Rechnerarchitektur SS 2014

Synchronisierung

Michael Engel

TU Dortmund, Fakultät für Informatik

Teilweise basierend auf Material von Gernot A. Fink und R. Yahyapour

17. Juni 2014

Literatur

- Culler/Singh/Gupta: Parallel Computer Architecture, 1999, Abschnitt 5.5, S. 334ff.
- Hennessy/Patterson: Computer Architecture, 3. Auflage, 2003, Abschnitt 6.7, S. 590ff.
- Huang: Advanced Computer Architecture, 1993, Abschnitt 11.3, S. 634ff.
- Mellor-Crummey/Scott: Algorithms for scalable synchronization on shared-memory multiprocessors, CACM 9(1), 1991.

Synchronisation

Ziele:

- Konfliktfreie Bearbeitung gemeinsamer Daten bzw.
  [Konflikt = Speicheroperationen unterschiedlicher Prozessoren greifen auf gleiche Speicherzelle zu und mindestens eine Op. schreibt]
- Zeitliche Synchronisation von Ereignissen in unabhängigen Kontrollflüssen paralleler Prozessoren

Synchronisations-Konstrukte:

- **Gegenseitiger Ausschluß**
  (Zugriff auf gemeinsam genutzte Daten[-strukturen])
- **Punkt-zu-Punkt Ereignisse**
  (Signalisierung einer best. Berechnungssituation von einem Prozessor an einen anderen, z.B. Ergebnis liegt vor)
- **Globale Ereignisse**
  (wie oben, nur erweitert auf viele/alle Prozessoren)

Problem: Durch Software allein nur bedingt zu erreichen!

Wie Synchronisation durch Hardware unterstützen?

Synchronisation II

- Klassische Lösung (Dijkstra 1965):
  - *Atomare* Lese-Operation und ...
  - ... *atomare* Schreiboperation ausreichend für Synchronisation

Synchronisation II

- Klassische Lösung (Dijkstra 1965):
  - *Atomare* Lese-Operation und ...
  - ... *atomare* Schreiboperation ausreichend für Synchronisation

Erfordert sequenzielle Speicherkonsistenz!

... sowie geeignetes Softwareprotokoll

Synchronisation II

- Klassische Lösung (Dijkstra 1965):
  - *Atomare* Lese-Operation und ...
  - ... *atomare* Schreiboperation ausreichend für Synchronisation

Erfordert sequenzielle Speicherkonsistenz!

... sowie geeignetes Softwareprotokoll

- Lösungen in der Praxis:
  - **Hardware**: Atomare *read-modify-write* Operationen
    (d.h. Lesen einer Speicherstelle, Verändern und Rückschreiben ohne zwischenzeitliche konkurrierende Zugriffe anderer Prozessoren möglich)
  - Weitere (komplexere) Synchronisationsmechanismen in **Software** a.d. Basis der Hardware-Primitive realisiert
    (z.B. Sperren/Freigeben eines kritischen Abschnitts)

Synchronisation III

Komponenten eines Synchronisationsereignisses:

- **Akquirierungs-Methode:** Methode, um Recht zur Synchronisation zu erlangen
  (z.B. Sperren eines krit. Abschnitts [*lock*])
- **Warte-Strategie:** Verfahren, mit dem Prozessor auf Verfügbarkeit der Synchronisation wartet
- **Freigabe-Methode:** Methode um andere Prozessoren über Synchronisationspunkt hinaus fortfahren zu lassen
  (z.B. Freigeben [*unlock*])

Beachte: Betrachten Implementierung von Synchronisationsoperationen auf bus-basierten cache-kohärenten Multiprozessoren (z.B. SMP)

Synchronisation VI

Warte-Strategie (unabhängig v. Typ der Synchronisation)

2 grundlegende Möglichkeiten:

- **Busy waiting**:
  - Prozessor testet in Schleife wiederholt (= "*spinning*"), ob best. Bedingung erfüllt (d.h. Speicherstelle = best. Wert).
  - Bedingung durch anderen Prozessor bei Freigabe erzeugt
- **Blockierendes Warten**:
  - Prozeß(!) wird angehalten und gibt Prozessor frei.
  - Wird wieder aktiviert, wenn erwartete Freigabe signalisiert.

Vor- und Nachteile:

- Hoher Overhead bei Blockieren (Prozessumschaltung nötig), aber Prozessor ist frei für andere Aufgabe
- Busy waiting effizienter (insbes. bei kurzem Warten), belegt aber Prozessor und verbraucht ggf. Cache-Bandbreite

Auch hybride Warte-Strategien möglich!

Gegenseitiger Ausschluss

Wichtigster/Bekanntester *use case* der Synchronisation: Gegenseitiger Ausschluss aus *kritischem Programmabschnitt* (engl. *mutual exclusion*)

- Realisiert mit Operationspaar (*lock* und *unlock*)
- Können mit einer Reihe von Algorithmen realisiert werden
  - Einfache Algorithmen: i.d.R. effizient bei geringer Anzahl konkurrierender Zugriffe, ineffizient bei großer Anzahl
  - Komplexe Algorithmen: Behandeln i.d.R. große Anzahl konkurrierender Zugriffe effizient mit höheren Kosten bei geringer Anzahl
- Heute keine *reine* Hardware-Realisierung von Locks mehr (z.B. mit [kleiner Anzahl von] *lock lines* auf bus-basiertem MP-System)
- Realisierung als Kombination von Hardware-Primitiven und Software (d.h. Algorithmus)

Gegenseitiger Ausschluss II

Naiver Lösungsversuch nur durch Software:

```
        ADDI  R2,  R0, 1      #  Konstante 1 erzeugen
lock:   LD    R1,  location   #  Speicherinhalt lesen
        BNEZ  R1,  lock       #  Wiederholen, falls != 0
        SD    R2,  location   #  Sperre durch Speichern
        ...                   #  von 1 markieren

unlock: ADDI  R2, R0, R0      # Konstante 0 erzeugen
        SD    R2, location    # 0 markiert Sperre als frei
        ...
```

Wo ist das Problem?

Gegenseitiger Ausschluss III

Beispielablauf des naiven Algorithmus:

Zeit	P_1	P_2
0	/* Annahme: location = 0 */	
1	`LD R1, location`	
2	`BNEZ R1, lock`	
3		`LD R1, location`
4		`BNEZ R1, lock`
5		`SD R2, location`
6	`SD R2, location`	

Gegenseitiger Ausschluss III

Beispielablauf des naiven Algorithmus:

Zeit	P_1	P_2
0	/* Annahme: location = 0 */	
1	`LD R1, location`	
2	`BNEZ R1, lock`	
3		`LD R1, location`
4		`BNEZ R1, lock`
	Beide Prozessoren glauben, Lock erhalten zu haben!	
5		`SD R2, location`
6	`SD R2, location`	

Gegenseitiger Ausschluss III

Beispielablauf des naiven Algorithmus:

Zeit	P_1	P_2
0	/* Annahme: location = 0 */	
1	`LD R1, location`	
2	`BNEZ R1, lock`	
3		`LD R1, location`
4		`BNEZ R1, lock`
	Beide Prozessoren glauben, Lock erhalten zu haben!	
5		`SD R2, location`
6	`SD R2, location`	

Problem: *lock* soll Atomizität für Zugriff auf kritischen Bereich sicherstellen, ist aber selbst nicht atomar!

Gegenseitiger Ausschluss III

Beispielablauf des naiven Algorithmus:

Zeit	P_1	P_2
0	/* Annahme: location = 0 */	
1	`LD R1, location`	
2	`BNEZ R1, lock`	
3		`LD R1, location`
4		`BNEZ R1, lock`
	Beide Prozessoren glauben, Lock erhalten zu haben!	
5		`SD R2, location`
6	`SD R2, location`	

Problem: *lock* soll Atomizität für Zugriff auf kritischen Bereich sicherstellen, ist aber selbst nicht atomar!

↯ Ex. keine Möglichkeit die zeitliche Verschränkung der Instruktionen verschiedener Prozessoren zu verhindern!

