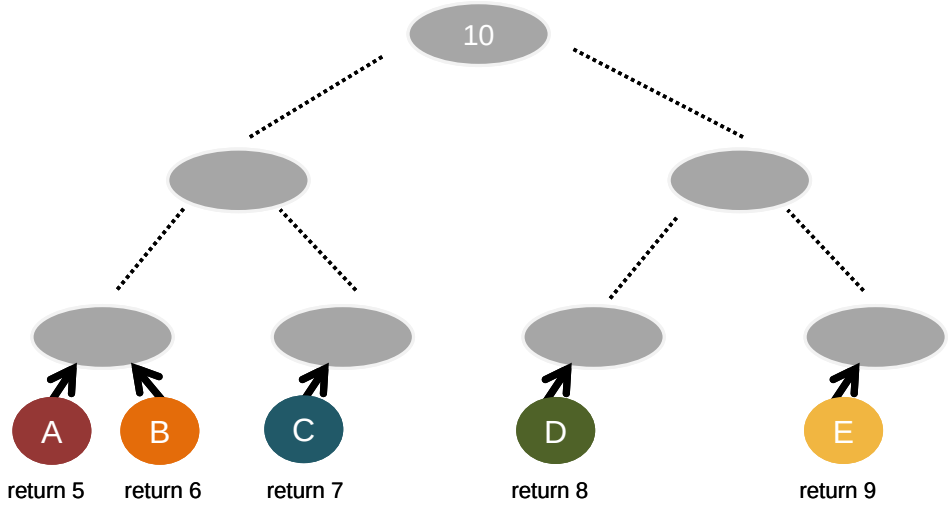


• Thread A, B und C geben ihre get-Werte zurück.
• Thread D gibt Thread E den gemerkten Wert + dem eigenen (Zähl-)Wert mit (8+1=9).

Software Combining – Beispiel



• Thread D und Thread E kommen in ihren Blättern an und haben dort mangels Partnern nichts zu tun.



• Die Threads D und E können nun auch ihre get-Werte zurückgeben.

Software Combining – Erweiterung

- Was passiert wenn die Threads nicht gleichzeitig in einem Knoten ankommen?

→ Lösung: Aufteilung des Algorithmus' in mehrere Phasen:

1. **Precombining-Phase**:
   Jeder Thread klettert den Baum nach oben, bis er zu einem Knoten kommt, an dem er *nicht* der erste ist (dort wartet er auf den ersten), oder bis zur Wurzel.
2. **Combining-Phase**:
   Jeder Thread läuft erneut den Weg ab. Trifft er dabei auf einen wartenden Thread, nimmt er dessen Wert mit.
   Jeder Knoten wird gesperrt; Threads, die nun noch eintreffen, müssen **warten** bis zum Ende der Aktion des aktiven Threads.
   Der Thread, der es bis zur Wurzel schafft, ändert deren Wert.
3. **Distributionsphase**:
   Der aktive Thread geht seinen Weg zurück und bringt den wartenden Threads ihre get-Werte mit (siehe vorhin).

Software Combining – Zusammenfassung

- Lange **Antwortzeit**: O(log n) statt O(1) bei einzelnem Aufruf
- Hoher **Durchsatz**: Bei mehreren Aufrufen von *getAndIncrement()* immer noch O(log n) anstatt O(n)

- Wann ist diese Datenstruktur besonders effizient?

→ Wenn die Konkurrenz *niedrig* ist, verpassen sich die Threads an den Knoten, können ihre Werte nicht kombinieren und müssen lange warten.

→ Der Durchsatz wird umso höher, je mehr Threads ihre Werte kombinieren können, also bei *hoher* Konkurrenz.
Eventuell kann auch an jedem Knoten gewartet werden (abhängig von der zu erwartenden Konkurrenz).

Software Combining ist also dann besonders effizient, wenn viele Threads parallel mit dem Zähler arbeiten wollen.

Es ist aber eher schlecht geeignet, wenn die Konkurrenz stark schwankt (fixe Wartezeit?)!

Counting Networks

Problem: Die Combining Trees müssen sehr gut koordiniert sein, damit sich die Threads nicht verpassen.

Frage: Gibt es irgendeine Möglichkeit, eine effiziente parallele Nutzung von Zählern zu ermöglichen, die auch bei schwankender Konkurrenz nicht schwächelt?

