Verteilte und Parallele Programmierung

Mit Virtuellen Maschinen

PD Stefan Bosse

Universität Bremen - FB Mathematik und Informatik

Parallelisierung von Algorithmen und Programmen

Wie können Algorithmen und Programmen parallelisiert werden?

Wann ist Kontrollpfadparallelität sinnvoll?

Was unterscheidet Datenpfad- von Kontrollpfadparallelität?

Unterschied zwischen Koroutinen, Threads, und Prozessen

Asynchrone vs. Synchrone Verarbeitung

Parallelisierungsmethoden

Sequenzielle Schleifen und Abrollung

- Schleifen stellen eine reine Sequenz von elementaren Prozessen dar (parametrisierbare Replikate): $I = i_1 \rightarrow i_2 \rightarrow \ldots$
- Das Abrollen von Schleifen ist eine der gängigsten Parallelisierungsmethoden bei der Schleifen durch Replikation der Schleifeninstruktionen (Schleifenkörper) in eine sequenzielle und parallelisierbare Anweisungssequenz
  - teilweise (partitioniert)
  - oder vollständig transformiert werden.
- Dabei wird der Schleifenkörper $B(I,p)$ in Abhängigkeit von der Iterationsvariable p parametrisiert
- Kontrollpfaparallelität durch Partitionierung: Jede Instanz oder eine Menege von B (p_i) kann einem separaten Prozess P_i zugeordnet werden

Parallelisierungsmethoden

```
for i = i1 to i2 do
  for j = j1 to j2 do
    ..
    IB(i,j,x(i,j),x(i+1,j),..,);
  end
end
_____________________________________
IB(i1,j1,x(i1,j1),x(i1+1,j1),..) ||
IB(i1+1,j1,x(i1+1,j1),x(i1+2,j1),..) ||
IB(i1+n,ij+m,x(i1+n,ij+m),x(i1+n+1,j1+m),..) ||
..
```

Def. 1. Schleifenparallelität und mögliche Parallelisierung (von Datenabhängigkeit verschiedener Instanzen B_i begrenzt)

Parallelisierungsmethoden

- Zu beachten sind Datenabhängigkeiten zwischen einzelnen Anweisungsiterationen und Kosten durch die Abrollung (Overhead)
- Beispiel: Abrollbar aber nicht parallelisierbar durch Datenabhängikeit $IS_b(IS_{b-1}(IS_{b-2}(..))$!

```
X := 1;
for i = a to b do
  IS: X := X*i;
end
```

Parallelisierungsmethoden

Deklarative Beschreibung von Vektoroperationen

- In der Bildverarbeitung werden Vektoroperationen (und Matrixoperationen) sehr häufig verwendet. Dabei werden häufig die gleichen Operationen auf alle Pixel eines Bildes (Feldelemente) angewendet → **Datenpfadparallelität**.
- Anstelle der iterativen Berechnung mittels Schleifen definiert man deklarativ die Berechnung eines Pixels (**Punktoperator**) oder eines Bereiches des Bildes (**Lokaloperator**):

Parallelisierungsmethoden

```
img' = F(img):: = { f(x)|x ∈ img }
```

```
procedure F (img: vector of type):
begin
  return f(img)
end
```

Def. 2. Deklarative Beschreibung von Vektoroperationen

Parallelisierungsmethoden

Beispiel Punktoperator

```
1   PROCEDURE rgb2color(g_red,g_green,g_blue: VECTOR OF gray):
2             VECTOR OF color;
3   (* transform gray image to color image *)
4   VAR res: VECTOR OF color;
5   BEGIN
6     res.red   := g_red;
7     res.green := g_green;
8     res.blue  := g_blue;
9     RETURN res;
10  END rgb2color;
```

```
1   PROCEDURE gray2color(img: VECTOR OF gray): VECTOR OF color;
2   (* transform gray image to color image *)
3   BEGIN
4     RETURN rgb2color(img,img,img)
5   END gray2color;
```

```
1   PROCEDURE color2gray(img: VECTOR OF color): VECTOR OF gray;
2   (* transform color image to gray image *)
3   BEGIN
4     RETURN (img.red + img.green + img.blue) DIV 3
5   END color2gray;
```