Gegenseitiger Ausschluss VI

Problem: Software(!)-*lock* soll Atomizität für Zugriff auf kritischen Bereich sicherstellen, ist aber selbst nicht atomar!

Lösung: Atomare *read-modify-write* Operation im Befehlssatz des jeweiligen Prozessors

- Liest Speicherinhalt in Register und ...
- speichert anderen Wert (ggf. Funktion des gelesenen Werts) ohne zwischenzeitlichen Speicherzugriff!

Realisierungsmöglichkeiten:

- *test&set*
- *swap*
- *fetch&op* (z.B. *fetch&add*, *fetch&increment*)
- *compare-and-swap*

Gegenseitiger Ausschluss V

Lösung mit test&set Befehl
(d.h. hardware-unterstützte Software-Implementierung)

Semantik von `T&S` (d.h. *test&set*):

- Speicherinhalt wird in Register gelesen und ...
- Best. Wert (hier: 1) in Speicher geschrieben (atomar!)

```
lock: T&S R1, location   # Speicherinhalt lesen und
                         # Zelleninhalt auf 1 setzen
      BNEZ R1, lock      # Vergleiche alten Wert mit 0,
                         # != 0 => lock bereits aktiv
      ...
```

Gegenseitiger Ausschluss V

Lösung mit test&set Befehl
(d.h. hardware-unterstützte Software-Implementierung)

Semantik von `T&S` (d.h. *test&set*):

- Speicherinhalt wird in Register gelesen und ...
- Best. Wert (hier: 1) in Speicher geschrieben (atomar!)

```
lock: T&S R1, location    # Speicherinhalt lesen und
                          # Zelleninhalt auf 1 setzen
      BNEZ R1, lock       # Vergleiche alten Wert mit 0,
                          # != 0 => lock bereits aktiv
      ...
```

$\Rightarrow$ Lock wird durch `T&S` implizit gesperrt (da Speicher $= 1$ gesetzt).

Gegenseitiger Ausschluss VI

Benchmark der einfachen Lock-Realisierung (Culler *et al.* 1999):

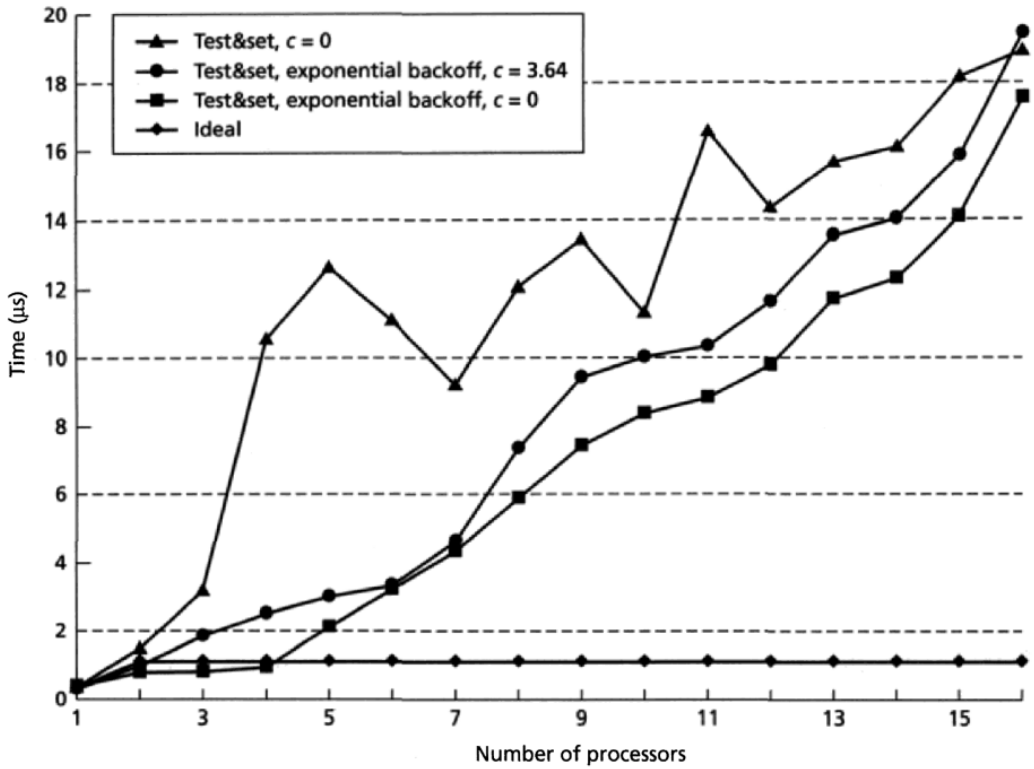
```
lock(L);
critical-section(c);
unlock(L);
```

- Dauer des kritischen Abschnitts c
- Benchmark-Characteristiken:
  - Laufzeitmessung der Lock-Akquisition für steigende Anzahl von Prozessoren
  - Gesamtzahl der Lock-Akquisitionen gleich (d.h. feste Anzahl von Aufgaben wird aus Warteschlange entnommen)
  - $$\text{Lock-Transfer-Zeit} = \frac{\text{Gesamtlaufzeit} - \text{Gesamtzeit im krit. Abschnitt}}{\text{Anzahl d. Lock-Akquisitionen}}$$
  - Ausführungsplattform: SGI Challenge, d.h. SMP-System

Gegenseitiger Ausschluss VII

Performanz der einfachen Lock-Realisierung (mit `T&S`):

`Code:`
lock(L);
critical-
section(c);
unlock(L);



Wodurch entsteht Performanzeinbruch / Abweichung vom Ideal?

Gegenseitiger Ausschluss VIII

Verbesserungsmöglichkeiten für den einfachen Lock-Algorithmus

- Häufigkeit der (erfolglosen) Lock-Akquisitionsversuche reduzieren
  - erneute Lock-Akquisition nach Fehlversuch verzögern
  - Kompromiss bei der Verzögerungszeit: exponentiell ansteigende Wartezeit $k \times c^i$

→ *test&set* Lock mit exponentiellem Backoff

- Schreibzugriffe bei Akquisitionsversuchen vermeiden ...
  - ... durch Ausnutzung der Cache-Kohärenz
  - Prüfung der Lock-Verfügbarkeit auf gecachter Kopie, d.h. kein direktes Schreiben mit `T&S`
  - wird invalidiert, sobald Lock frei
  - dann mit `T&S` Akquisition(sversuch)

→ *test-and-test&set* Lock

Gegenseitiger Ausschluss IX

test&set Lock mit exponentiellem Backoff

```
          ADDI  R3, R0, 1     # Konstante 1 erzeugen
lock:     T&S   R1, location  # Sperrversuch d. Locks
          BEQZ  R1, continue  # erfolgreich?
          SLL   R3, R3, 1     # Wartezeit skalieren
          Pause R3            # warten
          J     lock          # zum neuen Versuch
continue: ...
```

Weitere Verbesserungsmöglichkeit: randomisierte Wartezeiten (in best. Intervall)

Gegenseitiger Ausschluss X

test-and-test&set Lock

```
lock: LD   R1, location    # Lock auslesen
      BNEZ R1, lock        # != 0 => lock bereits aktiv
      T&S R1, location     # Speicherinhalt lesen und
                           # Zelleninhalt auf 1 setzen
      BNEZ R1, lock        # Vergleiche alten Wert mit 0,
                           # != 0 => lock bereits aktiv
      ...
```

- `LD` liest Lock-Status lediglich
- Wiederholtes Lesen auf gecachter Kopie
- Lockfreigabe führt zur Invalidierung der gecacheten Kopie

$\Rightarrow$ Andere (viele/alle?) Prozessoren konkurrieren um Akquisition (mit `T&S`)