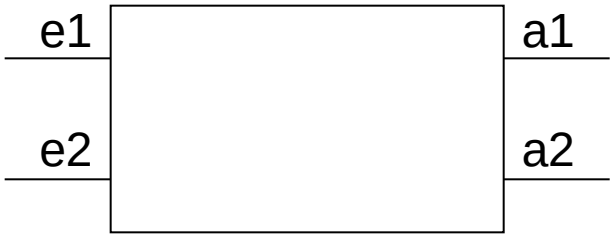
Lösung: Wir verwenden *mehrere* Zähler.
Wenn jeder der Zähler eine andere Zählweise hat als alle anderen, können wir uns die Synchronisation sparen.

→ Wie müssen die Counter zählen?

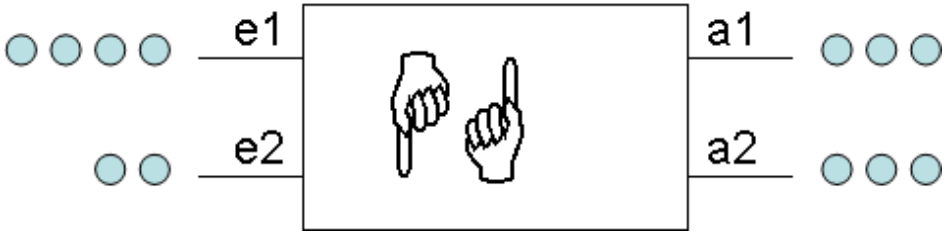
→ Wie verteilen wir die Threads auf die Counter, sodass das ganze „Quiescent Consistent“ wird (keine Werte überspringen, keine auslassen)?

Der Balancer

Ein Balancer ist ein elektronisches Bauteil mit zwei Eingängen und zwei Ausgängen.



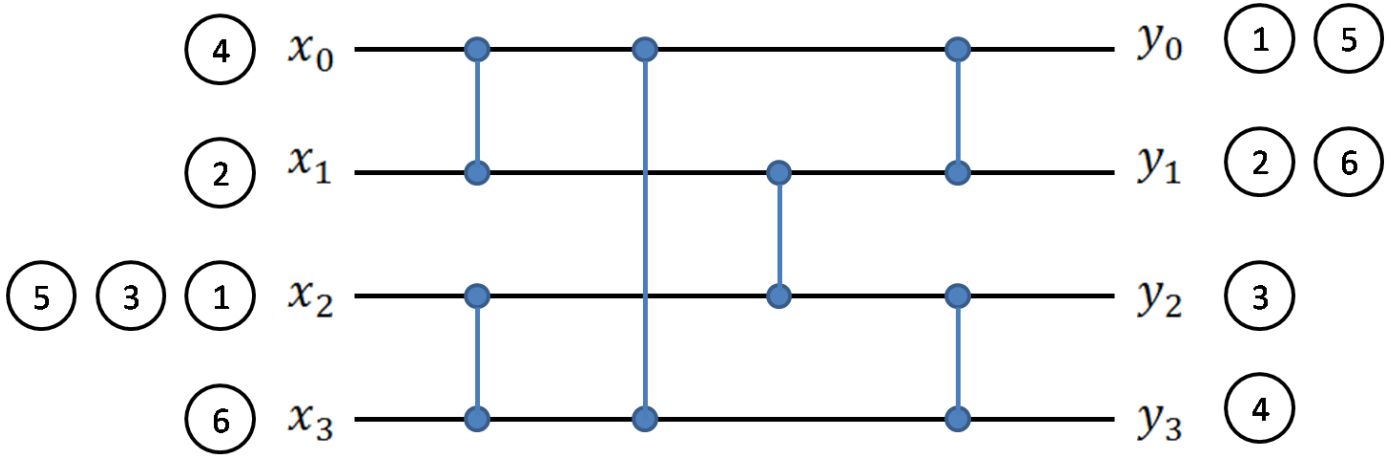
Kommt ein Signal auf einer Eingangsleitung, wird es auf eine Ausgangsleitung weitergeleitet, wobei abwechselnd die obere und die untere Ausgangsleitung verwendet wird (Balancer-Zustand „up“ und „down“).



