Amdahl'sches Gesetz

$$\frac{T_1}{T_n} = \frac{1}{s + p/n} = \frac{1}{1 - p + p/n} = S$$

T_n: Laufzeit mit n Prozessoren

s: Sequentieller Anteil
p: Paralleler Anteil

Speedup: $\frac{T_1}{T_n} = S$

Effizienz: S/n

Bsp.: $S_2 = \frac{T_1}{T_2} = \dots$

$S_4 = \frac{T_1}{T_4} = \frac{T_2}{T_4} \cdot \frac{T_1}{T_2} = \frac{S_4}{S_2}$

$S_{2\to4} = \frac{S_4}{S_2}$

gilt Speedup wenn man von n=2 auf n=4 geht.

Work/Span Analyse

$$T_1/n \le T_n \le T_\infty + \frac{T_1 - T_\infty}{n}$$

↑ greedy Scheduler

n: max. Anzahl von chunks

optimaler Scheduler:
$T_n \le T_1/n + T_\infty \le 2T_{opt}$

* ist der Faktor 2 optimal.

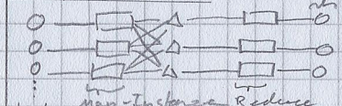
Falls $n \ll \frac{T_1}{T_\infty} \Rightarrow T_n \approx \frac{T_1}{n}$

$\Rightarrow$ Speedup $\approx n$

Parallelität: Gesamtprozess wird durch mehrere parallele Teilprozesse gelöst

Concurrency: Parallele Prozesse, die um gemeinsame Ressourcen kämpfen

Map-Reduce



Die Map-Instanzen verarbeiten die Daten und speichern die Ergebnisse "zeitlich" ab (Δ). Die Zwischenergebnisse werden von der Reduce Instanz gesammelt und die Ergebnisse (Output) generiert.

Cache-Kohärenz

Mehrere Prozessoren mit eigenen Caches. Problem: in den Caches für die gleiche Memory adresse enthalten verschiedene Daten.
→ Protokoll nötig

False Sharing

Mehrere CPUs mit Caches. Die Caches überlappen sich, da jeweils ganze words in den Cache geladen/geschrieben werden. Ändert nun eine CPU etwas in den Speicherblock, so muss die andere CPU diese neu laden
→ Performance einbüsse

Deadlockvermeidung

Lineare, globale, Ordnung einführen:
Transfer (a, b) lockt a, dann b
Transfer (b, c) lockt b, dann c
Transfer (c, a) lockt a, dann c
⇒ kein Deadlock

Pro/Con Locks

+ Korrektheit durch klare Isolation
+ unterstützt strukturierte Programmierung
- Verwaltungsaufwand
- längere sequentielle Abschnitte
- Overhead z.B. bei Serialprozessen
- pessimistischer Ansatz
- Gefahr von Deadlock, Starvation, Misspannen

R/W-Locks

- Gefahren:
  - Reader lesen alte Daten ("stale data")
  - Reader/Writer warten für immer (starvation)

- Prioritätsregel*: falls ein Writer wartet, kann kein Reader starten
  - ⇒ Writers können Readers verhungern lassen

* ⇒ Neue Regeln: hat es derzeit nur Reader, so hat ein neuer Writer Priorität
  - hat es nur Writer, oder Reader und Writer haben beide dieselbe Priorität

- lohnt sich nur, wenn der Protokollaufwand nicht mehr Bottleneck wird

Petersons Algo: Lösung

für gegenseitige Ausschluss. Ist starvationsfrei (und gegenseitiger Ausschluss natürlich)

Leslie Lamport's Bakery Algo

- gegenseitiger Ausschluss für N Prozesse
- Beruht auf der Idee einen jeweils höheren Nummer

Contention Problem

Alle Threads kämpfen um die gleiche Ressource ("spinning")
Auswege:
- Backoff: nach fehlgeschlagenen versuch mit wachsender Zeit warten und so den anderen Threads Chance geben.
- Elimination: fast gleichzeitige Pop und Push (Stack) können sich löschen, da sie sich aufheben.

ABA-Problem

$P_{1,2}$ Threads

- P_2 liest A aus Memory
- P_2 ändert A zu B und dann wieder zu A (im Innern entspricht neue A nicht der alten A)
- P_1 sieht keine Änderung und macht weiter

~ oder ~
Ein Element aus einer Liste wird gelöscht. Dann fügt P_2 ein neues Element ein, wo vorhin das gelöschte war. P_1 sieht keine Unterschied da beide Objekte gleiche Zeiger haben.