Gegenseitiger Ausschluss

Step	P0	P1	P2	Coherence state of lock at end of step	Bus/directory activity
1	Has lock	Begins spin, testing if lock = 0	Begins spin, testing if lock = 0	Shared	Cache misses for P1 and P2 satisfied in either order. Lock state becomes shared.
2	Set lock to 0	(Invalidate received)	(Invalidate received)	Exclusive (P0)	Write invalidate of lock variable from P0.
3		Cache miss	Cache miss	Shared	Bus/directory services P2 cache miss; write-back from P0; state shared.
4		(Waits while bus/ directory busy)	Lock = 0 test succeeds	Shared	Cache miss for P2 satisfied
5		Lock = 0	Executes swap, gets cache miss	Shared	Cache miss for P1 satisfied
6		Executes swap, gets cache miss	Completes swap: returns 0 and sets lock = 1	Exclusive (P2)	Bus/directory services P2 cache miss; generates invalidate; lock is exclusive.
7		Swap completes and returns 1, and sets lock = 1	Enter critical section	Exclusive (P1)	Bus/directory services P1 cache miss; sends invalidate and generates write-back from P2.
8		Spins, testing if lock = 0			None

Cache-Aktivität und Bus-Kommunikation beim Lock-Wechsel mit drei Prozessoren

Gegenseitiger Ausschluss XI

Performanz-Anforderungen an Locks:

- Geringe *Latenz* für Lock-Akquisition und -Freigabe
- Geringer *Kommunikationsoverhead*, d.h. geringe Anzahl Bus-Transaktionen zur Akquisition / Freigabe
- *Skalierbarkeit*, d.h. Aufwand für Lock-Akquisition und -Freigabe möglichst unabhängig von der Anzahl konkurrierender Prozessoren P
- Geringer *Speicheraufwand* zur Verwaltung des Lock-Status
- *Fairness*: keiner der um die Akquisition des Locks konkurrierenden Prozessoren wird benachteiligt
  (ideal: FCFS, Extremfall: *starvation*)

Gegenseitiger Ausschluss XI

Analyse der Performanz-Anforderungen
(Basis: Realisierung mit *test&set*):

- *Latenz*:
- *Kommunikationsoverhead*:
- *Skalierbarkeit*:
- *Speicheraufwand*:
- *Fairness*:

Gegenseitiger Ausschluss XI

Analyse der Performanz-Anforderungen
(Basis: Realisierung mit *test&set*):

- *Latenz*: gering, besonders wenn Lock mehrfach vom selben Prozessor acquiriert, unabhängig von P
  kaum höher für andere Realisierungen
- *Kommunikationsoverhead*:
- *Skalierbarkeit*:
- *Speicheraufwand*:
- *Fairness*:

Gegenseitiger Ausschluss XI

Analyse der Performanz-Anforderungen
(Basis: Realisierung mit *test&set*):

- *Latenz*: gering, besonders wenn Lock mehrfach vom selben Prozessor acquiriert, unabhängig von P
  kaum höher für andere Realisierungen
- *Kommunikationsoverhead*: hoch, wenn viele Prozessoren konkurrieren: Invalidierungen bei jedem `T&S`, alle Prozessoren versuchen Akquisition erneut
  geringer mit Backoff, wesentlich geringer mit "Vortest"
- *Skalierbarkeit*:

- *Speicheraufwand*:
- *Fairness*:

Gegenseitiger Ausschluss XI

Analyse der Performanz-Anforderungen
(Basis: Realisierung mit *test&set*):

- *Latenz*: gering, besonders wenn Lock mehrfach vom selben Prozessor acquiriert, unabhängig von P
  kaum höher für andere Realisierungen
- *Kommunikationsoverhead*: hoch, wenn viele Prozessoren konkurrieren: Invalidierungen bei jedem `T&S`, alle Prozessoren versuchen Akquisition erneut
  geringer mit Backoff, wesentlich geringer mit "Vortest"
- *Skalierbarkeit*: schlecht, auch bei verbesserten Varianten (selbst bei *test-test&set* versuchen alle Prozessoren nach Freigabe Lock-Akquisition erneut!)
- *Speicheraufwand*:
- *Fairness*:

Gegenseitiger Ausschluss XI

Analyse der Performanz-Anforderungen
(Basis: Realisierung mit *test&set*):

- *Latenz*: gering, besonders wenn Lock mehrfach vom selben Prozessor acquiriert, unabhängig von P
  kaum höher für andere Realisierungen
- *Kommunikationsoverhead*: hoch, wenn viele Prozessoren konkurrieren: Invalidierungen bei jedem `T&S`, alle Prozessoren versuchen Akquisition erneut
  geringer mit Backoff, wesentlich geringer mit "Vortest"
- *Skalierbarkeit*: schlecht, auch bei verbesserten Varianten (selbst bei *test-test&set* versuchen alle Prozessoren nach Freigabe Lock-Akquisition erneut!)
- *Speicheraufwand*: gering, nur eine Statusvariable
- *Fairness*:

Gegenseitiger Ausschluss XI

Analyse der Performanz-Anforderungen
(Basis: Realisierung mit *test&set*):

- *Latenz*: gering, besonders wenn Lock mehrfach vom selben Prozessor acquiriert, unabhängig von P
  kaum höher für andere Realisierungen
- *Kommunikationsoverhead*: hoch, wenn viele Prozessoren konkurrieren: Invalidierungen bei jedem `T&S`, alle Prozessoren versuchen Akquisition erneut
  geringer mit Backoff, wesentlich geringer mit "Vortest"
- *Skalierbarkeit*: schlecht, auch bei verbesserten Varianten (selbst bei *test-test&set* versuchen alle Prozessoren nach Freigabe Lock-Akquisition erneut!)
- *Speicheraufwand*: gering, nur eine Statusvariable
- *Fairness*: schlecht, *starvation* nicht ausgeschlossen!

Verbesserte Hardware-Primitive: LL-SC

Ziele:

- Verfügbarkeitstest nur mit Lesezugriff
- Fehlgeschlagene *read-modify-write* generieren *keine* Invalidierungen
- Vorteilhaft, wenn einzelnes Primitiv eine Reihe von *read-modify-write* Operationen erzeugen kann

Realisierung: *load-linked* und *store-conditional*

Load-linked (LL) lädt Daten aus Speicher und "markiert" Stelle (setzt Hardware-lock-Flag und lock-Adress-Register im Prozessor)

Store-conditional (SC) schreibt zurück,

- wenn kein zwischenzeitlicher Zugriff (Bustransaktionen mit lock-Adress-Register vergleichen, lock-Flag löschen bei Zugriff);
- Erfolg durch Registerinhalt (=1) signalisiert
- Fehlschlag generiert keine Invalidierungen (LL-SC muss erneut versucht werden)

Einfaches Lock mit LL-SC

```
lock:   LL   R1, location  # lock lesen -> R1
        BNEZ R1, lock      # gesperrt? -> Wiederholen
        ADDI R2, R0, 1     # Konstante 1 erzeugen
        SC   R2, location  # lock zu sperren versuchen
        BEQZ R2, lock      # Bei Fehlschlag wiederholen
        ...
```

`unlock` wie gehabt!

- `LL-SC` kann weitere *read-modify-write* Op. erzeugen ...
  - abhängig vom Code zwischen `LL` und `SC`
  - Implementierung klein halten $\Rightarrow$ Erfolg von `SC` wahrscheinlicher
  - Keine Befehle (*stores*), die rückgängig gemacht werden müssen
- `SC` schlägt (ohne Schreiben) fehl, wenn:
  - Zwischenzeitlicher Schreibzugriff wird detektiert
  - `SC` eines anderen Prozessors erhält zuerst Zugang zum Bus

Achtung:
- `LL` ist kein `lock` und `SC` ist kein `unlock`!
- Erfolgreiches `LL-SC`-Paar garantiert nicht, dass Befehle dazwischen atomar ausgeführt wurden!

Verbesserte Hardware-Primitive: LL-SC II

Vorteil der Grundoperationen (*load-linked / store conditional*):
Andere Synchronisationsprimitive können realisiert werden!

Beispiel: Beispiel: Atomares *fetch-and-increment*:

```
try: LL      R2,  location    # Speicherinhalt lesen
     DADDUI  R3,  R2, 0x01    # Inkrementieren
     SC      R3,  location    # bedingtes Speichern ...
     BEQZ    R3,  try         # ...erfolgreich?
                              # Nein -> Wiederholen!
```

Realisiert atomare Inkrementierung der der Speicherstelle location, z.B. zur Implementierung von Semaphoren

Was kann man damit anstellen?

