Verteilte und Parallele Programmierung

Mit Virtuellen Maschinen

Prof. Dr. Stefan Bosse

Universität Koblenz - FB Informatik - FG Praktiksche Informatik

Parallele Programmierung

Grundlagen der parallelen Programmierung

Unterscheidung zwischen Parallelisierung im Daten- und Kontrollfluß von Programmen

Prozessmodelle und Prozesskonstruktoren

Parallelisierungsklassen

Datenfluss

Der Datenfluss beschreibt den Fluss von Daten durch Verarbeitungs- und Speichereinheiten (Register).

Die Verarbeitungseinheiten sind Knoten eines gerichteten Graphens, die Kanten beschreiben den Datenfluss und bilden die Datenpfade. Die Verarbeitungseinheiten müssen aktiviert werden.

Kontrollfluss

Der Kontrollfluss beschreibt die temporale schrittweise Verarbeitung von Daten im Datenpfad durch zustandsbasierte selektive Aktivierung von Verarbeitungseinheiten.

Der Kontrollfluss kann durch **Zustandsübergangsdiagramme** beschrieben werden.

Parallelisierungsklassen

Programmfluss

Der Programmfluss setzt sich kombiniert aus Daten- und Kontrollfluss zusammen.

Parallelisierungsklassen

- Programmanweisungen werden Zuständen S_1,S_2,.. zugeordnet.
- Einfache Anweisungen (Berechnungen) werden jeweils einem Zustand, komplexe Anweisungen i.A. mehreren Unterzuständen zugeordnet.

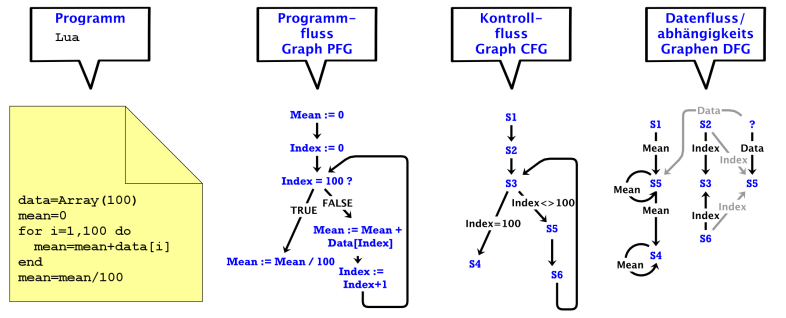


Abb. 1. Programm-, Kontroll-, und Datenflussgraphen

Parallelisierungsklassen

Datenparallelität

In vielen Programmen werden dieselben Operationen auf unterschiedliche Elemente einer Datenstruktur angewendet. Im einfachsten Fall sind dies die Elemente eines Feldes.

- Wenn die angewendeten Operationen unabhängig voneinander sind, kann diese verfügbare Parallelität dadurch ausgenutzt werden, um die zu manipulierenden Elemente der Datenstruktur auf verschiedene Prozessoren zu verteilen, so dass jeder Prozessor die Operation auf den ihm zugeordneten Elementen ausführt.
- Parallelität im Datenpfad → Feine Granularität

- Bei der Datenparallelität wird unterschieden zwischen:
  - Parallele Ausführung der gleichen Instruktion auf verschiedenen Daten (Vektorparallelität) → Vektoranweisung
  - Ausführung verschiedener Instruktionen die nur Daten verarbeiten (reine Datenanweisungen, keine Kontrollpfadverzweigungen)

Parallelisierungsklassen

- Zur Ausnutzung der Datenparallelität wurden sequenzielle Programmiersprachen zu datenparallelen Programmiersprachen erweitert. Diese verwenden wie sequenzielle Programmiersprachen einen Kontrollfluß, der aber auch datenparallele Operationen ausführen kann.
  - Z.B. µRTL: Bindung von mehreren Datenpfadanweisungen durch Kommasyntax: $x \leftarrow \varepsilon$, $y \leftarrow \varepsilon$, .. , $z \leftarrow \varepsilon$;
- Häufig werden Vektoranweisungen deklarativ und nicht prozedural imperativ beschrieben.

Parallelisierungsklassen

- Beispiel für eine deklarative Vektoranweisung und die dazugehörige imperative Anweisungssequenz in Schleifenform (Fortran 90):

```
a(1 : n) = b(0 : n - 1) + c(1 : n) ⇔
for (i=1:n)
  a(i) = b(i-1) + c(i)
end
```

Parallelisierungsklassen

- **Datenabhängigkeiten** können zwischen der linken und rechten Seite einer Datenzuweisung bestehen, so dass
  - zuerst auf alle auf der rechten Seite auftretenden Felder zugegriffen wird und die auf der rechten Seite spezifizierten Berechnungen durchgeführt werden,
  - bevor die Zuweisung an das Feld auf der linken Seite der Vektoranweisung erfolgt!
- Daher ist folgende Vektoranweisung nicht in die folgende Schleife sequenziell transformierbar:

```
a(1 : n) = a(0 : n - 1) + a(2 : n + 1) ≠
for (i=1:n)
  a(i) = a(i-1) + a(i+1)
end
```