Lazy Removal (Liste)

1. Phase: Knoten wird "entfernt" indem ein "gelöscht"-Flag gesetzt wird
2. Phase: physisches entfernen in späterem Durchlauf

Pointer Tagging (Liste)

Der Pointer wird zum Tupel (pointer, tag), womit label-pointer bim Compare Exchange entfällt

Transaktionsmodell

Jede Transaktion ist eine Folge von Instruktionen, die als ganzes Atomar sind.
Jede Transaktion führt die Instruktionen aus. Am Ende wird entschieden ob diese "committed" werden, also dauhgezogen werden und alle Resultate gespeichert werden, oder ob ein "abort" stattfindet, was zum verwerfen der Änderungen führt.

Locks vs. Transaktion

• pessimistisch	• optimistisch
• schützt Ressourcen	• schützt Ausführung
• potenzielle Deadlocks	• potenzielle Starvation
• HW unterstützung einfach	• HW unterstützung schwer

Semaphore

s: Zähler (wieviele dürfen gleichzeitig?)
P(): "reservieren", "Eingang"
V(): "freigeben", "Ausgang"
busy-waiting (spinning) als auch blockierend implementierbar

Barriere

Alle Threads treffen sich, bevor sie weiter machen.
→ Rendez-Vous: 2 treffen sich:

```
Thread A und B:
a = new Semaphore(0); // A arrived
b = new Semaphore(0); // B arrived
Thread A: ... a.V(); b.P(); ...
Thread B: ... b.V(); a.P(); ...
```

Sequentielle Konsistenz

(wird eigentlich auf die Maschiene angewendet und nicht Programme)

Thread — Befehlsfolge

A ≡ $A_1; A_2; A_3$

B ≡ $B_1; B_2$

Kann man die Reihenfolge der A und B Operationen einhalten werden: ok: A_1, B_1, A_2, A_3, B_2

nicht-ok: A_2, B_2, A_1, B_1, A_3

Volatile

- enthält compiler Optimierung aus
- Kein "echtes Caching" vor lese/schreibe wird immer "gegneed" mit Mainmemory

Lineare Konsistenz

Achtung: Pro Thread müssen die Zeitlinien "⟶" disjunkt sein.

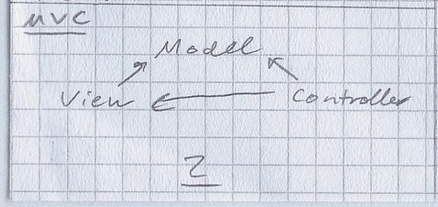
"Jeder parallele Methodenablauf im Objekt "p" ist äquivalent zu einem sequentiellen Ablauf, in welchen alle Methodenaufführungen zu disjunkten Intervallen in ihrer chronologischen Reihenfolge vorkommen"

Komposition

Seien p und q zwei linear konsistente Objekte. Dann ist auch die Komposition pq linear konsistent.

Seien p und q sequentiell konsistent. Die Komposition pq ist nicht zwingend sequentiell konsistent

MVC



2

wait-free

Jeder Thread macht nach endlicher Zeit Fortschritt.

lock-free

Die Menge aller Threads, die auf ein Objekt wirken machen nach endlicher Zeit als Ganzes Fortschritt. Einzelne Threads können jedoch uneingeschränkt verdrängt werden.

Lock-free/wait-free bauen

lassen sich aus atomaren "Registern" oder kund mittels Consensus Objekten bauen

Atomarer "Register"

- Operationen: Read/Write
- linear konsistent mit nur einem Wirkungspunkt
  => jede Read-Operation liefert neuesten Wert
  => "echtzeitreihenfolge" der nicht-überlappender Operationen bleibt bestehen

Konstruktionskaskade (Register)

SRSW: Single Read, Single Write

→ MRSW → MRMW

SRSW→MRSW

- Pro Reader 1 SRSW, Writer schreibt der Reihe nach in die SRSW. Nicht atomar
- oder: Jedes SRSW hat neben dem aktuellen Wert noch einen Timestamp. Writer schreibt immer nur in reg[i][i]. Reader mit ID "rid" sucht neuesten Wert in reg[i][rid], i=0...n und schreibt diesen in reg[rid][i], i=0...n. Timestamp gibt neuesten Wert an. Ist atomar.

MRSW→MRMW

Liste (1D Array) von MRSW Register mit Timestamp. Writer schreibt ins reg[wid], wid = Writer ID. Leser liest neuesten Wert aus reg[i], i=0...n. Sind zwei Timestamps gleich, so wird das mit kleinerem i verwendet.