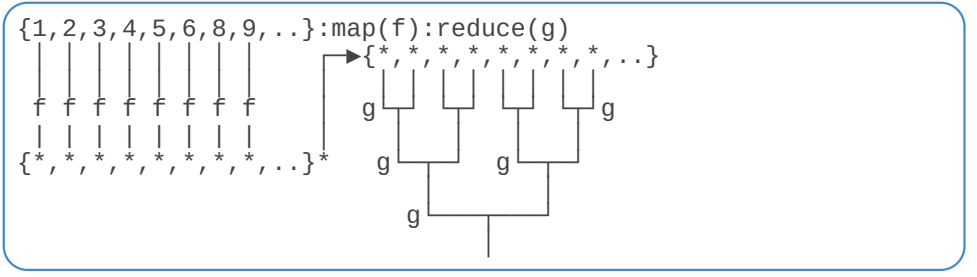
Abb. 1. Algorithmus: Transformation von Farb- in Grauwertbilder [2]

Parallelisierungsmethoden

Listen mit Map&Reduce

- Funktionale Komposition bietet inherente Parallelität bei Listenoperationen ("Punktoperatoren")
- Häufige Listenoperationen (und auch bei Arrays) sind:
  - **map** transformiert eine Liste in eine neue Liste gleicher Länge über eine Transformationsfunktion (Punktoperator) $f(x)$: $x \rightarrow y$
  - **reduce** reduziert eine Liste mittels eines Akkumulators entweder auf ein Element oder eine neue Liste mit kürzerer Länge über eine Reduktionsfunktion $f(accu,x)$: $accu,x \rightarrow accu$
- Mapping kann vollständig parallel ausgeführt werden (Unabhängigkeit der Teiloperationen)
- Reduce kann als Baumstruktur teil-parallel ausgeführt werden (dynamische Partitionierung)

Parallelisierung bei Map/Reduce



Def. 3. Parallele Anwendung der Elementfunktionen *f* und *g* → funktionale Datenpfadparallelität

> Real wird die Datenpfadparallelität durch partitionierte Kontrollpfadparallelität abgebildet!

Parallelisierungsmethoden

JavaScript und Lua

- JavaScript (und andere funktionalen Programmiersprachen) arbeiten intensiv mit Listen und Arrays
- Listen und Arrays lassen sich mittels eines Abbildungsoperators und einer Elementfunktion (Punktoperator) transformieren (map-and-reduce)

Parallelisierungsmethoden

```
local img = T{21,2,6,28,2,3,4,6,9,100,20};
function gray2color (img)
  return {r=img,g=img,b=img }
end
local img2 = img:map(gray2color);
local meanlevel = img2:map(function (p)
  return (p.r+p.g+p.b)/3
end):reduce(function (accu,level)
  return accu+level
end)
```

Bsp. 1. Bildtransformation mit map&reduce Operation

Lua Live (lua 5.3)

CLEAR RESET Point

Parallelisierungsmethoden

Lokaloperatoren

- Lokale Operatoren (Zelluläre Automaten!) sind rechenintensiver als Punktoperatoren. Um ein neues Pixel zu berechnen wird der alte Wert des Pixels sowie die alten räumlich benachbarten Werte der Pixel innerhalb einer definierten maximalen Entfernung verwendet.
- Für eine 3x3-Nachbarschaftsberechnung werden zum Beispiel Neun Datenwerte für jedes Pixel benötigt.
- Für die Berechnung lokaler Operatoren wird eine bestimmte Anzahl von parallelen Datenaustauschvorgängen benötigt → Kommunikation.
- Wenn jedes Pixel auf einem anderen Prozessor gespeichert wird ist Kommunikation teuer!