Parallelisierungsklassen

Datenparallelität benötigt i.A. keine weitere Synchronisation

Aber Datenabhängigkeiten können zu einer impliziten oder expliziten Synchronisation führen (z.B. mit Queues)

Parallelisierungsklassen

Instruktionsparallelität

- Parallelität im Kontrollpfad (Zustandsautomat) auf Instruktions- und Prozessebene → Grobe Granularität je nach Anzahl der Instruktionen pro Prozess (Task)
- Gebundene Instruktionsblöcke sind Parallelprozesse (*Par* Prozesskonstruktor)
- Bekannt als Multithreading als Programmier- und Ausführungsmodell (z.B. *pthreads*)
- Instruktionsparallelität benötigt i.A. Synchronisation zwischen den einzelnen Prozessen

Parallelisierungsklassen

Möglichkeiten der Parallelisierung

1. Vertikale Parallelität auf Bitebene (jegliche funktionale Operation)
2. Horizontale Parallelität durch Pipelining (nur Datenströme)
3. Parallelität durch mehrere Funktionseinheiten:
   - Superskalare Prozessoren
   - Very Large Instruction Word (VLIW) Prozessoren
4. Funktionsparallelität (Evaluierung der Funktionsargumente und Rekursion)
5. Vertiakle Parallelität auf Prozess- bzw. Threadebene (Kontrollpfadebene)

Fluss und Pfadgraphen

- Ein **Datenfluss** (Graph aus Operationen oder Variablen) beschreibt den Verlauf und die Verarbeitung von Daten durch Operationen → **Verhaltensbeschreibung**
- Ein Datenpfad beschreibt die Verbindung von Komponenten zur Implementierung des Datenflusses → **Strukturbeschreibung**
- Der Kontrollfluss beschreibt die zustandsbasierte Steuerung einer Berechnung
  - Bedingte (↔ Datenfluss) Verzweigungen im Kontrollfluss
  - Schleifen im Kontrollfluss
- Der Kontrollpfad implementiert den Kontrollfluss → **Zustandsautomat**

Daten- und Kontrollpfade

Jeder generische Mikroprozessor und i.A. jedes anwendungsspezifische Digitallogiksystem läßt sich in die zwei funktionale Bereiche aufteilen:

1. Datenfluss → Datenpfade
2. Kontrollfluss → Kontrollpfad → Zustandsautomat

Datenpfad

- Führt Berechnungen durch
- Kann rein funktional (speicherlos, kombinatorisch) oder speicherbasiert mir Register-Transfer Architektur sein

Daten- und Kontrollpfade

Kontrollpfad

- Der Kontrollpfad ist immer zustandsbasiert (Speicher) und bestimmt die **Reihenfolge** von Operationen und Berechnungen

Daten- und Kontrollpfade

Parallelisierung

Der Programmfluss kann in Daten- und Kontrollfluss zerlegt werden, die jeweils im Daten- und Kontrollpfad eines Datenverarbeitungssystems verarbeitet werden.

Kontrollpfad

Parallelität auf Prozessebene (Multithreading) mit Interprozesskommunikation → Mittlere bis geringe Beschleunigung, mittlerer Overhead

Datenpfad

Parallelität auf Dateninstruktionsebene ohne explizite Kommunikation → Hohe Beschleunigung, geringer Overhead

Daten- und Kontrollpfade

Programm

- Ein (prozedurales/imperatives) Programm besteht aus mindestens einem Daten- und Kontrollpfad (oder Fluss)
- Ein Programm ist aus einer Vielzahl von Komponenten zusammengesetzt → **Komposition**
- Komposition von Datenfluss und Datenpfad:
  - **Funktionale Komposition**
  - (Sequenzielle Komposition)
  - **Parallele Komposition**

Daten- und Kontrollpfade

- Komposition von Daten- und Kontrollfluss/pfad:
  - **Sequenzielle Komposition**
  - **Parallele Komposition**

Funktionale Programmierung

> Funktionale Programmierung beschreibt grundlegend den Datenpfad. Es gibt keinen Zustand und keinen expliziten Kontrollfluß.

Funktionale Komposition

Eine Berechnung setzt sich aus der verschachtelten und verketteten Applikation von Funktionen zusammen, die jeweils aus elementaren Ausdrücken $E=\{\varepsilon_1,\varepsilon_2,..\}$ bestehen → **Funktionale Komposition** $F(\mathbf{X})=f_1(f_2(f_3..(\mathbf{X})))$

- Jede Funktion f_i beschreibt einen Teil der Berechnung.