Balancing Networks

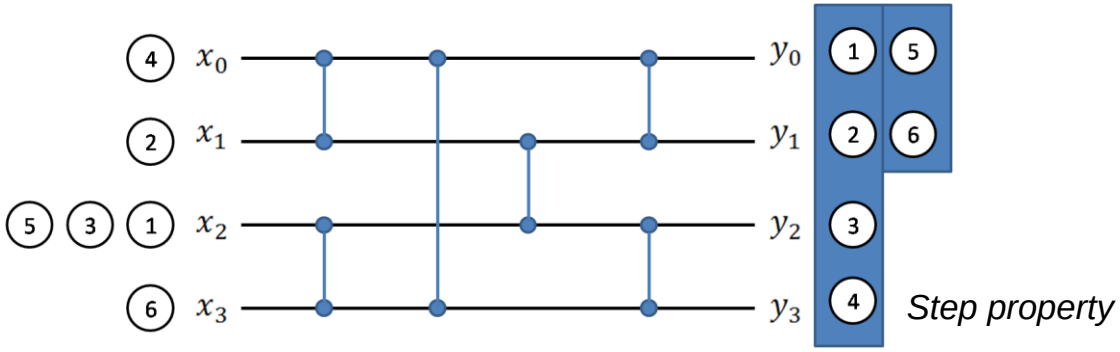
Das Konzept:

- Die Balancer werden zu einem Netzwerk zusammengeschaltet…
- …welches die *Step property* erfüllt:
  - egal in welcher Reihenfolge und Verteilung die Signale auf den Eingangsleitungen ankommen, auf den Ausgangsleitungen kommen sie balanciert an.
  - bei der Verteilung auf die Ausgangsleitungen wird von oben nach unten vorgegangen (→ in einer festgelegten Reihenfolge).



Vertikal: Balancer mit zwei Inputs und zwei Outputs

Counting Networks



- Balancing Networks, die die *Step property* erfüllen, werden *Counting Networks* genannt.
- An jeder Ausgangsleitung wird ein Zähler angebracht, der nach einem bestimmten Muster zählt.
- In unserem Beispiel hat jeder Zähler die Zählvorschrift

```
return (i * w + y)
```

`i` interner Zählerwert, bei jedem ankommenden Signal um 1 erhöht (begonnen bei 0)

`w` Anzahl der Zähler, in unserem Fall 4

`y` Nummer des Zählers, begonnen bei 1

Counting Networks – Umsetzung

- Die Balancer werden als Objekte im Speicher realisiert, die Ein- und Ausgangsleitungen sind Referenzen untereinander.
- Die Balancer bieten eine synchronisierte Funktion „traverse()“, die den aufrufenden Thread weiterweist.

```
public class Balancer {
    private boolean toggle =true;

    public synchronized int traverse() {
        try {
            if (this.toggle) {
                return 0;
            }
            else {
                return 1;
            }
        }
        finally {
            this.toggle = !this.toggle;
        }
    }
}
```

- Schwieriger umzusetzen als gedacht!
- Auch eine Implementierung ohne **synchronized** ist möglich, aber noch schwieriger.
- Eine Ausweitung auf $n=2^k$ ist möglich mit *rekursiven* oder *periodischen* Counting Networks.

Counting Networks – Zusammenfassung

- Im Gegensatz zum Combining Tree muss ein Counting Network nicht koordiniert sein, um eine gleichbleibende **Wartezeit** zu garantieren.
- Der **Durchsatz** steigt, je mehr Threads durch das Netz gehen (maximum, wenn die Zahl der Threads ca der der Balancer entspricht).