Parallelisierungsmethoden

Beispiel Lokaloperator

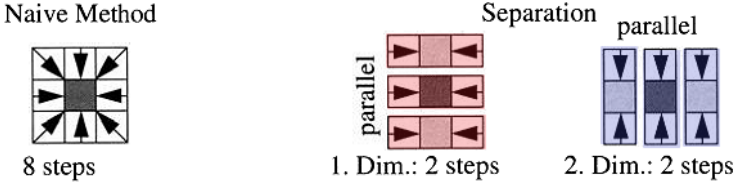


Abb. 2. Komplexitätsreduktion durch Separation der Operatoren: (1) Alle Zeilenoperationen werden parallel ausgeführt (2) Alle Spaltenoperationen werden parallel ausgeführt

Parallelisierungsmethoden

```
1  PROCEDURE sum_3x3(img: grid OF gray): grid OF INTEGER;
2  (* returns sum of local 3x3 neighborhood area *)
3  VAR res: grid OF INTEGER;
4  BEGIN
5    res:= img + MOVE.right(img) + MOVE.left(img);(*horizontal*)
6    res:= res + MOVE.down(res)  + MOVE.up(res);  (*vertical  *)
7    RETURN res;
8  END sum_3x3;
```

Abb. 3. Algorithmus: Parallele Nachbarschaftsberechnung

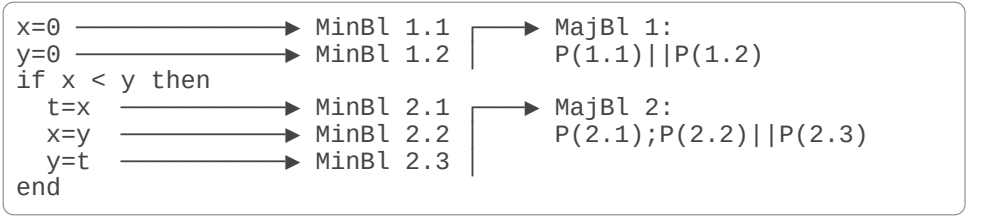
Parallelisierungsmethoden

Basisblock Optimierer und Scheduler

- Nebenläufigkeits- und Datenabhängigkeitsanalyse
- Ziele: **Reduktion von Zeitschritten, Minimierung der Latenz, Parallelisierung**

1. Basisblöcke sind Bereiche im Programmflussgraphen, die nur einen Kontrollzugang am Kopf und nur einen Kontrollausgang am Ende haben, und keine weiteren Seiteneingänge.
2. Ein Basisblock der nur aus Datenanweisungen besteht ist ein **Major-Block**. Dieser wird in elementare **Minor-Blöcke** zerlegt, die wenigstens eine Anweisung enthalten (bei einem gebunden Block der ganze Block)
3. Es werden Datenabhängigkeitsgraphen für jeden Major-Block erstellt.
4. Nicht abhängige Anweisungen werden durch einen Scheduler mit ASAP-Verhalten in einen **gebundenen Block** (paralleler Prozess) zusammengefasst.

Parallelisierungsmethoden



Bsp. 2. Basisblöcke und parallele Prozesskomposition

Funktionale Programmierung

- Determinismus macht paralleles Auswerten der Argumente und Parallelisierung bei Rekursion möglich
- Parameter von Funktionen sind datenunabhängig → Parallele Evaluierung der Funktionsargumente bei der Funktionsapplikation
- Funktionsaufrufe sind gänzlich oder partiell datenunabhängig → Parallelisierung der Funktionsaufrufe (Endrekursion, Kopfrekursion bringt kein Vorteil)

- Probleme:
  - Argumente haben teils kein ausreichendes Potential (Ausdrücke zu einfach und zu flach)
  - Große Rekursionstiefen können zu einer zu feinen Aufteilung führen → nicht gut auf Prozessoren aufzuteilen