Funktionale Komposition

- Ausdrücke bestehen aus:

1. Konstante und variable Werte `1,2.0,'c',"hello",x,y,z`
2. Einfache arithmetische Ausdrücke ε=`x+y`
3. Zusammengesetzte Ausdrücke (arithmetische Komposition) ε=`(x+1)*(y-1)*z`
4. Relationale Ausdrücke `x < 0`
5. Boolesche Ausdrücke `a and b or c`
6. Bedingte Ausdrücke liefern Werte `if x < 0 then x+1 else x-1`
7. Funktionsapplikation `f(x,y,z)`

Funktionale Komposition

- Entspricht dem mathematischen Modell was auf Funktionen und Ausdrücken gründet mit den Methoden
  - **Definition** $f(x) = \lambda x \rightarrow \varepsilon(x)$
  - **Applikation** $f(\varepsilon)$, bei mehreren Funktionsargumenten $f(\varepsilon_1, \varepsilon_2, ..)$
  - **Komposition** $f \circ g \circ h \circ .. = f(g(h(..(x))))$
  - **Rekursion** $f(x) = \lambda x \rightarrow \varepsilon(f, x)$
  - und **Substitution** $a=\varepsilon$
- Zeitliches Modell: unbestimmt ($t \mapsto 0$), *Auswertereihenfolge nicht festgelegt*

Funktionale Komposition

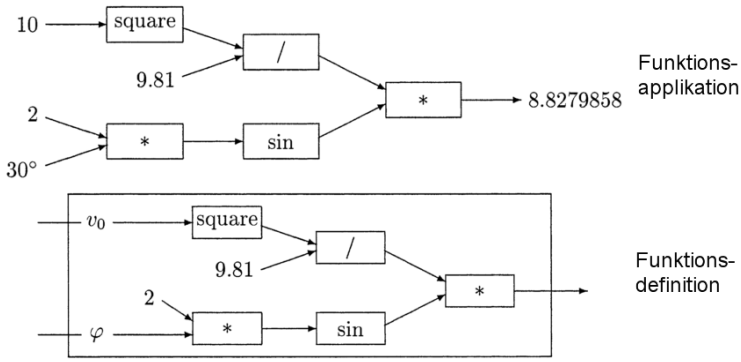


Abb. 2. Beispiel für Funktionsdefinition und Applikation

Funktionale Komposition von Systemen

> Parallelisierung durch Modellierung unterschiedlich detailierter Datenflussdiagramme, die untereinander einen hierarchischen Zusammenhang aufweisen.

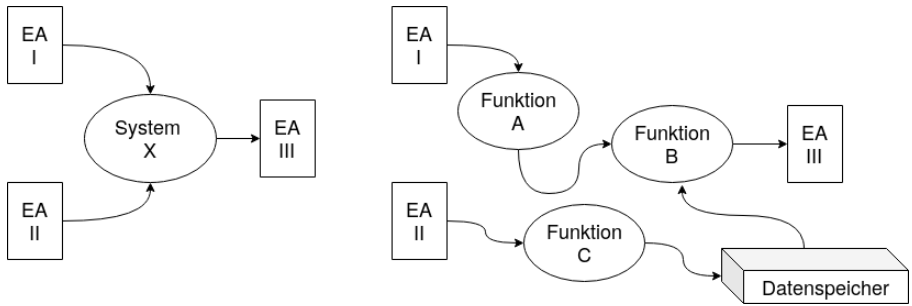


Abb. 3. (Links) Datenflüsse von Ein- und Ausgabegeräten EA eines Systems (Rechts) Komposition des Systems durch Funktionen mit inneren Datenflüssen (teils parallelisierbar)

Map&Reduce - Funktional

Grundprinzip ist Divide&Conquer Ansatz. Eine Datemenge $\boldsymbol{D}$ wird in Elemente $\mathbf{d}_i$ zerlegt (oder Partitionen). Die können unabhängig voneinander mittels einer Funktion F in Ausgabedaten $\boldsymbol{D}^o$ transformiert werden (Mapping). Anschliessend können diese Elemente mit einer Funktion G reduziert werden (Reduce).

```
l=[v1,v2,..]
l'=map(l,x => f(x))
l''=reduce(l',(a,b) => g(a,b))
```

Map&Reduce - Lua

- Im Idealfall findet ein 1:1 Mapping statt, d.h. alle Berechnungen mit *F* finden parallel in eigenen Prozessen statt
- Realistischer ist die Verarbeitung von Partitionen von Prozessen, d.h. eine Mischung aus Sequenz und paralleler Verarbeitung (mit sog. Worker Prozessen)
- Die Worker Prozesse werden entweder vorab als Pool oder zur Laufzeit gestartet und verarbeiten partitionierte Daten
- Es kann eine funktionale Kette aufgebaut werden die den Datenfluss beschreibt.