Consensus Objekt

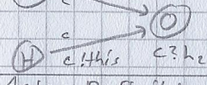
Atomares Objekt mit der Operation T propose(T v), T=int, bool, ...

Die Operation erscheint sequenziell ausgeführt. Allen aufrufenden Threads wird eine "decision"/"agreement" durch "propose" zurückgegeben, was ein Wert ist, der mal propose übergeben wurde.

CSP Notation:

- "seq" sequentiell
- "par" Parallel
- Lese operation "c?x"
- Schreiboperation "c!E"
- "c?x" und "c!E" ergibt: x = E

H + H + O → H_2O



H ⟸ c!this;

O ⟸ c?h1; c?h2

Actor Definition

Eine "computational entity", die als Antwort auf eine Message folgendes parallel ausführen kann:

- endliche Anzahl an Nachrichten an andere Actor schicken
- endliche Anzahl neuer Actor erzeugen
- Entscheiden, wie auf die nächste Nachricht reagiert wird.

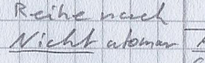
↳ parallel ausführbar

MPI

- Prozesse sind Kommunikationspartner
- Jeder Prozess hat eine ID. Dadurch kann verschiedener verteilt berechnet werden trotz gleichen Code
- Enthält blockierende sind nicht-blockierende senden/empfangen
- Es können Gruppen gebildet werden. Die Standard Gruppe ist "world". In jede Gruppe gibt es einen Kommunikator, der die Kommunikation innerhalb der Gruppe steuert.
- point-to-point kommunikation, Broadcast, ...

Prozessornetzwerke

- Prozessornetzwerk:
  - Knoten: Prozessor
  - Kante: Kommunikation
  - kanal
- Verbindung mit Memory
- Bsp.: "6x6 PE"
  - Inputs: north, west
  - Outputs: east, south



Interlocked (MSDN)

- Add(ref int, int), Decrement(ref int)
- Increment(ref int)
- CompareExchange<T>(T ref loc, T value, T comperand): T
  ↳ loc: Location whose value is compared to "comperand" and possibly replaced with "value".
  ↳ value: the value to put into "loc", if "loc" == "comp."
  ↳ Returns the original ("old") value in "loc". We compare it always with "comperand" to know if it was successful

Singleton

Singleton: singleton Var als volatile, Lock ums erzeugen des Objekts, also ein if(singleton == null){ lock ...}

Map-Reduce

```
void map(string id, string content){
  foreach(Word w in content)
    EmitIntermediate(w, pos);}

void reduce(string word, IEnumerable
  occList){ int count=0;
  foreach(in in occList) count++;
  EmitResult(word, count);}
```

RW-Lock

```
void AquireRLock(){
  lock(this){ while(writing)
    Monitor.Wait(this);
    readers++;}
}
void ReleaseRLock(){
  lock(this){
    if(--readers==0)
      Monitor.Pulse(this);
  }}
void AquireWLock(){lock(this){
  while(writing) Monitor.Wait(this);
  writing=true;
  owner=Thread.CurrentThread;
  while(readers!=0) Monitor.Wait(
                    this);
}}
void ReleaseWLock(){lock(this){
  if(owner==Thread.CurrentThread){
    writing=false;
    Monitor.PulseAll(this);
  }else{ throw new Exception();}}
Klasse: int readers=0,
  writers=0; bool writing=false;
Thread owner=null;
```

Petersen Algo für 2 Threads

(gegenseitiger Ausschluss).

```
Klasse: bool busy[]={false,false};
int victim=0;
Methode, i=Thred Nr.: sleep?
busy[i]=true; victim=i; //
while(busy[1-i]&&victim==i){}
  CS; //critical section
busy[i]=false;
```

(Read/Write atomar → Interlocked verwenden)

Lambort's Bakery Algo (N Threads)

```
Klasse+Lvl:
for(i=1; i<=N; i++) {
  t[i]=0; c[i]=false;}

Methoden-Level, i=ThreadID:
 c[i]=true;
t[i]=max(t[1],...,t[N])+1; c[i]=false;

for(j=1; j<=N; j++){
 while(c[j]){}
 while(t[j]>0 && <(t[i],i)>(t[j],j)){}
}
 CS;
t[i]=0;
```