Lua - Ausführungsmodelle

Systemprozesse

- Isolierte Programmprozesse ohne direkte Synchronisation und Datenaustausch zwischen den VMs
- Synchronsiation nur über
  - Dateien
  - Sockets (lokal, TCP, UDP)
  - Pipes (Streams)
- **Jeder Prozess führt eine VM aus** ohne (zunächst) direkte Kopplung und Kommunikation mit anderen Prozessen
  - Kommunikation über lokale Unix/Windows oder UDP/TCP Sockets, benannten Semaphoren (über Dateisystem), und geteilten Speicher
  - **Kontrollpfadparallelität**

Lua - Ausführungsmodelle

Threads

- Es werden parallele Prozesse auf **Threads** abgebildet die bestenfalls 1:1 auf den CPUs / Cores ausgeführt werden, ansonsten durch einen Scheduler im Zeitmultiplexverfahren ausgeführt werden → **Kontrollpfadparallelität**
- Prozesse können synchronisiert auf geteilte Objekte (konkurrierend) zugreifen:
  - Channel
  - Semaphore, Mutex
  - Event, Timer
  - Shared Memory Store
- Prozessblockierung blockiert den gesamten Thread (somit auch alle Fibers)
- **Jeder parallele Prozess in einem Thread läuft in eigener VM Instanz**
- Daten können *nicht* zwischen VMs ausgetauscht werden (Automatsisches Speichermanagement und GC) → nur Austausch über Serialisierung und Kopie von Daten oder geteilte Byte Datenspeicher (Shared Buffer)!

Lua - Ausführungsmodelle

Fibers und Koroutinen

- Es werden konkurrierende (aber nicht notwendig nebenläufige) Prozesse (**Koroutinen**) auf **Fibers** abgebildet die grundsätzlich in Lua nur sequenziell durch einen **Scheduler im Zeitmultiplexverfahren** ausgeführt werden.
- Diese Prozesse können synchronisiert auf geteilte Objekte (nicht nebeläufig) zugreifen:
  - Channel
  - Semaphore
  - Event, Barriere, Timer
- Prozessblockierung blockiert nur eine Koroutine
- **Alle quasi-parallelen Prozesse mit Fibers laufen in einer VM Instanz** und teilen sich den VM Datenkontext sowie besitzen den gleichen Programmkontext (direkte Teilung von Variablen und Funktionen)

Lua - Ausführungsmodelle

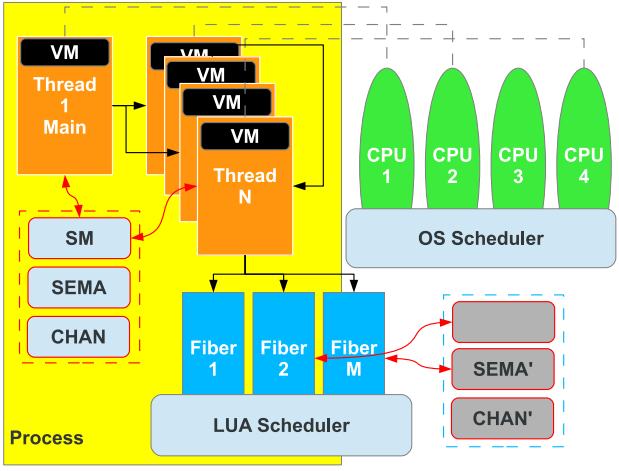


Abb. 4. Threadmodell und Kommunikation der Prozesse mit geteilten Objekten

Lua - Ausführungsmodelle

IO Paralellisierung

- Neben Threads, Fibers, und Prozessen kann auch die Parallelisierung von Ein- und Ausgabeoperation genutzt werden → **Kontrollpfadparallelität**
- Asynchrone IO wird sowohl in *nodejs* (JavaScript) als auch in Lua (*lvm*) mittels der *libuv* Bibliothek implementiert
  - Verwendung von Callback Funktionen für die Daten- und Ereignisverarbeitung
  - Anders als in JavaScript (*nodejs*) wird asynchrone IO in Lua eher seltener verwendet
- Fibers (Koroutinen) werden in Lua direkt durch die Lua VM verarbeitet
- Threads werden über die *libuv* einzelnen VM Instanzen zugeordnet
  - Verbindung der Instanzen über *libuv*
  - Jede VM Instanz hat ihre eigene Event Loop!