Map&Reduce - Lua

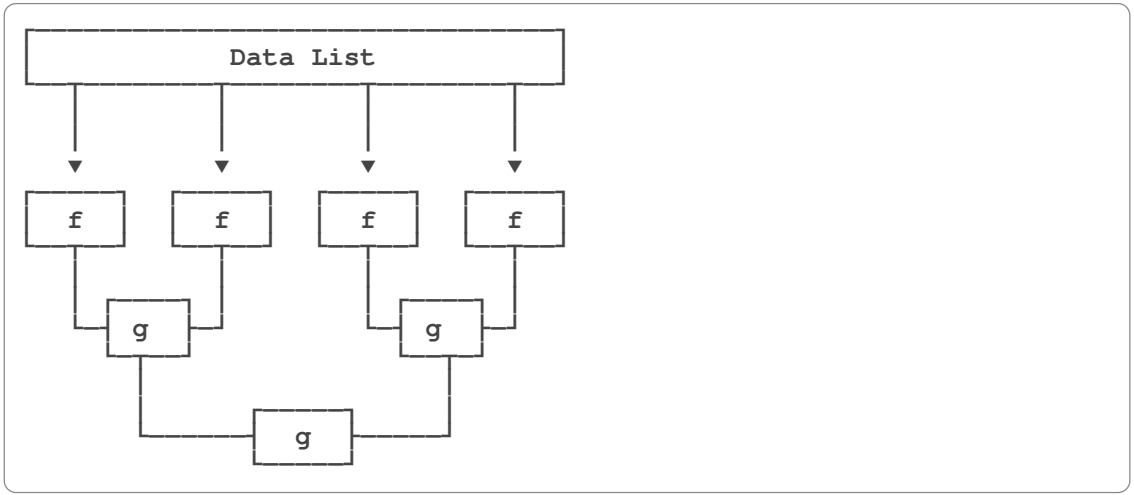


Abb. 4. Datenflussgraph von Map&Reduce (vertikal) und (mögliche) Parallelisierung (horizontal)

Map&Reduce - Lua

```
local options = {workers = 2, remap=true
-- Input data
local data = {34,35,36,37,38,39,40,41}
-- Worker Function
local function worker (set,id)
  -- set ist ganze Partition!
  local results = T{}
  for i = 1,#set do
    results:push(fib(set[i]))
  end
  return results
end
```

```
-- Parallel Processing
local Parallel = require('parallel')
local p = Parallel:new(data,options)
function sum (x,y) return x+y  end
p:time():
  map(worker):
  apply(function (r)
    print(r:print()) end):
  reduce(sum):
  apply(function (r)
    print(r:print()) end):
  time()
```

Bsp. 1. Beispiel eines parallelisierten funktionalen Map&Reducde Algorithmus (mapchunks=true)

Map&Reduce - Lua

```
local options = {workers = 2, remap=true
-- Input data
local data = {34,35,36,37,38,39,40,41}
-- Worker Function
local function worker (n,index,id)
  -- hier jetzt Aufruf mit einzelnen Dat
  -- zerlegung und Zusammenfügung
  -- einer Partition (chunk) dann in par
  return fib(n)
end
```

```
-- Parallel Processing
local Parallel = require('parallel')
local p = Parallel:new(data,options)
function sum (x,y) return x+y  end
p:time():
  map(worker):
  apply(function (r)
    print(r:print()) end):
  reduce(sum):
  apply(function (r)
    print(r:print()) end):
  time()
```

Bsp. 2. Beispiel eines parallelisierten funktionalen Map&Reducde Algorithmus (mapchunks=false)

Sequenzielle Komposition

> Eine imperative Berechnung setzt sich aus einer Sequenz **I**=(i_1 , i_2 , ..) von elementaren Anweisungen **A**={a_1,a_2,..} zusammen →
> **Sequenzielle Komposition** i_1 ; i_2 ; ...

- Die Anweisungen werden sequenziell in *exakt* der angegebenen Reihenfolge ausgeführt

- Die Menge der Anweisungen **A** lässt sich in folgende Klassen unterteilen:

1. Arithmetische, relationale, und boolesche **Ausdrücke** (in Verbindung mit 2./3.) `x+1, a*(b-c)*3, x<(a+b), ...`
2. **Datenanweisungen**: Wertzuweisungen an Variablen `x := a+b`
3. **Kontrollanweisungen** wie bedingte Verzweigungen und Sprünge (Schleifen)

```
if a < b then x := 0 end
while x < 100 do  x:=x+1 end
```

Sequenzielle Komposition

- Man fasst die Sequenz **I** als ein imperatives **Programm X** == Ausführungsvorschrift zusammen.
- Die **Ausführung** des Programms (statisch) bezeichnet man als **Prozess** (dynamisch)
- Ein Prozess befindet sich aktuell immer in einem **Zustand** σ einer endlichen Menge von Zuständen $\Sigma=\{\sigma_1, \sigma_2, ..\}$.
- Der gesamte Zustandsraum Σ des Prozesses setzt sich aus dem
  - **Kontrollzustand** $s \in \mathbf{S}=\{s_1, s_2, ..\}$, und dem
  - **Datenzustand** mit der Menge aller Speicherzustände (Daten) $d \in \mathbf{D}=\{d_1, d_2, ..\}$ zusammen.