Stack (LIFO) (ABA-Problem da!)

```
void Push(Item n){
  do{
    n.next=top;
   if(Interlocked.CompareExchange(
        ref top, n, n.next)==n.next)
       break;
  }while(true);}
Item Pop(){
  Item curtop;
  do{ curtop=top;
    if(curtop==null) break;
    if(Interlocked.CompareExchange(
       ref top, curtop.next, curtop)
       ==curtop)
     break;
  }while(true);
  return curtop;
}
klasse: Item top=null;
```

Liste (ABA-Problem vorhanden)

```
Klasse: Item head;
Init: head=new Item(int.MinValue)
      head.next=new Item(int.MaxValue)

void Insert(Item newItem){
   Item cur=head;
   do{
    while(cur.next.val<newItem.val)
       cur=cur.next;
    newItem.next=cur.next;
   if(Interlocked.CompareExchange(
        ref cur.next, newItem,
          newItem.next)==
            newItem.next) break;
  }while(true);
}
```

Semaphore: Spinlock Variante

```
class SpinLock{ int s=1;
public void P(){
  int prev, cur=0;
  do{ prev=s; //spinlock/busy waiting
   if(prev>0)
     cur=Interlocked.CompareExchange(
        ref s, prev-1, prev);
  }while(prev==0 || cur!=prev);
}
public void V(){Interlocked.Increment(
                  ref s);}
}
```

Semaphore: Sleeping Barber Variante

```
class Semaphore{ int s;
 object mutex=new object();
public Semaphore(int n){ s=n;}
public void P(){
 lock(mutex){ s--;
  if(s<0) Monitor.Wait(mutex);
 }
}
public void V(){
 lock(mutex){ s++;
  if(s<=0) Monitor.Pulse(mutex);
 }
}}
```

Singleton

```
public class Singleton{
  private static volatile Singleton inst;
  private static object mutex=new Object();
  private Singleton(){}
  public static Singleton Instance{
    get{
      if(inst==null){
        lock(mutex){
          if(inst==null)
            inst=new Singleton();
        }
      }
      return inst;
    }
  }
```

Divide and Conquer

Quick Sort, Merge Sort, Karatsuba, Fast Fourier Trans.

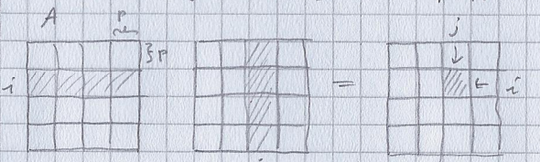
Matrix Multiplikation

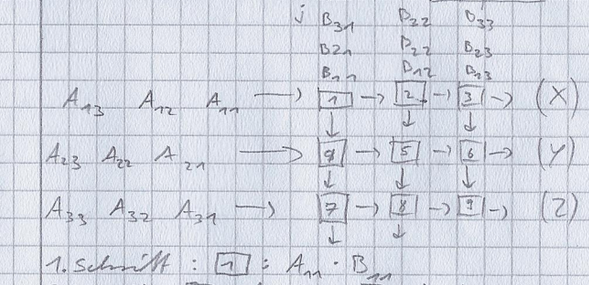
A, B $r \times r$ Matrizen, die in $p \times p$ Matrizen zerteilt werden

$C = A \cdot B$

$\Rightarrow C_{ij}$ ($p \times p$ Quadratblock)

$C_{ij} = A_{i1} B_{1j} + A_{i2} B_{2j} + \dots + A_{ip} B_{pj}$





1. Schritt: $\boxed{1}: A_{11} \cdot B_{11}$

2. Schritt: $\boxed{1}: A_{12} B_{21}$, $\boxed{2}: A_{11} B_{12}$, $\boxed{4}: A_{21} B_{11}$

3. Schritt: $\boxed{1}: A_{13} B_{31}$, $\boxed{2}: A_{12} B_{22}$, $\boxed{3}: A_{11} B_{13}$, $\boxed{4}: A_{22} B_{21}$, $\boxed{5}: A_{21} B_{12}$, $\boxed{7}: A_{31} B_{11}$

:

$(X) = C_{11}, C_{12}, C_{13}$

$(Y) = C_{21}, C_{22}, C_{23}$

$(Z) = C_{31}, C_{32}, C_{33}$

All-Pairs kürzeste Wege

Floyd-Algo

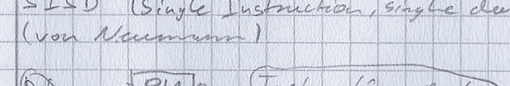
```
for(k=0; k<n; k++)
  for(j=0; j<n; j++)
    for(i=0; i<n; i++)
      A[i,j]=min(A[i,j], A[i,k]+A[k,j]);
```

$L_k(i,j)$

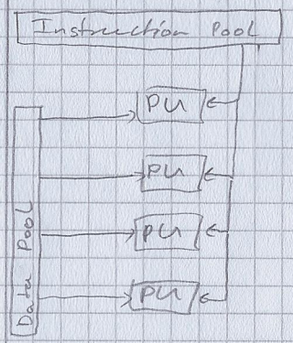
$L_k(i,j)$ kann für i,j parallel berechnet werden.