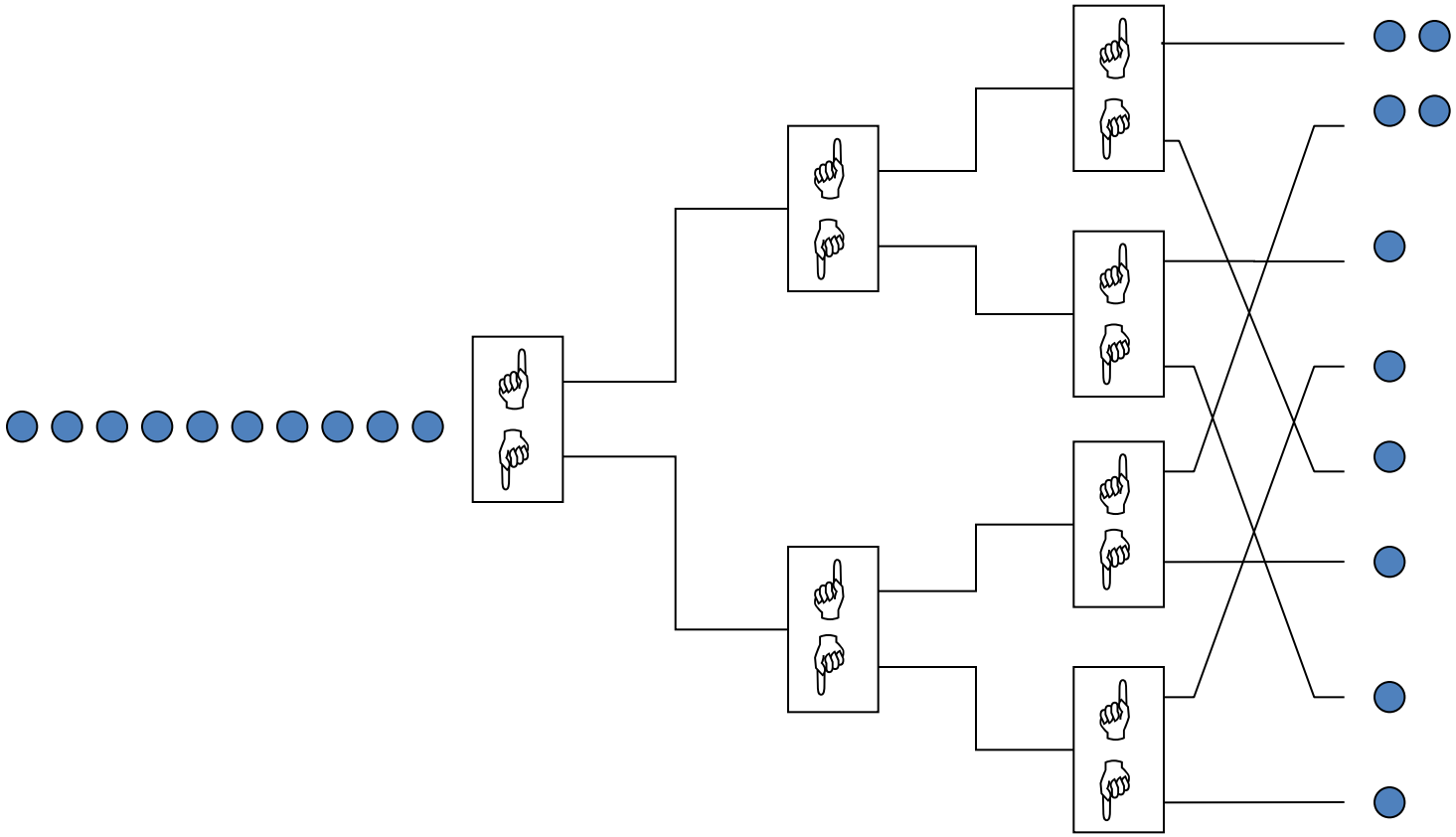
- Allerdings hängt die Wartezeit (pro Thread) von der Tiefe des Netzwerkes ab, diese ist $\Theta(\log^2 w)$ mit w = Netzbreite.

- …gibt es auch Counting Networks mit logarithmischer Tiefe?
  → ja, aber nein.
- Counting Networks mit logarithmischer Tiefe gibt es, aber diese sind aus verschiedenen Gründen unpraktikabel.

- Was tun wir also?

Diffracting Trees

- Wir verwenden Teile von Combining Trees und Teile von Counting Networks und bauen aus Balancern einen Baum, der die *Step property* erfüllt!



Diffracting Trees – Prisma

- Die Tiefe des Baums ist log w (im Gegensatz zum Counting Network mit $\log^2$ w).
- Problem: Alle Threads treten durch den ersten Balancer in den Baum ein, dieser wird ein Flaschenhals.
- Die Lösung:
  „*If an even number of tokens pass through a balancer, the outputs are evenly balanced on the top and bottom wires, but the balancer‘s state remains unchanged.*“
- Wir bauen ein *Prisma* vor jeden Balancer. Treffen hier zwei Threads aufeinander, geht der eine durch die obere Leitung des Balancers, und der andere durch die untere.
  Die `traverse()`-Funktion des Balancers (synchronized!) wird nur aufgerufen, wenn nach einer kurzen Wartezeit im Prisma kein zweiter Thread hinzugekommen ist.

Prisma

- Ein Prisma ist ein Array von **`Exchanger<Integer>`**-Objekten.
  Diese ermöglichen mit ihrer **`exchange(int)`**-Funktion, Werte zwischen zwei aufrufenden Threads auszutauschen.