Sequenzielle Komposition

- Der Fortschritt eines sequenziellen Programms bedeutet ein Änderung des Kontroll- und Datenzustandes → **Zustandsübergänge**
- Die Zustandsübergänge können durch ein Zustandsübergangsdiagramm dargestellt werden.

Das Prozeßmodell

Es werden zwei Prozesstypen unterschieden:

Sequenzieller Prozeß

Ein einziger Programm- und Kontrollfluß mit strikter sequenzieller Ausführung von Anweisungen

Paralleler Prozeß

Es gibt zwei oder mehr parallel ausgeführte Programm- und Kontrollflüsse (Ausführung kann zeitlich überlappend sein, muss aber nicht → Scheduling)

Sequenzieller Prozess

Ausführungszustände

- Ein Prozess *P* besitzt einen "makroskopischen" **Metaausführungszustand** *ps*. Die Basismenge der Ausführungszustände **PS** ist ("Milestones"):
  - Start/Bereit aber noch nicht rechnend (`START`)
  - Rechnend (`RUN`)
  - Terminiert und nicht mehr rechnend (`END`)
- Dazu kann es eine erweiterte Menge **PS***an Metaausführungszuständen geben:
  - Auf ein Ereignis wartend (`AWAIT`), z.B. Verfügbarket von Eingabedaten
  - Blockiert (`BLOCKED`), z.B. P1(*suspend*)-P2(*wakeup*) Paare
  - Rechenbereit aber nicht rechnend (nach Ereignis) (`READY`)

$$PS = \{\texttt{START}, \texttt{RUN}, \texttt{END}\}$$
$$PS^* = \{\texttt{AWAIT}, \texttt{BLOCKED}, \texttt{READY}\}$$
$$ps(P) \in PS \cup PS^*$$

- Befindet sich der Prozess im Ausführungszustand `RUN` (also ausführend), dann werden strikt sequenziell die Instruktionen des Programs $\boldsymbol{I}=\{i_1,i_2,..,i_n\}$ ausgeführt.
- Dabei ist jede Instruktion eine Anweisung aus der endlichen Anweisungsmenge $\boldsymbol{A}=\{a_1,..\}$ der Maschine (z.B. Bytecode Anweisung `add(x,y)`).
- Ein Wechsel des Ausführungszustandes führt nicht zu einer Änderung der Instruktionsreihenfolge.

Sequenzieller Prozess: Zeitliches Modell

- Ein Prozess führt die Anweisungen der Reihe nach aus. Das symbolische Semikolon teilt den Anweisungen eigene Ausführungsschritte (Zeitpunkte t_i) zu.

$$i_1 : t_1 \mapsto i_2 : t_2 \mapsto i_3 : t_3 \mapsto .. \mapsto i_n : t_n \text{ mit}$$
$$t_1 < t_2 < t_3 < .. < t_n$$
$$T = \sum_i t_i - t_{i-1}$$

- Die gesamte Ausführungszeit *T(p)* eines sequenziellen Prozesses *p* ist die Summe der Ausführungszeiten der Elementarprozesse

Sequenzieller Prozess

Prozessblockierung

- Das Konzept der Prozessblockierung ist elementar für kommunizierende Prozesse
- Kommunikation: Lesen und Schreiben von Daten, Ein- und Ausgabe, Netzwerknachrichten, usw.
- Befindet sich ein Prozess im Ausführungszustand `AWAIT` wartet dieser auf ein Ereignis:
  - Daten sind verfügbar
  - Zeitüberschreitung
  - Geteilte Ressource ist frei
  - Synchronisation mit anderen Prozessen

Sequenzieller Prozess

- Bei einem blockierten Prozess schreitet der Kontrollfluss nicht weiter voran (keine Terminierung der aktuellen Operation)
- Die Ausführung einer Operation (Instruktion) kann verzögert werden bis das zu erwartende Ereignis eintritt

> Das Ausführungsmodell von Lua unterstützt die Blockierung des Programmflusses

Koroutinen

- In Koroutinen (Fiber) z.B. mittels der *yield* Anweisung
- Der Programmfluß wird über einen Scheduler an eine andere Koroutine übergeben
- Schließlich wird die Programmausführung der ursprünglichen Koroutine nach der *yield* Anweisung wieder fortgesetzt (durch *resume* Anweisung)

Koroutinen

- Ebenso auch bei Ein- und Ausgabeoperationen (IO: *read*, *write*)
- In Threads ebenfalls via *yield* und IO Operationen
  - Aber hier findet das Scheduling außerhalb der VM statt (multiple VM Instanzen)

Symmetrische Koroutinen

Das Scheduling der Koroutinen erfolgt semiautomtisch und es gibt nur die *yield* Operation (Reaktivierung von Koroutinen implizit durch Scheduler).