Lua - Ausführungsmodelle

[docs.libuv.org/en/v1.x/design.html]

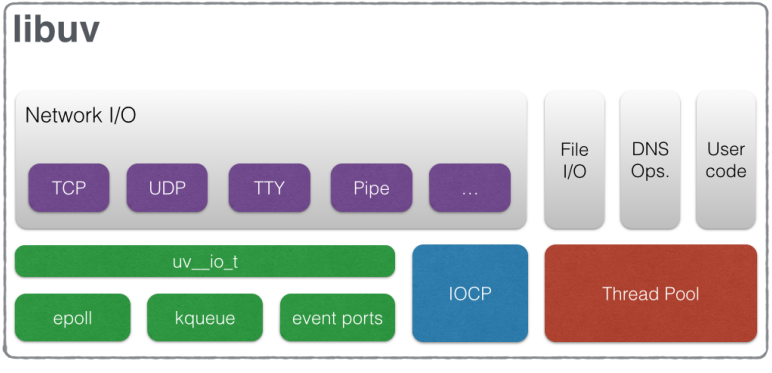


Abb. 5. Asynchrone IO mit der *libuv* Architektur

LVM

- Die Parallel LuaJit Virtual Machine (P)LVM erlaubt die parallele Programmierung auf allen drei Ebenen (Koroutinen, Threads, Prozesse)

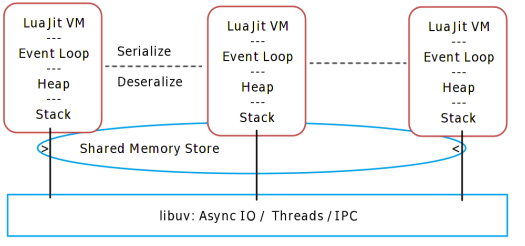


Abb. 6. Parallelisierung durch multiple VM Instanzen mit Inter-VM Kommunikation

Asynchrone Ereignisverarbeitung

- Neben der Parallelisierung der reinen Datenverarbeitung (Berechnung) kann auch eine Partitionierung von Ein- und Ausgabe bzw. der Ereignisverarbeitung erfolgen
- Bekanntes Beispiel: Geräteinterrupts mit einfachen Foreground-Background System mit zwei Prozessen:
  - P1: Hohe Priorität (Ereignisverarbeitung → Interrupthandler → Foreground Prozess)
  - P2: Niedrige Priorität (Berechnung → Hauptprogramm → Background Prozess) mit Preemption durch Ereignisverarbeitung
- Ein- und Ausgabeoperationen eines Programms können **synchron** (blockierend) oder **asynchron** (im Hintergrund verarbeitet und nicht blockierend) ausgeführt werden.

Alle Programme/Prozesse können blockieren, aber nur wenige Programmierumgebungen erlauben ein Scheduling (Multiprozessverarbeitung)

Asynchrone Ereignisverarbeitung

Beispiel Lua

- Synchrone Operationen liefern das Ergebnis in einer Datenanweisung zurück die den Programmfluss solange blockiert bis das Ereignis eintritt (Ergebnisdaten verfügbar sind).
- Dabei werden zwei aufeinander folgende Operationen sequenziell ausgeführt, und eine folgende Berechnung (nicht ereignisabhängig) erst nach den Ereignissen ausgeführt:

```
x = IO1(arg1,arg2,..);
y = IO2(arg1,arg2,..);
z = f(x,y);
```