Ablauf:

- Kommt ein Thread am Prisma an, wird ihm zufällig ein Exchanger-Objekt zugewiesen. In der exchange-Methode ist er geblockt, bis sie ein zweiter Thread aufruft, oder ein Timeout auftritt.
- Kommt ein zweiter Thread in die Methode, wird anhand der getauschten ThreadIDs entschieden, welcher Thread auf welcher Leitung weitergeht.
- Ein Thread bekommt also entweder eine Leitung (true/false oder 1/0) oder eine TimeoutException zurück.
- In letzterem Fall benutzt er die Traverse-Funktion des Balancers.

Diffracting Trees – Zusammenfassung

- Die Tiefe des Baums ist log w (im Gegensatz zum Counting Network mit $\log^2 w$).
- Die Effizienz hängt davon ab, wie wir die Kapazitäten der Prismen und die Wartezeiten in ihren exchange-Methoden einstellen.
  - → zu kleine Kapazität: Stau vor den Prismen
  - → zu große Kapazität: Stau vor den Balancern (Threads verfehlen sich in den Prismen)
  - → zu kurze Wartezeit: Stau vor den Balancern (Threads verfehlen sich in den Prismen)
  - → zu lange Wartezeit: Stau vor den Prismen, Zeitverschwendung

Sortieren

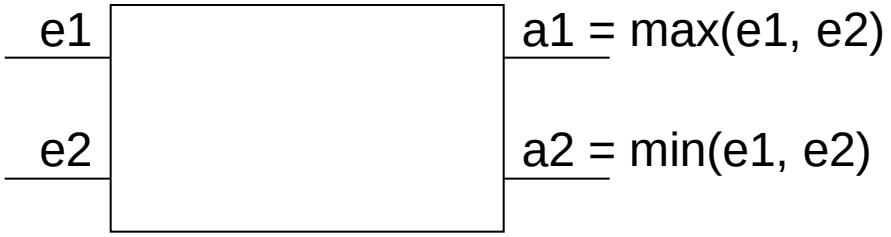
- Wie wir alle wissen ist Sortieren wahnsinnig wichtig, schwierig, häufig, etc.

- Bei der Wahl eines Algorithmus kommt es auf die Zahl der Elemente an, ihre Verteilung, und ob sie im Speicher sind oder auf einem externen Speichermedium.

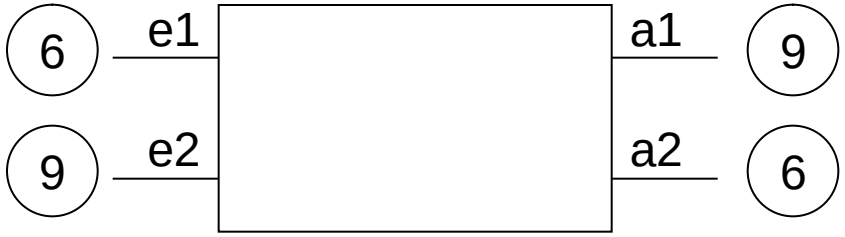
- Wir betrachten zwei Typen der Sortier-Parallelisierung:
  → **Sorting Networks**
  für kleine Mengen, die im Speicher sein können
  → **Sample Sorting**
  für größere Mengen, die auf ext. Speichermedien sind (z.B. Festplatte)

Comparators

- Ein Sorting Network ist ein Netzwerk aus Comparators.