Asymmetrische Koroutinen

Die Suspendierung und Reaktivierung von Koroutinen erfolgt explizit (*yield* und *resume* Operationen).

Koroutinen

> Tatsächlich gibt es in Lua auf programmatischer Ebene aber nur **einen einzigen Kontrollfluß** und Pfad (eine VM Instanz)!

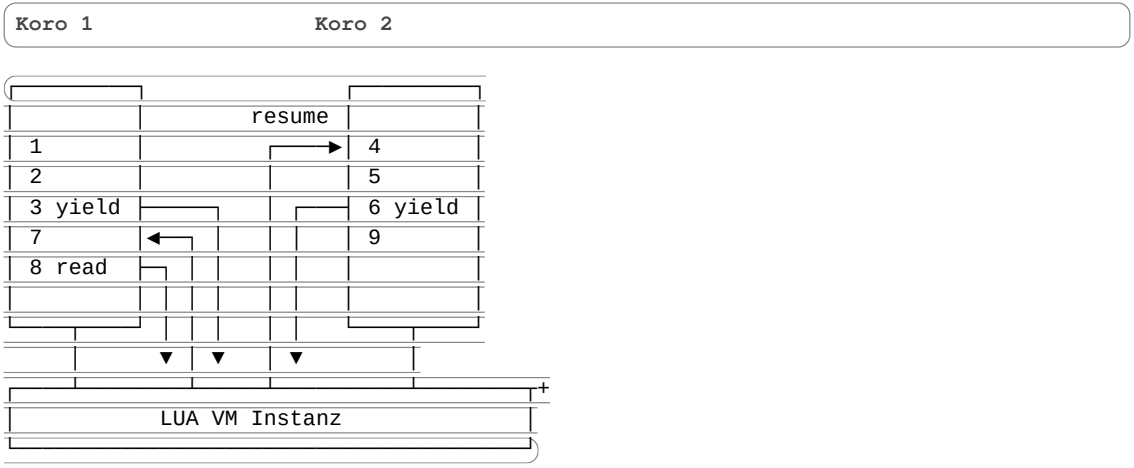


Abb. 5. Zwei Koroutinen im Wechsel; eine VM Instanz

Koroutinen

```
co1 = coroutine.create(function ()
          for i=1,10 do
            print("co1", i)
            coroutine.yield()
          end
        end)
co2 = coroutine.create(function ()
          for i=1,10 do
            print("co2", i)
            coroutine.yield()
          end
        end)
coroutine.resume(co1)
coroutine.resume(co2)
...
```

Bsp. 3. Asymmetrische Koroutinen in Lua

Sequenzieller Prozess

Fibers

Über die Eventloop Bibliothek libuv (multithreaded) und Lua Bindungen luv stehen neben Threads auch Fibers zur Verfügung. Luv Fibers sind symmetrische Koroutinen, werden aber auch strikt sequenziell und nicht präemptiv verarbeitet!

Achtung: Die klassischen Lua Koroutinen sind inkompatibel zu Luv (da alle IO Operationen über die uv Bibliothek laufen). In *lvm* gibt es eine alternative Implementierung *corof* die auf Fibers aufbaut. Diese ist automatisch aktiviert (als Ersatz unter dem gleichen Namen *coroutine*).

Fibers

```
co1 = luv.fiber.create(function ()
           for i=1,5 do
             print("co1", i)
             luv.fiber.yield()
           end
           print("co1 finished.")
         end)
co2 = luv.fiber.create(function ()
           for i=1,5 do
             print("co2", i)
             luv.fiber.yield()
           end
           print("co2 finished.")
         end)
co1:ready()
co2:ready()
co1:join()
co2:join()
print ("Done.")
```

Parallel LuaJit Virtual Machine (LVM)

CLEAR GET RESET RESTART EXIT co1 coff2

Parallele Prozesse

Ausgangspunkt sind sequenzielle Prozesse

Ein sequenzieller Prozess *Ps* besteht aus einer Sequenz von Anweisungen **I**=(i_1 ; i_2 ;..) (Berechnungen), die schrittweise und nacheinander ausgeführt werden:
$Ps=P(i_1 ; i_2 ; .. ; i_n) \Rightarrow p_1 \mapsto p_2 \mapsto .. \mapsto p_n$

Parallele Komposition

Datenverarbeitungssystem als paralleler Prozess kann aus einer Vielzahl nebenläufig ausgeführter Prozesse $P=\{P_1, P_2, P_3,...\}$ bestehen

Parallele Komposition $P=P_1 \parallel P_2 \parallel P_3 \parallel ..$

Paralleler Prozess

Prozesskonstruktor

$$P = (P_1 \parallel P_2 \parallel .. \parallel P_n) \quad \Leftrightarrow$$

```
PAR
  i1
  i2
  ..
  in
```