Verbesserte Hardware-Primitive: LL-SC III

Beispiel für Anwendung von fetch&increment:
Betrachten Schleife mit (daten-)unabhängigen Schleifenrümpfen:

```
for (i=1; i<100; i++)
        Schleifenkörper(i);
```

Kann mit *fetch&increment* bzw. *fetch&add* in parallelisierbare Variante umgeschrieben werden:

```
        j = Fetch&Add(i,1);
L1:     if (j > 100)
            goto L2
        Schleifenkörper(j);
        j = Fetch&Add(i,1);
        goto L1;
L2:     ...
```

Effizientere Algorithmen für Software-Locks

- Probleme mit dem einfachen `LL-SC` Lock:
  - Keine Invalidierung bei Fehlschlag, aber *read misses* für alle wartenden Prozessoren sowohl nach Freigabe als auch nach erfolgreicher Akquisition
  - Algorithmus ist nicht fair
- Anforderungen an bessere Algorithmen für bus-basierte MP-Systeme (a.d. Basis von *read-modify-write* Operationen bzw. `LL-SC`):
  1. Nur ein Prozessor versucht Lock nach Freigabe zu ackquirieren
  2. Nur ein Prozessor erhält *read miss* nach Freigabe
  3. Fairness, d.h. FCFS-Semantik
- Algorithmen:
  - *Ticket-Lock* erfüllt 1. Forderung, ist fair
  - *Array-basiertes Lock* erfüllt alle Forderungen

Effizientere Software-Locks: Ticket Lock

Prinzip wie Ziehen einer Bearbeitungsnummer (z.B. bei Behörde):

- Ex. zwei Zähler pro Lock (`next_ticket`, `now_serving`)
- *Acquire*: `fetch&inc next_ticket`; warten bis Ergebnis gleich `now_serving`
- *Release*: `now_serving` inkrementieren

Wieso nicht `fetch&inc`?

Charakteristika:

- Nur ein *read-modify-write* von nur einem Prozessor bei Lock-Akquisition
- FCFS-Semantik
- Geringe Latenz bei kleiner Anzahl konkurrierender Prozessoren
- minimal erhöhter Speicheraufwand (2 Statusvariablen)
- Dennoch: $O(P)$ *read misses* bei Lock-Freigabe, da alle Prozessoren dieselbe Statusvariable beobachten
- Backoff zur Reduktion d. Bustransaktionen beim Warten sinnvoll

Effizientere Software-Locks: Ticket Lock II

```
struct lock { int next_ticket = 0; int now_serving = 0; };

void acquire_lock(lock* L) {
  // returns old value arithmetic overflow is harmless
  int my_ticket = fetch_and_inc(L->next_ticket);
  while (1) {
    // consume this many units of time
    // on most machines, subtraction works correctly despite overflow
    pause(my_ticket - L->now_serving);
    if (L->now_serving == my_ticket)
       return;
  }
}
void release_lock(lock* L) {
  L->now_serving++;
}
```

Mellor-Crummey/Scott: Algorithms for scalable synchronization on shared-memory multiprocessors, CACM 9(1), 1991.

Effizientere Software-Locks: Array-basiertes Lock

Grundprinzip: Alle Prozessoren beobachten unterschiedliche Statusvariablen in einem Feld der Größe P

- *Acquire*: `fetch&inc` liefert Adresse der Statusvariable, die beobachtet werden muss $\Rightarrow$ dann "normaler" Acquire-Versuch

- *Release*: Setzt Freigabesignal an nächster Stelle im Array $\Rightarrow$ entsprechender wartender Prozessor wird "aufgeweckt"

Charakteristika:

- Aufwand $O(1)$ bei Lock-Akquisition mit kohärentem Cache
- FCFS-Semantik (= FIFO)
- Speicheraufwand $O(P)$ pro Lock!
- Gute Performanz ... auf bus-basierten Maschinen

Effizientere Software-Locks: Array-basiertes Lock

Grundprinzip: Alle Prozessoren beobachten unterschiedliche Statusvariablen in einem Feld der Größe P

- *Acquire*: `fetch&inc` liefert Adresse der Statusvariable, die beobachtet werden muss $\Rightarrow$ dann "normaler" Acquire-Versuch

  Array-Elemente müssen in unterschiedlichen Cache-Lines liegen!

- *Release*: Setzt Freigabesignal an nächster Stelle im Array $\Rightarrow$ entsprechender wartender Prozessor wird "aufgeweckt"

Charakteristika:

- Aufwand $O(1)$ bei Lock-Akquisition mit kohärentem Cache
- FCFS-Semantik (= FIFO)
- Speicheraufwand $O(P)$ pro Lock!
- Gute Performanz ... auf bus-basierten Maschinen

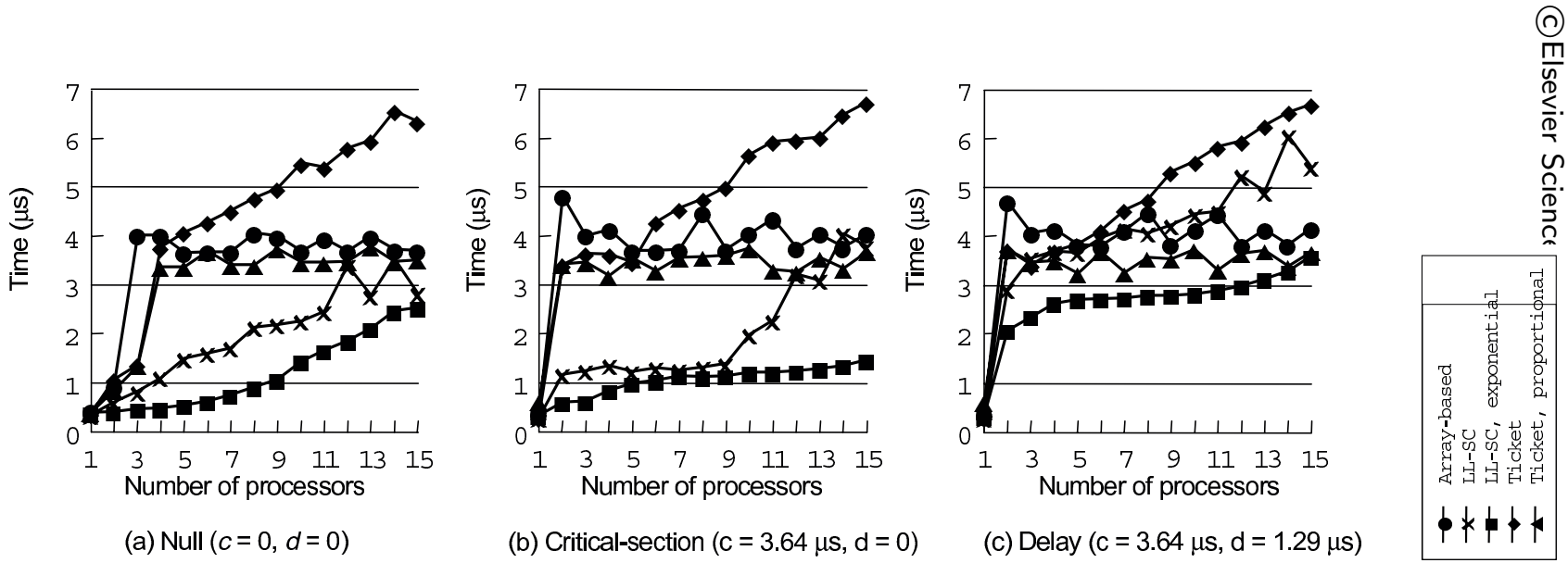
Effizientere Software-Locks: Array-bas. Lock II

```
enum wait { has_lock, must_wait };
struct lock { enum wait slots[numprocs] =
                { has_lock, must_wait, must_wait, ..., must_wait };
              int next_slot = 0};
void acquire_lock(lock* L, int* my_place) {
    // parameter my_place, below, points to a private variable
    *my_place = fetch_and_increment (L->next_slot);
    // returns old value
    if (*my_place % numprocs == 0)
        // avoid problems with overflow return value ignored
        atomic_add (L->next_slot, -numprocs);
    *my_place = *my_place % numprocs;
    do {} while (L->slots[*my_place] == must_wait);
    L->slots[*my_place] = must_wait;      // init for next time
}
void release_lock(lock* L, int* my_place) {
    L->slots[(*my_place + 1) % numprocs] = has_lock;
}
```