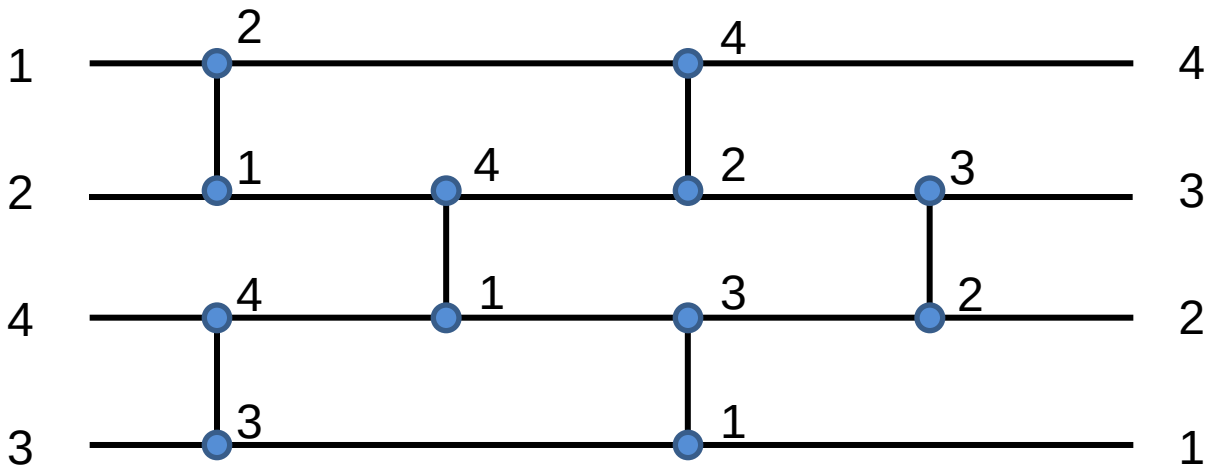


- Zwei eingehende Nummern werden so ausgegeben, dass die größere oben ist.
  → Comparators sind *synchronous*!



Sorting Networks

- Wie könnte ein solches Sorting Network aussehen?



- Balancing Networks und Compariso Networks sind isomorph: Wenn man in einem Counting Network die Balancers durch Comparators ersetzt, hat man ein Sorting Network!

Sorting Networks

- Ein Sorting Network kann eine Laufzeit von $O(s * \log^2 p) = O(\log^2 p)$ haben,
  mit p = Anzahl der Threads (und Prozessoren)
  2*p*s = Anzahl der zu sortierenden Elemente (p*s Zweierpotenz)
  p*s = Anzahl der Comparators pro Netz-Layer

- Jeder der p Threads emuliert dann die Arbeit von s Comparators.

- Vor jedem Layer des Netzes ist ein „Barrier“ eingebunden, der dafür sorgt, dass alle Comparators einer Schicht ihre Arbeit beenden haben, bevor neue Daten rein dürfen (Synchronisation).

- Ermöglicht das in-place-Sortieren eines Arrays.

- Doch was ist, wenn die Einträge nicht alle in den Speicher passen?

Sample Sorting

- Sinnvoll für größere Datenmengen (viel mehr Elemente als Threads), die nicht mehr in den Hauptspeicher passen.
- Zugriff auf ein Element ist teuer (Zeit!), deshalb ist es besonders effizient, wenn ein Thread Elemente sequentiell sortiert.

  → **Divide et impera** (Teile und herrsche)

- Wir teilen die zu sortierenden Elemente unter den sortierenden Threads auf, und jeder sortiert sequentiell:
- Vorgehen für p Threads und n Elemente:
  1. p-1 *Split-Schlüssel* werden (zufällig) ausgewählt und allen Threads bekannt gemacht.
  2. Jeder Thread liest n/p Elemente und sortiert sie in Häufchen, die durch die Split-Schlüssel getrennt sind („Vorsortierung“).
  3. Jeder Thread nimmt sich ein Häufchen und sortiert es sequentiell.

Sample Sorting

- Laufzeitkomplexität des Algorithmus`:
  1. O(1) für die erste Phase.
  2. (n/p) log p
     für die zweite Phase (Durchlauf (n/p), binäre Suche nach den Splittern log p).
     (n/p) dominiert wegen langsamerem Zugriff auf Platten etc.
  3. O(b log b)
     für die zweite Phase (abhängig vom Sortieralgorithmus im Thread), b ist die Häufchengröße.

- Die Kosten für Phase 3 sind am größten, da hier die meisten Plattenzugriffe stattfinden.
- Die Dauer von Phase 3 ist bestimmt durch das größte Häufchen.
  → Split-Schlüssel gut auswählen!

Distributed Coordination

- Einige Methoden, die wir gesehen haben, z.B. **Combining Trees**, **Sorting Networks** und **Sample Sorting**, besitzen „high parallelism“ und geringen Overhead.
  Diese Methoden müssen allerdings synchronisiert werden (Threads müssen sich zum Datenaustausch treffen oder an Barriers warten).

- Andere Methoden, wie **Counting Networks** und **Diffracting Trees**, warten Threads nie aufeinander (auch wenn es, z.B. in den Prismen zu Verbesserungen führen kann).
  Dafür müssen die Threads die verteilten Informationen untereinander austauschen!

- *Randomization* hilft uns oft weiter (z.B. in den Prismen, oder beim 1. Schritt von Sample Sort).

- Pipelining kann oft (z.B. bei den Netzwerken und Diffracting Trees) den Durchsatz erhöhen.

```
try {
        // Bla, bla, bla
}

finally {
```

Danke für die Aufmerksamkeit!!! ☺

```
}
```