Lua

```
Par({
  function (pid) .. end,
  function (pid) .. end,..
},{
  -- shared environment
  v = ε, ..
})
```

Paralleler Prozess

Ausführungsreihenfolge

- Achtung! Wenn keine Kommunikation (Synchronisation) zwischen den Prozessen stattfindet ist die Ausführung und das Endergebnis (der Berechnung) gleich der sequenziellen Ausführung der Prozesse:

$$P_1 \parallel P_2 \parallel .. \parallel P_n \Rightarrow P_1; P_2; ..; P_n$$
$$P_1 \parallel P_2 \Rightarrow P_1; P_2 \Rightarrow P_2; P_1 !$$

Paralleler Prozess: Zeitliches Modell

$$P = P_1 \parallel P_2 \parallel .. \parallel P_n$$

$$P_1 = (i_{1,1} : t_{1,1}; i_{1,2}, : t_{1,2}; ..) \cdots P_n = (i_{n,1} : t_{n,1}; i_{n,2} : t_{n,1}; ..), \text{ mit}$$

$$t_{i,1} < t_{i,2} < .. < t_{i,m} \forall \text{Prozess mit } m \text{ Instr. und}$$

$$t(P) = [t_1, t_2] = [t_{1,1}, t_{1,q}] \cup [t_{2,1}, t_{2,r}] \cup .. \cup [t_{n,1}, t_{n,s}]$$

$$t_1 = \min\{t_{1,1}, t_{2,1}, .., t_{n,1}\}$$

$$t_2 = \max\{t_{1,q}, t_{2,r}, .., t_{n,s}\}\ T = t_2 - t_1$$

Prozessflussdiagramme

- Der Prozessfluss in einem parallelen System kann durch Übergangs- und Terminierungsdiagramme dargestellt werden. (wie bereits im seq. Fall gezeigt wurde).
- Eingehende Kanten (Pfeile) des Graphen beschreiben den Start eines Prozesses, ausgehende Kanten die Terminierung.

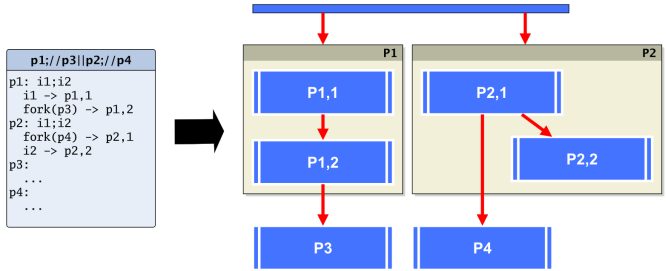


Abb. 6. Beispiel eines Prozessflussdiagramms für vier Prozesse. P_1 startet P_3, und P_2 spaltet P_4 ab. (mit $P_{\_,i}$ = *Instruktion*(P,i))

Paralleler Prozess

Prozesserzeugung und Synchronisation

- Ein neuer paralleler Prozess kann durch den entsprechenden Prozesskonstruktor erzeugt werden
- Geschieht dies innerhalb eines Prozesses P_i so wird dessen Prozessausführung blockiert *bis* alle Teilprozesses des parallelen Prozessen terminiert sind!

Forking

- Ein neuer Prozess P_j kann mittels Forking erzeugt werden (Aufspaltung des Kontrollflusses, so auch von einem anderen Prozess P_i) ohne dass ein ausführender Prozess auf die Terminierung des Parallelprozesses warten muss:

$$P_1; /\!/ P_2; P_3 = P_1; P_3 \parallel P_2$$
$$/\!/ P_1; /\!/ P_2; P_3 = P_1 \parallel P_2 \parallel P_3$$

Lua

```
Fork({
  function (pid) .. end,
  function (pid) .. end, ..
},{
  -- shared environment
})
```

Parallele Prozesse in Lua

Parallel LuaJit Virtual Machine (LVM)

CLEAR GET RESET RESTART EXIT Par Fork

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 (Automatisches 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