Mellor-Crummey/Scott: Algorithms for scalable synchronization on shared-memory multiprocessors, CACM 9(1) 1991

Effizientere Software-Locks: Performanz

Benchmark: `lock; delay(c); unlock; delay(d);`



- Einfaches *LL-SC* Lock am besten bei kleinem P
- Ticket-Lock (mit Backoff) und Array-Lock skalieren gut
- Fazit: Scheinbar relativ einfaches Problem (Locking) birgt grosse Herausforderungen bei der Implementierung

Semaphore

- ↯ Locking-Algorithmen a. d. Basis von *busy-waiting* (sog. *spin locks*) führen zu unnötigen Wartezyklen in MP-Systemen mit Mehrprogrammbetrieb

⇒ Prozessorauslastung kann durch Vermeidung von *busy-waiting* verbessert werden

⇒ Realisierung von gegenseitigem Ausschluss mit blockierendem Warten über Semaphore

Semaphore II

Definition (Semaphor)

Ein *Semaphor* ist eine Datenstruktur mit einer nichtnegativen Zählvariable und zwei speziellen Methoden P und V, die die Synchronisation von mehreren Prozessen erlaubt.

- Die *V-Operation* (niederl. *verhoog* = "erhöhe") inkrementiert den Zähler in einer atomaren Operation.
- Die *P-Operation* (niederl. *prolaag* = "erniedrige") dekrementiert den Zähler in einer atomaren Operation, sobald das Ergebnis nichtnegativ wird.

Dijkstra EWD 123)

Hinweis: Es lassen sich *binäre* und *allgemeine* Semaphore unterscheiden.

Semaphore III

Definition (Semaphor)

Ein *Semaphor* ist eine Datenstruktur ...

- Die *V-Operation* (holl. *verhoog* = "erhöhe") ...
- Die *P-Operation* (holl. *prolaag* = "erniedrige") ...
  sobald das Ergebnis nichtnegativ wird.

(Dijkstra EWD 123)

- P(S) muss blockieren (kein *busy-waiting!*), falls Zählvariable Null
- Zusätzlich zum internen Zustand des Semaphors müssen die am Semaphor wartenden Prozesse in einer Warteschlange ($\Rightarrow$ Teil der Datenstruktur) verwaltet werden!

Semaphore VI

Erzeuger-Verbraucher-Beispiel (aus (Dijkstra EWD 123))

```
begin integer number of queuing portions, buffer manipulation;
      number of queuing portions:= 0;
      buffer manipulation:= 1;
      parbegin
      producer: begin
              again 1: produce next portion;
                       P(buffer manipulation) ;
                       add portion to buffer;
                       V(buffer manipulation);
                       V(number of queuing portions);
                       goto again 1
               end;
      consumer: begin
              again 2: P(number of queuing portions);
                       P(buffer manipulation);
                       take portion from buffer;
                       V(buffer manipulation);
                       process portion taken;
                       goto again 2
               end
      parend
end
```

Semaphore V

- Semaphore garantieren gegenseitigen Ausschluss bei Zugriff auf geteilte *Betriebsmittel*
  - binärer Semaphor: nur eine Instanz des Betriebsmittels existiert (z.B. I/O-Gerät)
  - allgemeiner Semaphor: ex. mehrere Instanzen des Betriebsmittels / der Ressource (z.B. Speicher, zuteilbar in Seiten)
    $\Rightarrow$ zusätzlich Verwaltung der Ressource erforderlich!
- Wichtigestes Problem bei der Betriebsmittelzuteilung: Deadlock, d.h. parallele Prozess(or)e(n) halten Betriebsmittel und verhindern gegeseitig Abschliessen der Programmausführung
- Deadlocks können verhindert werden, falls nicht alle folgenden 4 Bedingungen zutreffen:
  - *Mutual exclusion* – Proz. haben exklusive Kontrolle über Ressourcen
  - *Nonpreemption* – Proz. geben Ressourcen nicht "freiwillig" frei
  - *Wait for* – Ressourcen können beim Warten auf andere gehalten werden
  - *Circular wait* – Ex. zyklische Abhängigkeit bei der Ressourcenanforderung

Synchronisation für Punkt-zu-Punkt-Ereignisse

- Software Methoden:
  - Interrupts
  - *Busy-waiting*: Verwendung einer Speicherstelle als *flag*
  - Blockieren: Verwendung eines Semaphors
- Vollständige Hardwareunterstützung: *full-empty* Bits für **jedes** Speicherwort
  - Gesetzt, wenn Wort "voll" mit neuen Daten (d.h. beim Schreiben)
  - Gelöscht, wenn Wort "geleert" / "verbraucht" (d.h. beim Lesen)
  - Geeignet für wort-basierte Erzeuger-Verbraucher-Synchronisation
    - Erzeuger: schreibt falls leer, setzt auf voll
    - Verbraucher: liest falls voll, setzt auf leer
  - Hardware garantiert Atomizität der Bitmanipulation beim Zugriff
  - Zentrales Problem: Flexibilität
    - Mehrere Verbraucher? ... mehrere Schreibzugriffe vor Lesen?
    - Hochsprachenunterstützung erforderlich (wann verwendet?)
    - Zusammengesetzte Datenstrukturen?

Barrieren

- Software: Algorithmen implementiert mit Locks, Flags und Zählern
- Hardware-Barrieren:
  - *Wired-AND* Leitung zusätzlich zu Adress-/Daten-Bus
  - Bei Ankunft auf *high* gesetzt, warten auf Ausgabe *high* zum Verlassen
  - I.d. Praxis mehrere Leitungen für Wiederverwendbarkeit

✓ Nützlich für globale und sehr häufig benutzte Barrieren

↯ Schwierig verschiedene Gruppen von Prozessoren zu unterstützen (noch schwieriger mit mehreren Prozessen pro Prozessor)

↯ Schwierig dynamisch Anzahl und Identität der Teilnehmer zu ändern (z.B. bei Prozessmigration)

⇒ Nicht üblich auf heutigen bus-basierten Maschinen

Barrieren

- Software: Algorithmen implementiert mit Locks, Flags und Zählern
- Hardware-Barrieren:
  - *Wired-AND* Leitung zusätzlich zu Adress-/Daten-Bus
  - Bei Ankunft auf *high* gesetzt, warten auf Ausgabe *high* zum Verlassen
  - I.d. Praxis mehrere Leitungen für Wiederverwendbarkeit
  - ✓ Nützlich für globale und sehr häufig benutzte Barrieren
  - ↯ Schwierig verschiedene Gruppen von Prozessoren zu unterstützen (noch schwieriger mit mehreren Prozessen pro Prozessor)
  - ↯ Schwierig dynamisch Anzahl und Identität der Teilnehmer zu ändern (z.B. bei Prozessmigration)
  - ⇒ Nicht üblich auf heutigen bus-basierten Maschinen

Betrachten im folgenden Software-Barrieren unterstützt durch einfache Hardware-Primitive!