$\Rightarrow O(n)$

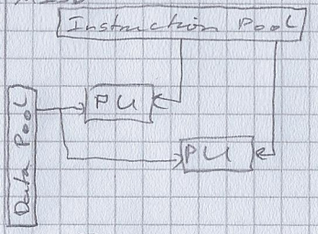
SISD (Single Instruction, single data) (von Neumann)



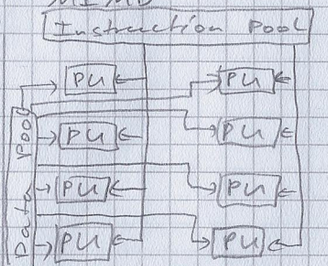
SIMD (Vektor)



MISD



MIMD



Unity Formalismus

- benutzt Nichtheit der HW und Implementationsdetails
- Schrittweise Verfeinerung, wobei in jedem Schritt die Korrektheit garantiert wird. Schritte: Spezifikation → ... → Implementation
- Ein und dasselbe Programm kann für verschiedene HW Architekturen generiert werden
- Syntax:
  - Programmabschnitte: declare, initially, always, assign
  - Anweisungen: $=, \in, \mapsto$, unless, stable, invariant, ensure, $\neg, \neq, <, >, \leq, \geq, \subseteq, \in$, min, +, −, path

Unity... (code)

```
Program shortestpath_floyd_algo
declare
    n,k: integer,
    D: array [0..n-1, 0..n-1] of
                integer
initially
    n = 10 #
    k = 0 #
    ||  i,j : 0 <= i < n and
              0 <= j < n ::
                D[i,j] = rand()%100 >
assign
    ||  i,j : 0 <= i < n and 0 <= j < n ::
            D[i,j] := min(D[i,j], D[i,k] +
                          D[k,j]) > ||
    k := k+1 if k < n - 1
end
```

UMA: Uniform Memory Access

- Zentraler Speichercontroller
- Einheitliche Zugriffszeit

Non-Uniform Memory Access (NUMA)

- integrierter Speichercontroller
- Differenzierte Zugriffszeit

Thread-Pool Pro's

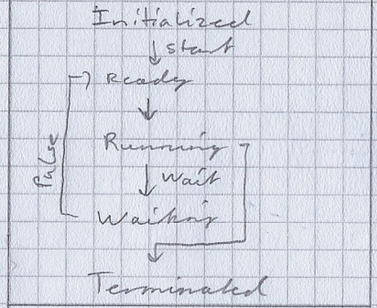
- Orchestrierung abgeben ans System (don't invent the wheel)
- Unterstützt wiederverwendung von Threads → weniger Overhead
- Entkoppelt Problemraum (Task) vom Systemraum (Thread) und führt zu systemunabhängigen Programme
- Erlaubt systemabhängige Threadanzahl (Faustregel: #Prozessoren + #I/O)

Pulse/Pulse All

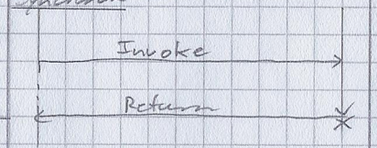
Pulse: Alle warten auf gleiche Bedingung, oder nur ein Thread kann fortfahren, wenn Bedingung erfüllt.

Pulse All: Alle Threads warten auf verschiedene Bedingungen oder mehrere Threads fortfahren dürfen, wenn Bedingung gegeben ist.

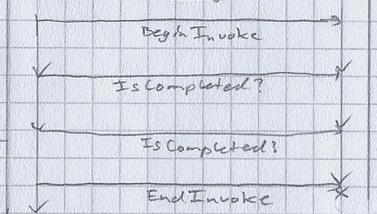
Thread-Zustände



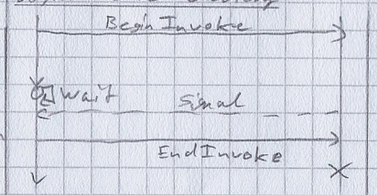
Synchron



asynchron polling



asynchrone waiting



asynchron callback