Asynchrone Ereignisverarbeitung

- Bei der **asynchronen (nebenläufigen) Ausführung** von ereignisabhängigen Operationen wird das Ergebnis über eine **Callback Funktion** verarbeitet. Die IO Operation blockiert nicht. Vorteil: Folgende Berechnungen können unmittelbar ausgeführt werden.

```
IO1(arg1,arg2,..,function (res) x=res; end);
IO2(arg1,arg2,..,function (res) y=res; end);
z = f(x,y); -- Problem?
```

- Prozessmodell ohne Synchronisation (so wie in Lua/JS):

$$//P(IO1); //P(IO2); P$$

- Mit Synchronisation:

$$(P(IO1)||P(IO2)); P$$

Asynchrone Ereignisverarbeitung

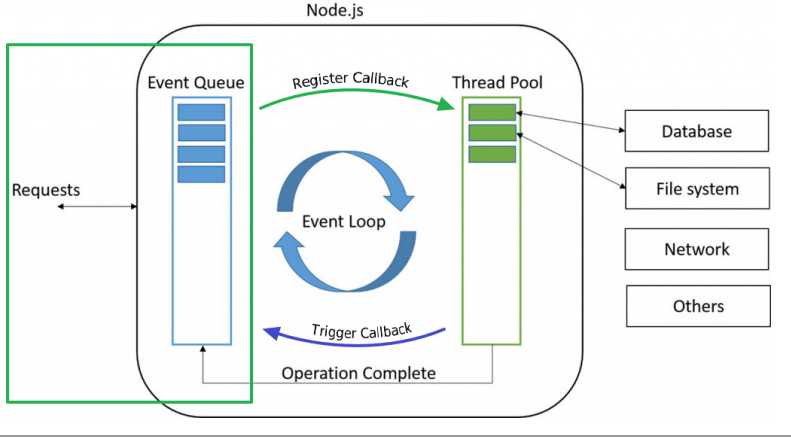


Abb. 7. Ereignisbasierte Verarbeitung von asynchronen Operationen in *lvm*: Ein Lua Thread verbunden über die Eventloop mit *N* IO Threads

Asynchrone Ereignisverarbeitung

Synchronisation

- Asynchrone Ereignisverarbeitung mit preemptiven (unterbechenden) Verhalten benötigt *explizite* Synchronisation (Locks...) zur Atomarisierung von kritischen Bereichen
- Asynchrone Ereignisverarbeitung spaltet den Kontroll- und Datenfluss auf und benötigt Daten- und Ergebnissynchronisation über **Prädikatfunktionen oder explizite Synchronisation**:

```
local x,y,z;
function P(f,x,y,z)
  if x~=nil and y~=nil then
    return f(x,y)
  else return z end
end
IO1(arg1,arg2,..,function (res) x=res; z=P(f,x,y,z) end);
IO2(arg1,arg2,..,function (res) y=res; z=P(f,x,y,z) end);
```

Def. 4. Verwendung einer Prädikatfunktion *P* zur Auflösung von Abhängigkeiten und Asynchronität / Unbekannter Ausführungsreihenfolge

Synchronisierung von Callbacks

- Neben der Prädikatfunktion kann Deasynchronisierung verwendet werden um den Programmfluss zu sequenzialisieren und zu synchronisieren
- Dazu wird eine zusätzliche Synchronisation eingeführt die die auszuführende asynchrone Funktion blockiert
- Der Kontrollfluss spaltet sich dabei auf (der Hauptfluss wird verlassen)
  - Die Callback Funktion ist der Nebenfluss der schließlich die Blockierung des Hauptflusses wieder aufhebt
  - In Lua gibt es asynchrones Schedulung (*yield*,*resume*) dafür
  - JavaScript kann dieses nur durch Modifikation der VM erreichen

Deasynchronisierung in Lua

```lua
function fooAsync (callback)
  .. callback(result) ..
end
-- Nur in Koroutine ausführbar!
function fooSync ()
  local result
  fooAsync(function (_result)
    result=_result
    coroutine.resume()
  end)
  coroutine.yield()
  return result
end
```