Einfache Zentralisierte Barriere

Gemeinsamer Zähler für Anzahl angekommener Prozess(or)e(n)

- Wird bei Ankunft inkrementiert (gegenseitiger Ausschluß!)
- Überprüfen, wann Anz. Prozessoren erreicht $\Rightarrow$ Freigabe!

```
struct bar_type { int counter;
                  struct lock_type lock; int flag = 0;} bar_name;
BARRIER (bar_name, p) {
  LOCK(bar_name.lock);
  if (bar_name.counter == 0)
        bar_name.flag = 0;      /* reset flag if first to reach */
  mycount = bar_name.counter++; /* mycount is private */
  UNLOCK(bar_name.lock);
  if (mycount == p) {           /* last to arrive */
        bar_name.counter = 0;   /* reset for next barrier */
        bar_name.flag = 1;      /* release waiters */
  }
  else while (bar_name.flag == 0) {}; /* busy wait for release */
}
```

Quelle: Foliensatz von Culler/Singh/Gupta

Einfache Zentralisierte Barriere

Gemeinsamer Zähler für Anzahl angekommener Prozess(or)e(n)

- Wird bei Ankunft inkrementiert (gegenseitiger Ausschluß!)
- Überprüfen, wann Anz. Prozessoren erreicht $\Rightarrow$ Freigabe!

```
struct bar_type { int counter;
                  struct lock_type lock; int flag = 0;} bar_name;
BARRIER (bar_name, p) {
  LOCK(bar_name.lock);
  if (bar_name.counter == 0)
        bar_name.flag = 0;      /* reset flag if first to reach */
  mycount = bar_name.counter++; /* mycount is private */
  UNLOCK(bar_name.lock);
  if (mycount == p) {           /* last to arrive */
        bar_name.counter = 0;   /* reset for next barrier */
        bar_name.flag = 1;      /* release waiters */
  }
  else while (bar_name.flag == 0) {}; /* busy wait for release */
}
```

Wo ist das Problem?

Quelle: Foliensatz von Culler/Singh/Gupta

Zentralisierte Barriere

Problem: Wiederholtes Starten derselben Barriere funktioniert nicht!

$\Rightarrow$ Erneutes Betreten muss verhindert werden, bis alle Prozess(or)e(n) Barriere verlassen haben

Lösung: *Sense reversal*, d.h. Umkehr der Bedeutung des Freigabeflags bei wiederholter Verwendung

```
BARRIER (bar_name, p) {
  local_sense = !(local_sense); /* toggle private sense variable */
  LOCK(bar_name.lock);
  mycount = bar_name.counter++; /* mycount is private */
  if (bar_name.counter == p)
        UNLOCK(bar_name.lock);
        bar_name.flag = local_sense; /* release waiters*/
  else
      { UNLOCK(bar_name.lock);
        while (bar_name.flag != local_sense) {}; }
}
```

Quelle: Foliensatz von Culler/Singh/Gupta

Zentralisierte Barriere II

Alternative Realisierung mit `fetch&decrement`:

```
shared count : integer := P
shared sense : Boolean := true
processor private local_sense : Boolean := true
procedure central_barrier
    local_sense := not local_sense
            // each processor toggles its own sense
    if fetch_and_decrement (&count) = 1
        count := P
        sense := local_sense
                // last processor toggles global sense
    else
         repeat until sense = local_sense
```

Mellor-Crummey/Scott: Algorithms for scalable synchronization on shared-memory multiprocessors, CACM 9(1), 1991.

Zentralisierte Barriere II

Alternative Realisierung mit `fetch&decrement`:

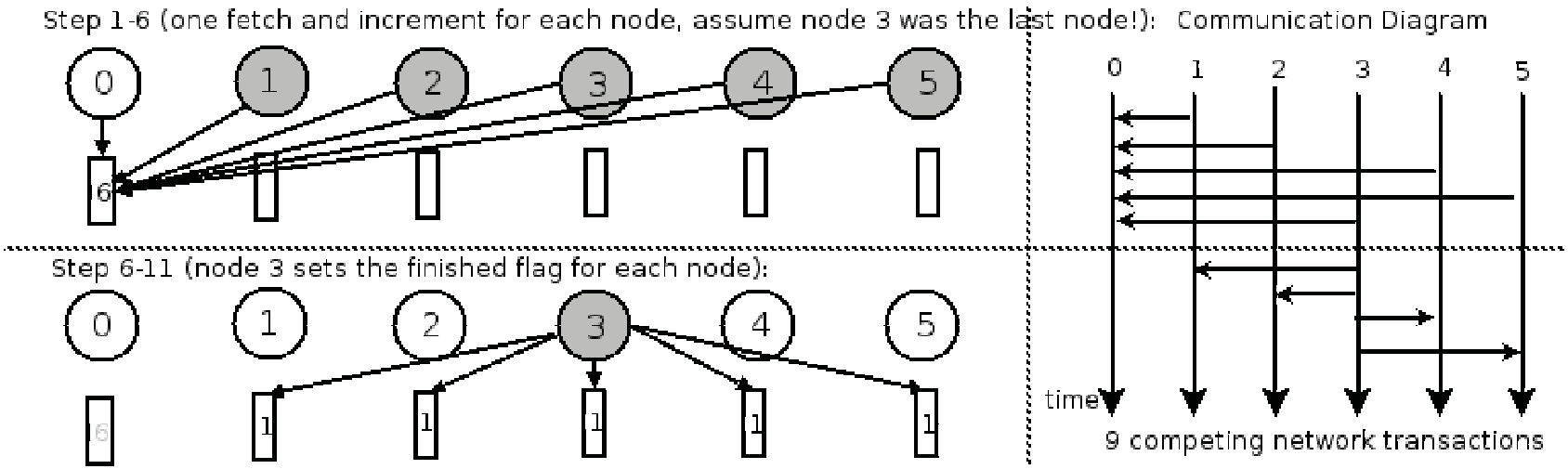
```
shared count : integer := P
shared sense : Boolean := true
processor private local_sense : Boolean := true
procedure central_barrier
    local_sense := not local_sense
            // each processor toggles its own sense
    if fetch_and_decrement (&count) = 1
        count := P
        sense := local_sense
                // last processor toggles global sense
    else
         repeat until sense = local_sense
```

Mellor-Crummey/Scott: Algorithms for scalable synchronization on shared-memory multiprocessors, CACM 9(1), 1991.

Wo liegt der Vorteil?

Zentralisierte Barriere III

Kommunikationsstruktur



Hoefler *et al.*: A Survey of Barrier Algorithms for Coarse Grained Supercomputers, Chemnitzer Informatik-Berichte, 12/2004.

Performanz der Zentralisierten Barriere

Latenz: Zentralisierte Barriere hat kritischen Pfad (= Sequenz abhängiger Operationen zur "Erfüllung" der Barriere) mit Länge $O(P)$

Kommunikationsoverhead: Barrieren sind globale Operationen $\Rightarrow$ möglichst wenige Bus-Transaktionen pro Barriere

Skalierbarkeit: Latenz und Kommunikationsaufwand sollten möglichst unabhängig v.d. Anzahl beteiligter Prozessoren sein (bzw. nur langsam damit wachsen)

Speicheraufwand: Idealerweise möglichst gering

Fairness: kein Bias bei der Auswahl, welcher Prozessor Barriere als letzter verläßt bzw. FCFS-Ordnung erhalten

Performanz der Zentralisierten Barriere

Latenz: Zentralisierte Barriere hat kritischen Pfad (= Sequenz abhängiger Operationen zur "Erfüllung" der Barriere) mit Länge $O(P)$

Kommunikationsoverhead: Barrieren sind globale Operationen $\Rightarrow$ möglichst wenige Bus-Transaktionen pro Barriere

Skalierbarkeit: Latenz und Kommunikationsaufwand sollten möglichst unabhängig v.d. Anzahl beteiligter Prozessoren sein (bzw. nur langsam damit wachsen)

Speicheraufwand: Idealerweise möglichst gering

Fairness: kein Bias bei der Auswahl, welcher Prozessor Barriere als letzter verläßt bzw. FCFS-Ordnung erhalten

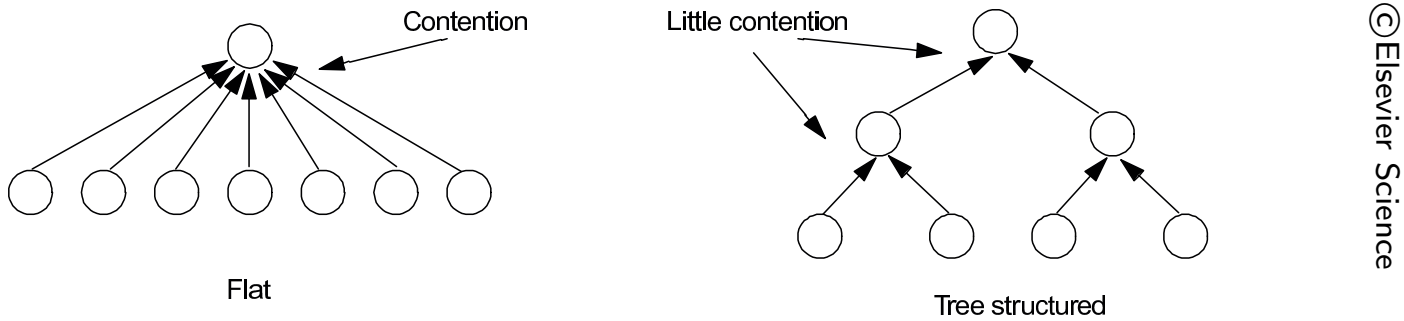
Hauptprobleme der zentralisierten Barriere: Latenz und Kommunikationsoverhead!