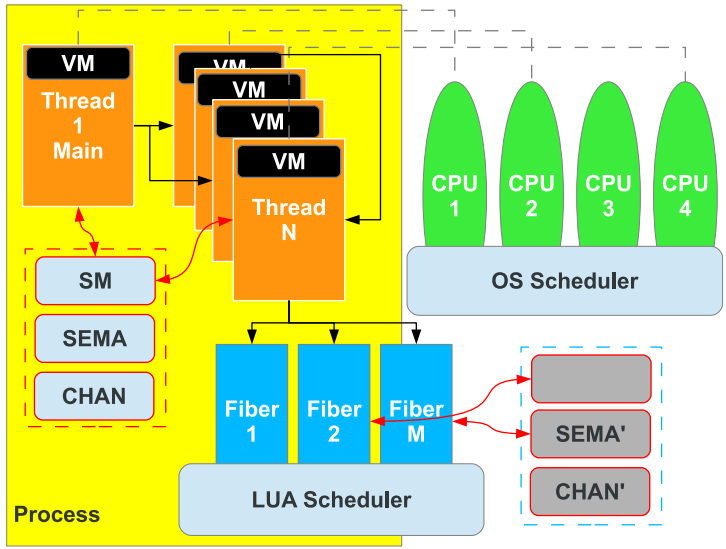


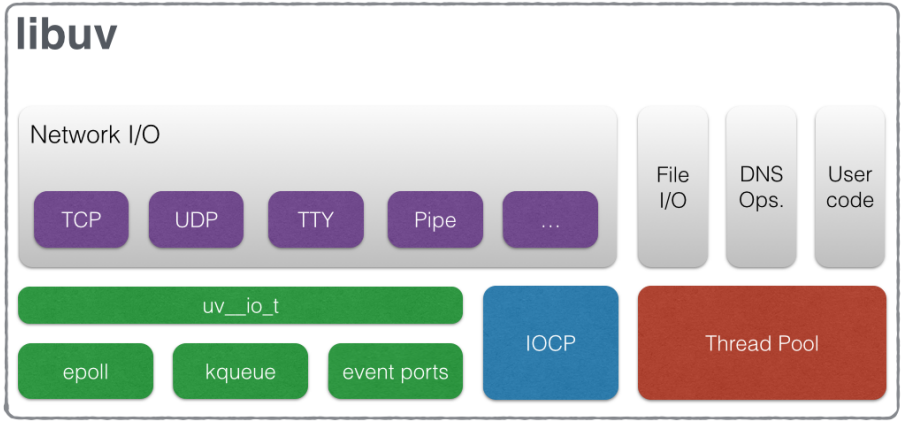
Abb. 7. Threadmodell und Kommunikation der Prozesse mit geteilten Objekten

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. 8. 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)

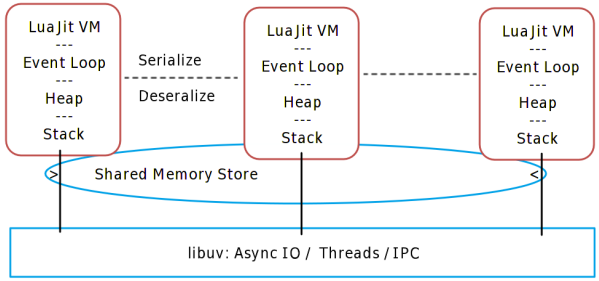


Abb. 9. Parallelisierung durch multiple VM Instanzen mit Inter-VM Kommunikation

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

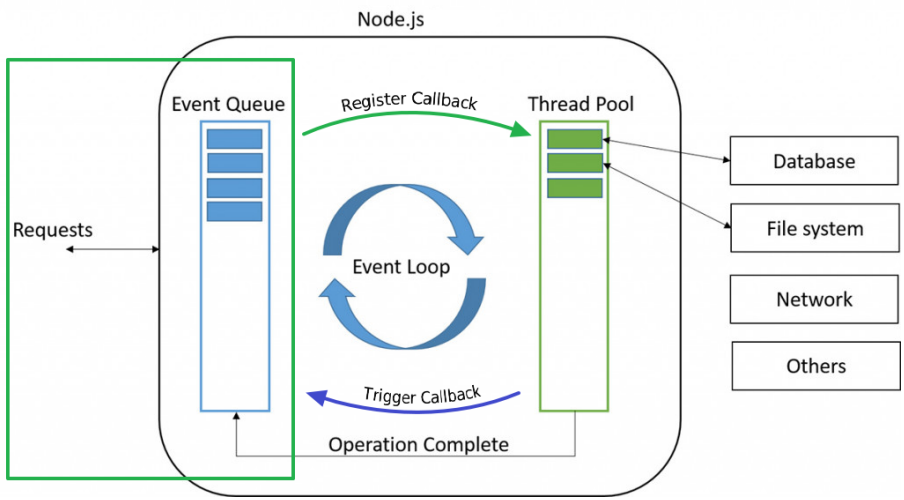


Abb. 10. Ereignisbasierte Verarbeitung von asynchronen Operationen in *lvm*: Ein Lua Thread verbunden über die Eventloop mit *N* IO Threads

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. 1. 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

```
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. 2. Deasync Wrapper für asynchrone Funktionen

Zusammenfassung

Parallele Prozesse werden aus nebenläufig oder verwoben ausgeführten sequenziellen Prozessen gebildet

Prozesse besitzen drei wesentliche Ausführungszustände: Bereit, Rechnend, Terminiert

Prozessynchonisation kann durch Prozessflussgraphen (PFG) dargestellt werden

Ein nachfolgender Prozess (Knoten im PFG) wird erst gestartet wenn der vorherige terminert

Prozessblockierung ist wesentliche Eigenschaft von synchronisierenden Prozessen