Def. 5. Deasync Wrapper für asynchrone Funktionen

Koroutinen in JavaScript

- In JavaScript ist zunächst kein Scheduling des Kontrollflusses möglich
- Es kann nur der gesamte Programmfluss blockieren
- Ansonsten müssen asynchrone Funktionen verwendet werden
  - Aber: Auch hier führt die VM entweder den Hauptfluss oder Callbacks (Nebenfluss über die Event Loop) aus

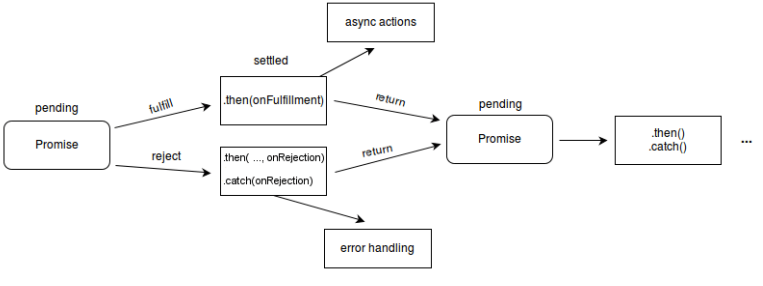
```
function fooAsync(data) {
  ... compute A1 ...
}
...compute M1 ...
setInterval(fooAsync,1000)
...compute M2...
```

Bsp. 3. Beispiel einer asynchronen Ausführung einer Berechnung durch einen Timer. Solange in M1/M2 berechnet wird findet keine Ausführung von fooAsync statt (wird verzögert)

Promises und Async/Await

- Promises wurden in EcmaScript ES2015 eingeführt um die Verschachtelung von callbacks zu linearisieren ("Callback Hölle")
- Ein Promise ist eine Funktion die die Ausführung einer Sequenz von Operationen sequenzialisiert (sequenzielle Blockierung) → Ansatz von Scheduling
- Aber: Ein Promise blockiert nicht den Hauptfluss!
- Mit *await* kann auf eine explizit gekennzeichnte *async* Funktion aber auf deren Terminierung eines Promise gewartet werden

[developer.mozilla.org]



```
var myPromise = new Promise(function (resolve, reject) {
  setTimeout(function () {
    if (Math.random()<0.5) resolve('granted');
    else reject('denied');
  }, 300);
});
myPromise
  .then(handleResolvedA, handleRejectedA)
  .then(handleResolvedB, handleRejectedB)
  .then(handleResolvedC, handleRejectedC);
```

- Koroutinen (mit Scheduling) lassen sich in JavaScript nun mittels neuer *async* Funktionen (Definition) und warten auf Terminierung mittels *await* (Applikation) implementieren

Aber: Asynchrone Funktionen und *await* Blockierungen dürfen hier nicht geschachtelt werden (anders als in Lua mit *yield*/*resume*!!)

```
var waiters=[];
function yield() {
  var getResolver = function (resolve) {
    waiters.push({resolve:resolve,event:'yield'});
  }
  return new Promise(getResolver);
}
async function coro() {
  for(;;) { ..; await yield(); .. }
}
function schedule() {
  var next = waiters.shift();
  if (next) {
    next.resolve();
  }
}
coro() .. schedule() ..
```

Koroutinen mit Kontrollflussverzweigung in JavaScript

JavaScript Live

CLEAR HELP coro1

Zusammenfassung

Funktionale Ausdrücke und Vektoroperationen bieten Möglichkeiten der Parallelisierung auf Datenpfadebene

Sequenzielle Schleifen und Rekursion bieten Möglichkeiten der Parallelisierung auf Kontrollpfadebene

Basisblockparititionierung erlaubt Parallelisierung auf Daten- und Kontrollpfadebene

Asynchrone (quasi nebenläufige) Ausführung von Funktionen bietet Parallelisierung auf Kontrollpfadebene