Verbesserte Agorithmen für Barrieren

Grundidee: Latenz *und* Kommunikationsoverhead reduzieren durch größere Lokalität der Interaktion

Prinzipien:

- Baum-organisierte Barrieren



  $\Rightarrow$ nur k von P Prozessoren greifen auf gleiche Speicherstelle zu (mit k Verzweigungsgrad des Baums)

- "Ausbreitungs-Barrieren" mit paarweiser Kommunikation in mehreren "Kommunikationsrunden"

Combining-Tree Barrier

Nach (Yew/Tzeng/Lawrie 1987), d.h. *einfache* baum-basierte Barriere

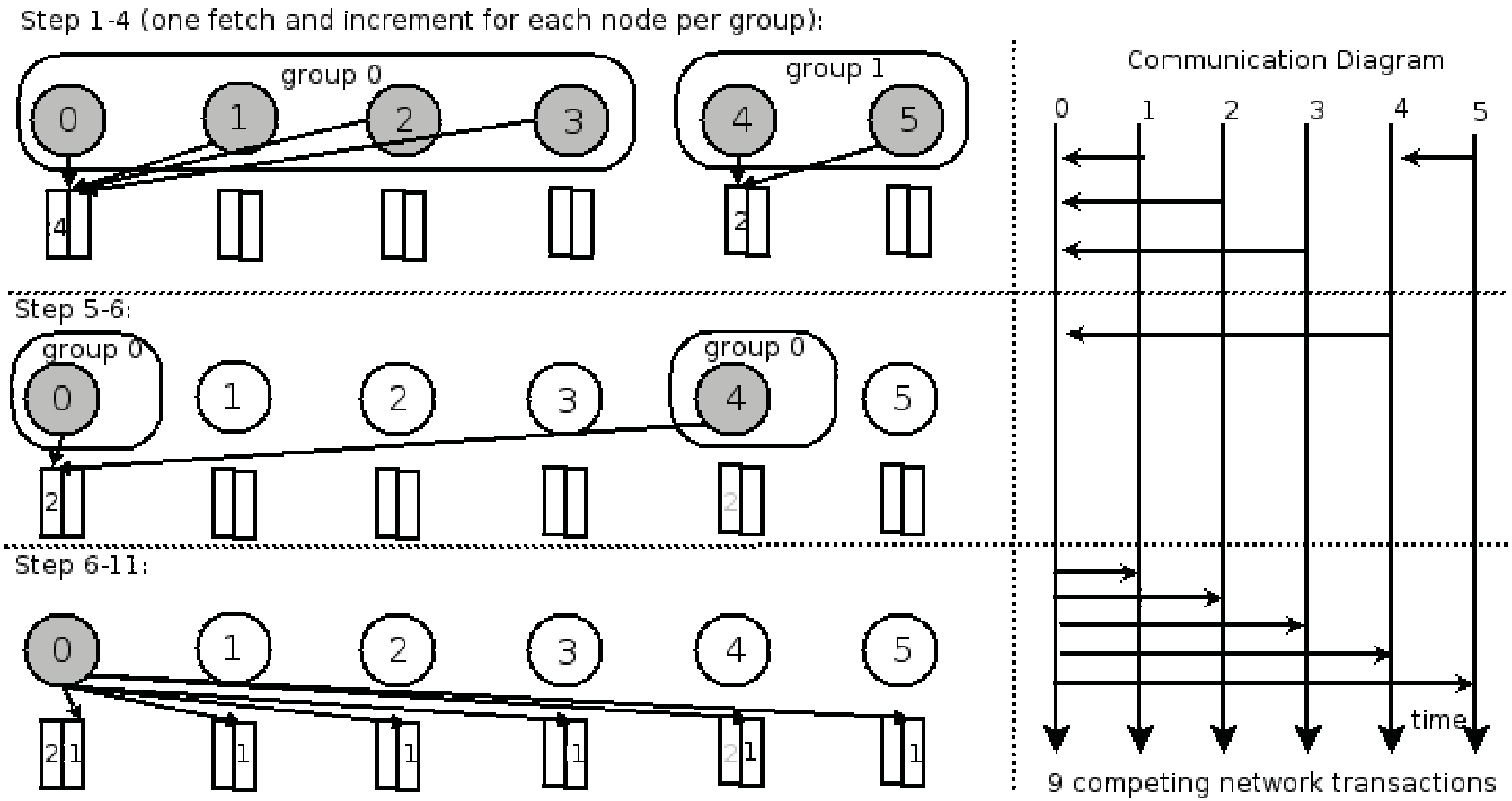
- Baumstruktur zur Beschleunigung der Kommunikation
- Prozessoren werden in Untergruppen eingeteilt
- Jede Untergruppe verwendet zunächst eine (separate) zentralisierte Barriere
- Sobald alle Prozessoren einer Gruppe die jeweilige Barriere erreicht haben, wird Synchronisation auf nächster Ebene im Baum fortgesetzt $\Rightarrow$ letzte/erste Prozessor, die Barriere erreichen bilden neue Gruppen
- Prozess wird wiederholt, bis nur eine Gruppe verbleibt
- Ausgewählter Prozessor (erster/letzter) der letzten Gruppe informiert *alle* über Erreichen der (Gesamt-)Barriere

... ex. Varianten des Grundschemas

... z.B. durch Verwendung eines Benachrichtigungsbaums

Combining-Tree Barrier II

Kommunikationsstruktur



Hoefler *et al.*: A Survey of Barrier Algorithms for Coarse Grained Supercomputers, Chemnitzer Informatik-Berichte, 12/2004.

Tournament Barrier

Nach (Hensgen *et al.* 1988), d.h. “Tournier-Barriere”

- Grundprinzip ähnlich wie *Combining-Tree Barrier*, aber mit binärer Baumstruktur
- Je zwei Prozessoren “treten in Tournier gegeneinander an”
- Gewinner steht *a-priori* fest
- Dieser fährt auf der nächsten Baumebene mit “Tournier” fort

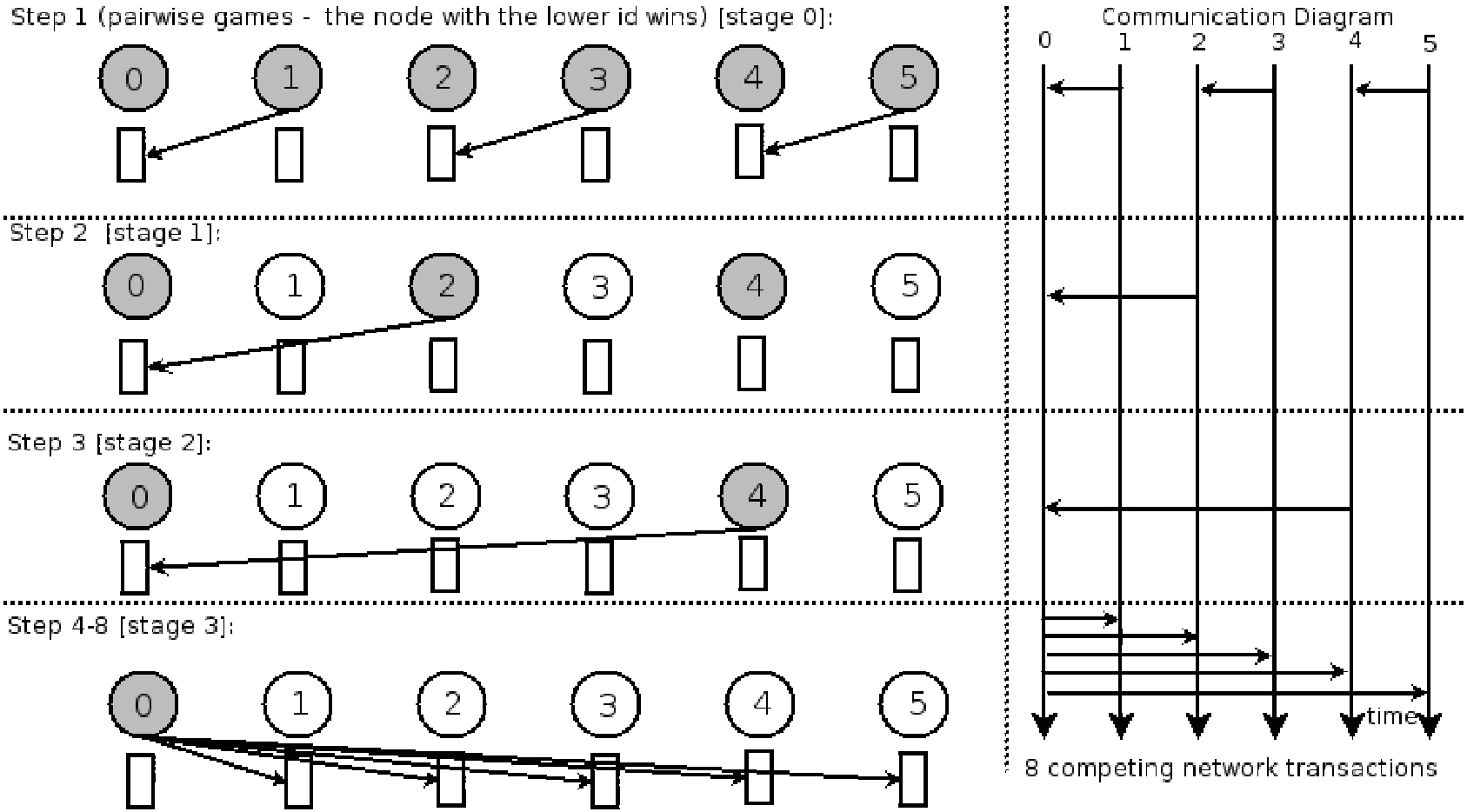
✓ Keine Notwendigkeit für `fetch&op` Instruktionen

✓ Geeignet für cache-kohärente MP-Systeme

Mit *wake-up tree* (Mellor-Crummey/Scott 1991) kann erreicht werden, dass jeder Prozessor eigenes, statisch alloziertes Benachrichtigungsflag verwendet
$\Rightarrow$ kein *broadcast* bei Erreichen der Barriere notwendig!

Tournament Barrier II

Kommunikationsstruktur



Hoefler *et al.*: A Survey of Barrier Algorithms for Coarse Grained Supercomputers, Chemnitzer Informatik-Berichte, 12/2004.

Dissemination Barrier

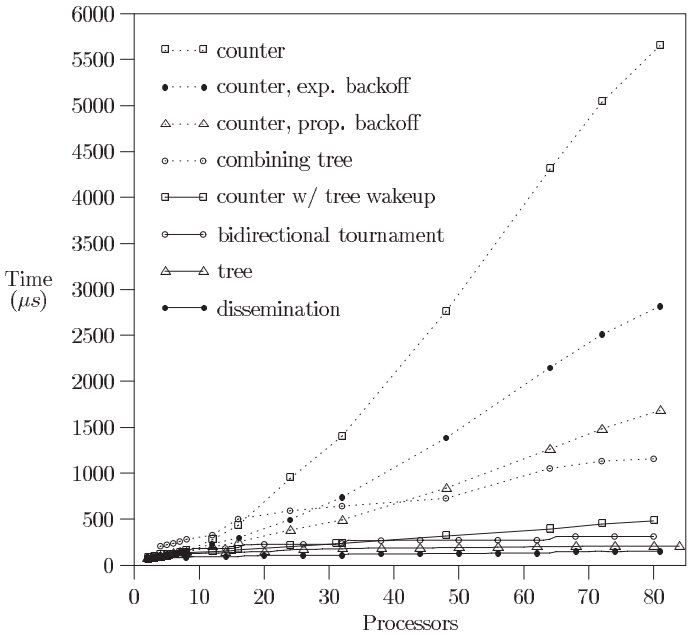
Grundidee: Paarweise Synchronisierung zwischen Prozessoren
($\hat{=}$ Original "butterfly barrier" (Brooks 1986))

- Synchronisation läuft über $\log_2 P$ Runden
- In k-ter Runde Synchronisation d. Prozessorpaars $(i, 2^k \text{xor} i)$
- Falls $P \neq 2^n$: Prozessoren müssen "Doppelrolle" übernehmen

Erweiteres Schema: "Dissemination barrier" (Hensgen *et al.* 1988)

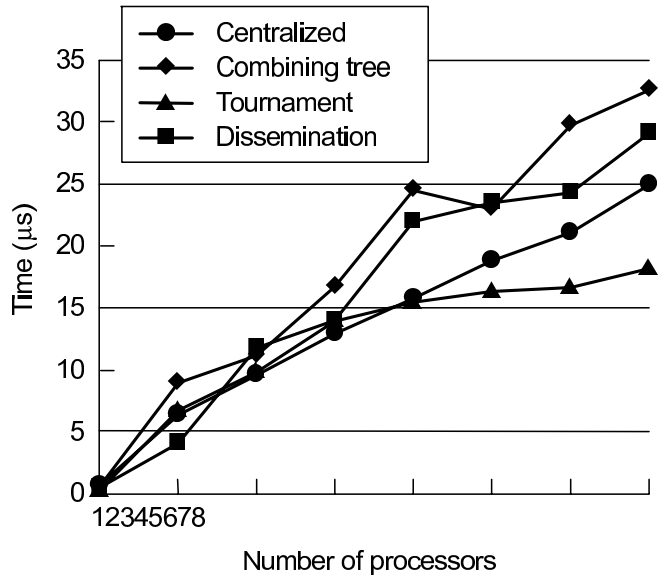
- Synchronisation nicht mehr paarweise
- Je Runde benachrichtigt Proz. i Proz. $i + 2^k \, mod P$
- Wechselnder Satz von Benachrichtigungsvariablen pro Runde zur Vermeidung von Interferenzen
- Warte-Flags werden statisch bestimmt, keine zwei Prozessoren verwenden gleiches Flag
  $\Rightarrow$ ermöglicht *local spinning*

Barrieren: Performanz



Mellor-Crummey/Scott: Algorithms for scalable synchronization on shared-memory multiprocessors, CACM 9(1), 1991.

Benchmark auf BBN Butterfly (DSM-System mit Switch als Verbindungsstruktur)



Benchmark auf SGI Challenge (SMP-System)

Synchronisation: Zusammenfassung

- Synchronisation von
  - Ereignissen oder
  - zum gegenseitigen Ausschluss
- Gegenseitiger Ausschluss realisiert mit Locks
  - Fortschrittliche Algorithmen versuchen Latenz und Kommunikationsoverhead zu vermeiden,
  - gute Skalierbarkeit (mit Anz. Prozessoren) und
  - Fairness zu bieten.
- Blockierender gegenseitiger Ausschluss: Semaphore
- Ereignissynchronisation über viele/alle Prozessoren: Barrieren
  - Fortschrittliche Algorithmen bieten hohe Lokalität der Speicherzugriffe und damit
  - (relativ) gute Skalierbarkeit *auch in DSM-Systemen*.